Write a Program to add two numbers :

```
public class Addition
{
        public static void main(String[] args)
        {
                int x = 100;
                int y = 200;
                int z = x + y;
                System.out.println(z);
        }
}
```

Note : Here we are getting the output 300 but, It is not user-friendly.

How to provide user-friendly message :

```
public class Sum
{
        public static void main(String[] args)
        {
                int x = 12;
                int y = 24;
                int z = x + y;
                System.out.println("Sum is :",z);  //error [No Suitable method]
        }
}
```

In order to provide user-friendly message, We should use + operator.

Behavior of '+' Operator :

While working with '+' operator if any of the operand is String then this '+' operator will behave as String Concatenation Operator.

Example :

1 + 1 = 2 [Here '+' Operator will behave as a Arithmetic Operator]

"2" + 1 = 21 [Here '+' Operator will behave as a String concatenation Operator]

"2"+ "2" = 22 [Here '+' Operator will behave as a String concatenation Operator]

12 + "1" = 121 [Here '+' Operator will behave as a String concatenation Operator]

Sum.java

```
public class Sum
{
        public static void main(String[] args)
        {
                int x = 12;
                int y = 24;
                int z = x + y;
                System.out.println("Sum is :"+z);
        }
}
```

WAP to add two numbers without using 3rd Variable :

```
public class AdditionWithout3rdVariable
{
        public static void main(String[] args)
        {
                int x = 100;
                int y = 200;
                System.out.println("Sum is :"+x+y); //100200
                System.out.println(+x+y);//300
                System.out.println(""+x+y); //100200
                System.out.println("Sum is :"+(x+y));//300
        }
}
```

Command Line Argument :

Whenever we pass any argument to the main method then it is called Command Line
Argument.

Example :

```
public void m1(int x)  -> Argument
{
}
```

public static void main(String [] args) -> Command Line Argument
{

}

By using Command Line Argument we can pass some value at runtime by using command(CMD);

The advantage of command line argument is, Single time compilation and number of times execution with different values.

```
public class Command
{
        public static void main(String[] args)
        {
                System.out.println(args[0]);
        }
}
```

javac Command.java
java Command 90 80 23

It will print 90

```
public class CommandName
{
        public static void main(String[] x)
        {
                System.out.println(x[1]);
        }
}
```

java CommandName Virat Bumrah Rohit

It will print Bumrah

```
public class CommandFullName
{
        public static void main(String[] args)
        {
                System.out.println(args[0]);
        }
```

}

java CommandFullName "Virat Kohli"

It will print Virat Kohli

How to print the length of the Array using Command Line Argument ?

We have length property available in Array class so, with any array variable we can use this length property to print the length of the array.

```java
public class ArrayLength
{
        public static void main(String[] args)
        {
                System.out.println("Array Length is :"+args.length);  //length property
        }
}
```

java ArrayLength 100 200 300
Output is : Array Length is 3

java ArrayLength 100 200 300 400
Output is : Array Length is 4

WAP to add two numbers by using Command Line Argument :

```java
public class CommandAddition
{
        public static void main(String[] x)
        {
                System.out.println(x[0] + x[1]);
        }
}
```

java CommandAddition 100 200

Output = 100200 [Here x is String type so, '+' operator will
             work as a String concatenation Operator]

How to convert a String value into integer :

In java.lang package, there is a predefined class called Integer, this Integer class contains a predefined static method called pareInt(String x). This parseInt(String x) method accept String as a parameter and return type is int.

Example :

```
public class Integer
{
  public static int parseInt(String x)
  {
     return an int value (convert String into int value)
  }
}
```

```
public class CommandAdd
{
        public static void main(String[] args)
        {
                String x = args[0];
                String y = args[1];

                //Converting String into int value
                int a = Integer.parseInt(x);
                int b = Integer.parseInt(y);

                System.out.println("Sum is :"+(a+b));
        }
}
```

WAP to find out the Square of the number by using Command Line Arg
----
```
public class GetSquare
{
        public static void main(String[] args)
        {
                int num = Integer.parseInt(args[0]);
                System.out.println("Square of "+num+" is :"+(num*num));
        }
}
```

WAP to find out the area of rectangle :

```java
public class AreaOfRect
{
        public static void main(String[] args)
        {
                int length = Integer.parseInt(args[0]);
                int breadth = Integer.parseInt(args[1]);

                int area = length * breadth;

                System.out.println("Area of Rectangle is :"+area);
        }
}
```

WAP to find out the area of circle :

```java
public class AreaOfCircle
{
        public static void main(String[] args)
        {
                //Converting String to double
                double radius = Double.parseDouble(args[0]);

                final double PI = 3.14;
                double area = PI * radius * radius;
                System.out.println("Area of Circle is :"+area);

        }
}
```

Note : While Working with Command line Argument program, If we are passing the expected
value at runtime then we will get an exception
java.lang.ArrayIndexOutOfBoundsException

WAP in java to pass some value from command line argument based on the following criteria :

If the array length is 0 : It should print length is 0
If the array length is 1 : It should find the cube of the number
if the array length is 2 : It should print sum of the number

```java
class ArrayLengthCalculate
{
        public static void main(String[] args)
```

```
        {
                int length = args.length;

                if(length == 0)
                {
                        System.out.println("No value from command Line argument");
                }
                else if(length ==1)
                {
                        int num = Integer.parseInt(args[0]);
                        System.out.println("Cube of "+num+" is :"+(num*num*num));
                }
                else if(length ==2)
                {
                        int a = Integer.parseInt(args[0]);
                        int b = Integer.parseInt(args[1]);
                        System.out.println("Sum of "+a+" and "+b+" is :"+(a+b));
                }
        }
}
```

20-09-2024

How Integer.parseInt(String x) works internally ?

```
class Integer
{
        public static int doSum(int x, int y)
        {
                return (x+y);
        }

        public static int getSquareOfTheNumber(int num)
        {
                return (num*num);
        }

}


public class Calculation
{
        public static void main(String[] args)
```

```
        {
                int result = Integer.doSum(10,20);
                System.out.println("Sum is :"+result);


                result = Integer.getSquareOfTheNumber(5);
                System.out.println("Square is :"+result);
        }
}
```

How to convert String into float :
----
float x = Float.parseFloat(String x);

How to convert String into double :
----
double x = Double.parseDouble(String x);
===
Working with Eclipse IDE :

IDE stands for "Integrated Development Environment".

By using Eclipse IDE, We can develop, compile and execute the java
program in a single window.

The main purpose of Eclipse IDE, to reduce the development time, Once development time is
reduced then automatically the cost of the project will be reduced.

What is a package?

A package is folder in windows. In java to create a package we are using package keyword, It
must be first statement of the program.

Example :
----
package basic;

public class Hello
{
}

How to compile : javac -d . Hello.java
```

[javac space -d space . space FileName.java]

Here one folder will be created called basic (package name ) and Hello.class file will be automatically placed in the basic folder.

Note : if we write package name like com.tcs.shopping the 3 folders will be created. First folder name is com, inside that com folder tcs folder will be created and inside that tcs folder shopping folder will be created then then finally the .class file will be placed inside shopping folder.

Types of Packages :
---------------------------
1) Predefined OR Built-in package : The packages which are created by java software people for arranging the programs are called predefined package.

Example : java.lang, java.util, java.io, java.sql, java.net and so on

2) Userdefined Package OR Custom package : The packages which are created by user for arranging the user-defined programs are called user-defined package.

Example :
basic;
com.ravi.basic;
com.tcs.online_shopping;
===================================================================
How to execute Command Line Argument Program by using Eclipse IDE :
-----------------------------------------------------------------
WAP to add two numbers by using command Line Argument :
-------------------------------------------------------
package com.ravi.basic;

```
public class CommandAdd
{
        public static void main(String[] x)
        {
            int a = Integer.parseInt(x[0]);
            int b = Integer.parseInt(x[1]);

            System.out.println("Sum is :"+(a+b));


        }

}
```

Steps to execute the command Line Argument Program using Eclipse IDE

Right click on the program -> Run As -> Run configuration -> Check your main class name -> select argument tab -> pass the appropriate value -> Run

==

Naming Convention in java :

While writing the Java code we should always follow the naming convention given by java language. It has two benefits

 a) Industry Standard Code (Industry Acceptable Code)
 b) Enhancing the Readability of the code

1) How to write a class in java :

In order to write a class in java, We should follow Pascal Naming Convention, According to this each word first letter must be capital and It should not contain any space. In java a class represents noun.

Example :
ThisIsExampleOfClass
Integer
String
System
ArrayIndexOutOfBoundsException
DataInputStream
BufferedReader

2) How to write a method in java:

In order to write a method in java we should follow camel case naming convention. According to this second word onwards, first letter of each word must be capital. In java a method represents verb.

Example :
thisIsExampleOfMethod()
read()
readLine()
parseInt()
charAt()
next()

nextLine()

3) How to write a variable in java :

In order to write a variable in java we should follow camel case naming convention. According to this second word onwards, first letter of each word must be capital.

Example :
thisIsExampleOfVariable

rollNumber;
customerBill;
employeeSalary;
employeeNumber;

4) How to write final and static variable :

In order to write final and static variable we should use snake_case naming convention.According to this all characters must be capacital and between each wordwe must have _ symbol.

Example :

Integer.MIN_VALUE
Integer.MAX_VALUE

5) How to write a package :

A package must be written in lower case only. Generally it is reverse of company name.

com.nit.basic
com.tcs.introduction
com.wipro.shopping
===
Tokens in java :

Token is the smallest unit of the program which is identified by the compiler.

Without token we can't complete statement or expression in java.

Token is divided into 5 types in java

 1) Keyword

2) Identifier
3) Literal
4) Punctuators (Seperators)
5) Operator

Keyword

A keyword is a predefined word whose meaning is already defined by the compiler.

In java all the keywords must be in lowercase only.

A keyword we can't use as a name of the variable, name of the class or name of the method.

true, false and null look like keywords but actually they are literals.

As of now, we have 67 keywords in java.

Identifiers :

A name in java program by default considered as identifiers.

Assigned to variable, method, classes to uniquely identify them.

We can't use keyword as an identifier.

Ex:-

class Fan
{
   int coil ;

   void start()
   {
   }
}

Here Fan(Name of the class), coil (Name of the variable) and start(Name of the Method) are identifiers.

Rules for defining an identifier :
----
1) Can consist of uppercase(A-Z), lowercase(a-z), digits(0-9), $ sign, and underscore (_)

2) Begins with letter, $, and _
3) It is case sensitive
4) Cannot be a keyword
5) No limitation of length

Literals :

It is a constant value which we are assigning to a variable.

In java we have 5 types of Literals :

 1) Integral Literal
 2) Floating point Literal
 3) Boolean Literal
 4) Character Literal
 5) String Literal

Note : null is also literal which we assign to reference variable.

Integral Literal :

If any numberic literal does not contain decimal or fraction then it is called Integral literal.
Example : 12, 89, 56 and so on

In integral literal we have 4 data types :
  a) byte (8 bits)
  b) short (16 bits)
  c) int (32 bits)
  d) long (64 bits)

An integral literal we can represent in 4 different forms :
   a) Decimal Literal (Base 10)
   b) Octal Literal (Base 8)
   c) Hexadecimal Literal (Base 16)
   d) Binary Literal (Base 2)

Decimal Literal :

By default our integral literal in Decimal Literal only. Here Base is 10 hence it accepts 10 digits i.e 0 to 9.

Example : int x = 20;

```
        int y = 50;
           int z = 100;
```

Octal Literal :

If any integral literal starts with 0 (zero) then it will become
octal literal. Here Base/Radix is 8 so, It will accept 8 digits i.e.
0 to 7.

```
Example : int x = 0777; //Valid
         int y = 015; //Valid
            int z = 018; //Invalid [Here digit 8 is out of range]
```

Hexadecimal Literal :

If any integral literal starts with 0X (zero capital X) OR 0x (Zero small x) then it will become
hexadecimal Literal. Here Base is 16 so it will accept 16 digits i.e 0 to 9 and a to f.

```
Example :  int x = 0X15; //Valid
          int y = 0xadd; //Valid
            int z = 0Xface; //Valid
            int a = 0xage; //Invalid [Here g is out of the range]
```

Binary Literal :

It is available from Java 7 version. If any integral literal starts with 0B (zero capital B) OR 0b
(zero small b) then it will become
Binary Literal. Here base is 2 so, It will accept only 2 digits i.e 0 and 1.

```
Example : int x = 0B101; //Valid
         int y = 0b111; //Valid
            int z = 0B112; //Invalid
```

As a programmer we can represent an integral literal in four different forms like decimal, octal,
hexadecimal and binary but to get the output JVM will always convert these numbers in decimal
format only.

Program on Octal Literal :

Test.java

```java
public class Test
{
        public static void main(String [] args)
        {
          int x = 015;
          System.out.println(x); //13

        }
}
```

Program on Hexadecimal Literal :

```java
public class Test
{
        public static void main(String [] args)
        {
          int x = 0Xadd;
          System.out.println(x);  //2781

        }
}
```

Program on Binary Literal :

```java
public class Test
{
        public static void main(String [] args)
        {
          int x = 0B111;
          System.out.println(x);

        }
}
```

By default every integral literal is of type int only. byte and short are below than int so we can assign integral literal(Which is by default int type) to byte and short but the values must be within the range. [for Byte -128 to 127 and for short -32768 to 32767]

Actually whenever we are assigning integral literal to byte and short data type then compiler internally converts into corresponding type.

byte b = (byte) 12; [Compiler is converting int to byte]

short s = (short) 12; [Compiler is converting int to short]

In order to represent long value we should use either L OR l (Capital L OR Small l) as a suffix to integral literal.

According to IT industry, we should use L bacause l looks like digit 1.

```
/* By default every integral literal is of type int only*/
public class Test4
{
public static void main(String[] args)
        {
            byte b = 128;
                System.out.println(b); //error

                short s = 32768;
                System.out.println(s); //error
    }
}
```

Here byte and short both the values are out of the range so compilation error.

```
//Assigning smaller data type value to bigger data type
public class Test5
{
public static void main(String[] args)
        {
            byte b = 125;
                short s = b;  //[Implicit OR Automatic OR Widening]
                System.out.println(s);
        }
}
```

```
//Converting bigger type to smaller type
public class Test6
{
public static void main(String[] args)
        {
                short s = 127;
                byte b = (byte)s;  //[Explicit OR Manual OR Widening]
                System.out.println(b);
        }
}
```

Note : User is responsible to provide Explicit type casting (In case user is trying to assign bigger to smaller type) but here there is chance of loss of data.

```java
public class Test7
{
public static void main(String[] args)
{
              byte x = (byte) 1L;
              System.out.println("x value  = "+x);

              long l = 29L;
              System.out.println("l value  = "+l);

      int y =  (int) 18L;
              System.out.println("y value  = "+y);


 }
}
```

Is java pure Object Oriented Language ?

No, Java is not a pure object oriented langauge because it is accepting primary data type,
Actually any language which accepts primary data type is not a pure object oriented language.

Only Objects are moving in the network but not the primary data type so java has introduced
Wrapper class concept to convert the primary data types into corresponding wrapper object.

Primary Data type		Wrapper Object
byte	:	Byte
short	:	Short
int	:	Integer
long	:	Long
float	:	Float
double	:	Double
char	:	Character
boolean	:	Boolean

Note : Apart from these 8 data types, Everything is an object in java so, if we remove all these 8 data types then java will become pure OOP language.

//Wrapper claases

```java
public class Test8
{
        public static void main(String[] args)
        {
                Integer x = 24;
                Integer y = 24;
                Integer z = x + y;
                System.out.println("The sum is :"+z);

                Boolean b = true;
                System.out.println(b);

                Double d = 90.90;
                System.out.println(d);

                Character c = 'A';
                System.out.println(c);
        }
}
```

Note : From JAVA 1.5 version we have two conecpt :

AutoBoxing : Converting Primitive to Wrapper Object (int to Integer)
UnBoxing : Converting Wrapper object back to primitive (Integer to int)

How to find out the minimum, maximum value as well as size of different data types :

The Warpper classes like Byte, Short, Integer and Long has provided predefined static and final variables to represent minimum value, maximum value as well as size of the respective data type.

Example :

If we want to get the minimum value, maximum value as well as size of byte data type then Byte class (Wrapper class) has provided the following final and static variables

Byte.MIN_VALUE   : -128

Byte.MAX_VALUE   : 127

Byte.SIZE   : 8 (bits format)

```java
 //Program to find out the range and size of Integeral Data type
public class Test9
{
        public static void main(String[] args)
        {
                System.out.println("\n Byte range:");
                System.out.println(" min: " + Byte.MIN_VALUE);
                System.out.println(" max: " + Byte.MAX_VALUE);
                System.out.println(" size :"+Byte.SIZE);

                System.out.println("\n Short range:");
                System.out.println(" min: " + Short.MIN_VALUE);
                System.out.println(" max: " + Short.MAX_VALUE);
                System.out.println(" size :"+Short.SIZE);

                System.out.println("\n Integer range:");
                System.out.println(" min: " + Integer.MIN_VALUE);
                System.out.println(" max: " + Integer.MAX_VALUE);
                System.out.println(" size :"+Integer.SIZE);

                System.out.println("\n Long range:");
                System.out.println(" min: " + Long.MIN_VALUE);
                System.out.println(" max: " + Long.MAX_VALUE);
                System.out.println(" size :"+Long.SIZE);

        }
}
```

What is the difference between System.out.print() and System.out.println()

When we use print() method with some parameter then after printing the data the cursor will be available in the same line
but on the other hand when we use println() then after printing the data the cursor will move to the next line.

```java
class Test
{
        public static void main(String[] args)
      {
       System.out.println("Hello");
          System.out.println("World");
         }
}
```

Output : Hello World in two different lines.

24-09-2024

Providing _ (underscore) in integeral Literal :

In Order to enhance the readability of large numeric literals, Java software people has provided _ from JDK 1.7V. While writing the big numbers to separate the numbers we can use _

We can't start or end an integral literal with _ we will get compilation error.

```
//We can provide _ in integral literal
public class Test10
{
        public static void main(String[] args)
        {
           long mobile = 98_1234_5678L;
                System.out.println("Mobile Number is :"+mobile);
        }
}
```

```
public class Test11
{
        public static void main(String[] args)
      {
        final int x = 127; //final we are using in the place
                         of const keyword
          byte b = x;
          System.out.println(b);
        }
}
```

Note : Here program will be executed without any error.

----

var keyword :

From java 10v, java software people has provided var keyword.

This var keyword can hold any kind of value but initialization is compulsory in the same line [compiler will come to know about the variable type based on value]

```
   var x = 12;   //x is int type
```

After initialization it will hold same kind of value till the end of the program.

```
        var a = 12;
            a = true; //Invalid
```

It can be used for only for local variable.

```
//var keyword [Introduced from java 10]
public class Test12
{
        public static void main(String[] args)
        {
                var x = 12;
                System.out.println(x);

                 x = 90;
                        System.out.println(x);

                        // x = "NIT";  //Invalid


        }
}
```

How to convert decimal number to Octal, Hexadecimal and Binary :

Integer class has provided the following static methods to convert decimal to octal, hexadecimal and binary.

1) public static String toBinaryString(int x) : Will convert the decimal into binary in String format.

2) public static String toOctalString(int x) : Will convert the decimal into octal in String format.

3) public static String toHexString(int x) : Will convert the decimal into hexadecimal in String format.

```
// Converting from decimal to another number system
public class Test13
{
    public static void main(String[] argv)
    {
                //decimal to Binary
        System.out.println(Integer.toBinaryString(5)); //111
```

```
        //decimal to Octal
System.out.println(Integer.toOctalString(15)); //17

        //decimal to Hexadecimal
System.out.println(Integer.toHexString(2781)); //add


    }
}
```
==
Floating Point Literal :

If a numeric literal contains decimal OR fraction then it is called Floating Point Literal.

Example : 12.90, 56.78, 1.0

In floating point literal we have 2 data types :
    a) float (32 bits)
    b) double (64 bits)

By default every floating point literal is of type double only so, the following statement will compilation error

```
        float f = 1.1;  //error
```

We can use the following 3 statements to convert double into float type.
```
        float f1 = (float) 1.0;
            float f2 = 5.6F;
            float f3 = 2.1f;
```


Even though every floating point literal is of type double only but to enhance the redability of the code compiler has provided two flavors to represent double value explicitly.

```
        double d1 = 1.2d;
            double d2 = 2.5D;
```

A floating point literal we can represent in exponent form.
```
        double d = 15e2; [15 X 10 to the power of 2]
```


* We can represent an integral literal in four different forms decimal, octal, hexadecimal and binary but in floating point literal we have only one form i.e decimal [Floating point literal we can't represent in octal, hexadecimal and binary]

* An integral literal i.e byte, short, int and long, we can assign to floating point literal(float and double) but a floating point literal we can't assign to integral literal.

Programs :

```java
public class Test
{
        public static void main(String[] args)
        {
                float f = 2.0; //error
                System.out.println(f);
        }
}
```

```java
public class Test1
{
        public static void main(String[] args)
        {
                float b = 15.29F;
                float c = 15.25f;
                float d = (float) 15.30;

                System.out.println(b + " : "+c+ " : "+d);

        }
}
```

```java
public class Test2
{
        public static void main(String[] args)
        {
                double d = 15.15;
                double e = 15d;
                double f = 15.15D;

                System.out.println(d+" , "+e+" , "+f);
        }
}
```

```java
public class Test3
{
```

```java
        public static void main(String[] args)
        {
                double x = 0129.89;

                double y = 0167;

                double z = 0178; //error

                System.out.println(x+","+y+","+z);
        }
}
------------------------------------------------------------
class Test4
{
        public static void main(String[] args)
        {
                double x = 0X29;

                double y = 0X9.15; //error

                System.out.println(x+","+y);
        }
}
------------------------------------------------------------
public class Test5
{
        public static void main(String[] args)
        {
                double d1 = 15e-3;
                System.out.println("d1 value is :"+d1);

                double d2 = 15e3;
                System.out.println("d2 value is :"+d2);
        }
}
------------------------------------------------------------
public class Test6
{
        public static void main(String[] args)
        {
                double a = 0791; //error

                double b = 0791.0;
```

```
            double c = 0777;

            double d = 0Xdead;

            double e = 0Xdead.0; //error
        }
}
------------------------------------------------------------
public class Test7
{
        public static void main(String[] args)
        {
          double a = 1.5e3;
          float b = 1.5e3;
          float c = 1.5e3F;
          double d = 10;
          int e = 10.0;
          long f = 10D;
          int g = 10F;
          long l = 12.78F;
        }
}
------------------------------------------------------------
//Range and size of floating point literal
public class Test8
{
        public static void main(String[] args)
        {
                System.out.println("\n Float range:");
                System.out.println(" min: " + Float.MIN_VALUE);
                System.out.println(" max: " + Float.MAX_VALUE);
                System.out.println(" size :"+Float.SIZE);

                System.out.println("\n Double range:");
                System.out.println(" min: " + Double.MIN_VALUE);
                System.out.println(" max: " + Double.MAX_VALUE);
                System.out.println(" size :"+Double.SIZE);
        }
}
------------------------------------------------------------
25-09-2024
------------
```

Boolean literal :

It is used to represent two states i.e true or false.

 Example : boolean isValid = true;
          boolean isEmpty = false;

In boolean literal we have only one data type i.e boolean data type which accepts 1 bit of memory as well as it depends upon JVM implementation.

Unlike C and C++, we can't assign integral literal to boolean data type.

   boolean b = 0; [Valid in C language but Invalid in java]
   boolean c = 1; [Valid in C language but Invalid in java]

We can't assign String literal to boolean data type.

    boolean b = "true"; //Invalid

== public class Test1
{
    public static void main(String[] args)
    {
       boolean isValid = true;
       boolean isEmpty = false;

       System.out.println(isValid);
       System.out.println(isEmpty);
    }
}
----
public class Test2
{
    public static void main(String[] args)
    {
       boolean c = 0; //error
       boolean d = 1; //error
       System.out.println(c);
       System.out.println(d);

```
        }
}
-------------------------------------------------------------
public class Test3
{
        public static void main(String[] args)
        {
                boolean x = "true"; //error
                boolean y = "false"; //error
                System.out.println(x);
          System.out.println(y);
        }
}
```
===

Char Literal :

It is also known as Character Literal.

In character Literal we have only one data type i.e char data type which accepts 16 bits of memory.

We can represent character literal by using the following ways :

  a) Single Character enclosed with single quotes.

     Example : char ch = 'A';

  b) In older languages like C and C++, which supports ASCII format and the range is 0 - 255, On the other hand java supports UNICODE format where the range is 0 - 65535. [0 is the minimum range and 65535 is the maximum range]

```
        char ch1 = 65535;  //Valid
          char ch2 = 65536;  //Invalid
```

  c) We can assign character literal to integral literal to know the UNICODE numeric value of that particular character.

```
            int x = 'A';   [UNICODE = ASCII + NON ASCII]
```

  d) We can also represent a char literal in 4 digit hexadecimal number where the format is

```
            '\udddd'   [Hexadecimal format]
```

Here u menas Unicode and d represents digits.

In this hexadecimal format the range is given below :

'\u0000'   [Minimum Range]

'\uffff'   [Maximum Range]

e) All the escape sequences are also represented as a char literal

char c = '\n';

```
public class Demo
{
        public static void main(String[] args)
        {
    int ch1 = '\u0000';
                System.out.println(ch1);  //0

                int ch2 = '\uffff';
                System.out.println(ch2); //65535

                //charcter 'a' 4 digit hexadecimal  representation
                char ch3 = '\u0061';
                System.out.println(ch3);

                //charcter 'A' 4 digit hexadecimal  representation
                char ch4 = '\u0041';
                System.out.println(ch4);

    }
}
```

```
public class Test1
{
        public static void main(String[] args)
        {
```

```java
            char ch1 = 'a';
            System.out.println("ch1 value is :"+ch1);

            char ch2 = 97;
            System.out.println("ch2 value is :"+ch2);


        }
}
------------------------------------------------------------
public class Test2
{
        public static void main(String[] args)
        {
                int ch = 'A';
                System.out.println("ch value is :"+ch);
        }
}
------------------------------------------------------------
//The UNICODE value for ? character is 63
public class Test3
{
        public static void main(String[] args)
        {
                char ch1 = 63;
                System.out.println("ch1 value is :"+ch1);

                char ch2 = 64;
                System.out.println("ch2 value is :"+ch2);

                char ch3 = 65;
                System.out.println("ch3 value is :"+ch3);
        }
}
------------------------------------------------------------
public class Test4
{
        public static void main(String[] args)
        {
                char ch1 = 47000;
        System.out.println("ch1 value is :"+ch1);

                char ch2 = 0Xadd;
```

```
                System.out.println("ch2 value is :"+ch2);



        }
}
```

Note : We will get the output as ? because the equivalant language translator is not available in the System.

//Addition of two character in the form of Integer
public class Test5
{
public static void main(String txt[ ])
 {
        int x = 'A';
      int y = 'B';

        System.out.println(x + y); //131
        System.out.println('A'+'A'); //130
  }
}

//Range of UNICODE Value (65535) OR '\uffff'
class Test6
{
        public static void main(String[] args)
        {
                char ch1 = 65535;
                System.out.println("ch value is :"+ch1);

                char ch2 = 65536; //error [out of range]
                System.out.println("ch value is :"+ch2);
        }
}

//WAP in java to describe unicode representation of char in hexadecimal format
public class Test7
{
        public static void main(String[] args)
        {
                int ch1 = '\u0000';
                System.out.println(ch1);
```

```java
                int ch2 = '\uffff';
                System.out.println(ch2);

                char ch3 = '\u0041';
        System.out.println(ch3); //A

                char ch4 = '\u0061';
                System.out.println(ch4); //a
        }
}
```

```java
class Test8
{
        public static void main(String[] args)
        {
                char c1 = 'A';
                char c2 = 65;
                char c3 = '\u0041';

                System.out.println("c1 = "+c1+", c2 ="+c2+", c3 ="+c3);
        }
}
```

```java
class Test9
{
        public static void main(String[] args)
        {
                int x = 'A';
                int y = '\u0041';
                System.out.println("x = "+x+" y ="+y);
        }
}
```

```java
//Every escape sequence is char literal
class Test10
{
        public static void main(String [] args)
        {
                char ch ='\n';
                System.out.println("Hello");
                System.out.println(ch);

        }
```

```
}
---------------------------------------------------------------
public class Test11
{
        public static void main(String[] args)
        {
                System.out.println(Character.MIN_VALUE); //white space
                System.out.println(Character.MAX_VALUE); //?
                System.out.println(Character.SIZE); //16 bits


        }
}
---------------------------------------------------------------
```

26-09-2024

String Literal :

String is a predefined class available in java.lang Package.

String is a collection of alpha-nemeric character which is enclosed by double quotes. These characters can be alphabets, numbers, symbol or any special character.

In java we can create String object by using following 3 ways :

 1) By using String Literal
    String str1 = "india";

2) By using new keyword
   String str2 = new String("Hyderabad");

3) By using Character array [Old Technique]
   char ch[] = {'R', 'A', 'J'};

```
//Three Ways to create the String Object
public class StringTest1
{
        public static void main(String[] args)
        {
                String s1 = "Hello World";      //Literal
                System.out.println(s1);

                String s2 = new String("Ravi"); //Using new Keyword
                System.out.println(s2);
```

```java
            char s3[] = {'H','E','L','L','O'}; //Character Array
            System.out.println(s3);


    }
}
```

```java
//String is collection of alpha-numeric character
public class StringTest2
{
        public static void main(String[] args)
        {
                String x="B-61 Hyderabad";
                System.out.println(x);

                String y = "123";
                System.out.println(y);

                String z = "67.90";
                System.out.println(z);

                String p = "A";
                System.out.println(p);
        }
}
```

```java
//IQ
public class StringTest3
{
        public static void main(String []args)
        {
                String s = 15+29+"Ravi"+40+40; //44Ravi4040
                System.out.println(s);


        }
}
```
----
4) Punctuators :

It is also called separators.

It is used to inform the compiler how things are grouped in the code.

() {} [] ; , . @ ☐ (var args)

----

5) Operators

It is a symbol which describes that how a calculation will be performed on operands.

Types Of Operators :

----

1) Arithmetic Operator (Binary Operator)

2) Unary Operators

3) Assignment Operator

4) Relational Operator

5) Logical Operators (&& || !)

6) Boolean Operators (& |)

7) Bitwise Operators (^ ~)

8) Ternary Operator

*9) Member Operator(Dot . Operator)

*10) new Operator

*11) instanceof Operator [It is also relational operator]

Basic Concepts of Operators :

```
class Test
{
        public static void main(String[] args)
        {
    int x = 15;
                int y = x++;
                System.out.println(x + " : "+ y);
        }
}
```

----

class Test

```java
{
        public static void main(String[] args)
        {
    int x = 15;
                int y = --x;
                System.out.println(x + " : "+ y);
        }
}
```

```java
class Test
{
        public static void main(String[] args)
        {
    int x = 15;
                int y = ++5; //error
                System.out.println(x + " : "+ y);
        }
}
```

```java
class Test
{
        public static void main(String[] args)
        {
        int x = 5;
          int y = ++(++x); //error
          System.out.println(x +": "+y);
        }
}
```

```java
class Test
{
        public static void main(String[] args)
        {
    char ch = 'A';
          ch++;
          System.out.println(ch);
        }
}
```

```java
class Test
{
        public static void main(String[] args)
        {
```

```
        double d1 = 12.12;
            d1++;
            System.out.println(d1);
        }
}
```

Note : Increment and decrement operator we can apply on any primitive data type except boolean.

What is a local variable :

If we declare a variable inside a method OR block OR Constructor then it is called local/Automatic/Temporary/Stack variable.

Example :

```
public void m1()
{
   int x = 100;  //Local Variable
}
```

A local variable must be initialized by the developer before use.

We can't apply any kind of access modifier on local veriable
except final.

As far as it's scope is concerned, It is accessible within the same method OR block OR constructor.

A local variable we can't use without pre declaration.

```
 public static void main(String[] args)
 {
        System.out.println(x);
        int x = 100;
 }
```

Program :

```
class Test
{
        public static void main(String[] args)
        {
```

```
            final int x = 100;
            System.out.println(x);


        }
}
```
==
Why we can't use a local variable outside of the method OR block OR Constructor ?

In java, Methods are executed in a special memory area called Stack area.

Stack is data structure which works on the basis of LIFO.

In java whenever we call a method then one stack frame is cretaed internally.

This Stack frame will automatically deleted, once the execution of the method is completed so with that stack frame all the local variables and parameter variables are also deleted from the memory as shown in the Program below.[Diagram 26-SEP-24]

MethodExecution.java

```java
public class MethodExecution
{
        public static void main(String[] args)
        {
          System.out.println("Main Method Started");
           m1();
           System.out.println("Main Method Ended");
        }

        public static void m1()
        {
          System.out.println("M1 Method Started");
           m2();
           System.out.println("M1 Method Ended :");
        }

        public static void m2()
        {
           int x = 100;
            System.out.println("I am m2 method "+x);
        }
```

}

Note : The local variable x is declared inside the method m2 so once the m2 method execution is over x will be deleted from the Stack frame so we can't use x variable outside of the method i.e. outside of the Stack Frame.

Limitation of Command Line Argument :

As we know by using Command Line Argument, we can pass some value at runtime, These values are stroed in String array variable and then only the exceution of the program will be started.

In Command line Argumenet we can't ask to enter the value from our end user as shown in the Program.

```java
public class CommandLineArgumentLimitation
{
        public static void main(String[] args)
        {
                System.out.println("Enter Your Age :");
                int age = Integer.parseInt(args[0]);

            if(age>18)
                {
                        System.out.println("Eligible 4 voting");
                }
                else
                {
                        System.out.println("Not Eligible 4 voting");
                }

        }
}
```

Note : Here we can't asK to enter the age from our end user so it is not user-friendly.
--How to read the gender from Command Line Argument :

```java
package com.ravi.command;

public class ReadGender
{
        public static void main(String[] args)
```

```
        {
                System.out.println("Please enter your Gender :");

                char ch = args[0].charAt(0);
                System.out.println("You gender is :"+ch);
        }

}
```

Note : charAt(int index) is a predefined method of String class
       which will retrieve the character based on the index
       position and the return type of this method is char.

       public char charAt(int index)

How to read the data from client OR End user ?

In order to read the data from the client we can use the following techniques :

 1) DataInputStream class (Java.io)
 2) BufferedReader (java.io)
 3) Console (Java.io)
 4) System.in.read(); (java.lang)
 5) Scanner (java.util)

Scanner class :

It is a predefined class available in java.util package though which we can read the data from
the client.

It is available from JDK 1.5v.

Static Variables of System class :

System class has provided 3 static and final variables :

1) System.out : It is used to provide normal message on the screen.

2) System.err : It is used to provide error message on the screen (red color)

3) System.in : It is used to take the input from the Source.

How to create an Object for Scanner class :

Scanner sc = new Scanner(System.in);

Methods of Scanner class :
----
1) public String next() : It will read a single word from the
                    user.

2) public String nextLine() : It will read multiple words or a
                    complete line from the user.

3) public byte nextByte() : It will read byte data.

4) public short nexShort() : It will read short data.

5) public int nextInt() : It will read int data.

6) public long nextLong() : It will read long data.

7) public float nextFloat() : It will read float data.

8) public double nextDouble() : It will read double data.

9) public boolean nextBoolean() : It will read boolean data.

10) public char next().charAt(0) : It will read first character
                        from the given word.

How to read your Name from the keyword :

```java
import java.util.*;
public class ReadName
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter your Name :");
                String name = sc.nextLine();
                System.out.println("Your Name is :"+name);
        }
}
```

How to read age from Sacnner class :

```
package com.ravi.scanner_demo;

import java.util.Scanner;

public class ReadAge
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter your Age :");
        int age = sc.nextInt();
        System.out.println("Your Age is :"+age);
        sc.close();
            }


}
```

How to read Employee Data from Scanner class :

```
package com.ravi.scanner_demo;

import java.util.Scanner;

public class EmployeeData {

        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);

                System.out.print("Enter your employee id :");
                int eid = sc.nextInt();

                System.out.print("Enter your employee Name :");
                String ename = sc.nextLine(); //\n
                ename = sc.nextLine(); //read client data

                System.out.println("Employee Id is :"+eid);
                System.out.println("Employee Name is :"+ename);

                sc.close();
```

```
        }

}
```

How to read gender from the Client using Scanner class :

```java
package com.ravi.scanner_demo;

import java.util.Scanner;

public class ReadGender {

        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter your Gender :");
                char gender = sc.next().charAt(0);
                System.out.println("Your Gender is :"+gender);
            sc.close();


        }

}
```

Expression Conversion :

Whenever we are working with Arithmetic Operator (+,-,*,/,%) or unary minus operator, after
expression exeution the result will be converted (Promoted) to int type, Actually to store the
result minimum 32 bits data format is required.

```java
public class Test
{
        public static void main(String [] args)
        {
                short s = 12;
                short t = 14;
                short u = s + t; //error
                System.out.println(u);

                byte b = 1;
                byte c = 2;
                int d = b + c;    //Valid
```

```
        }

}
```

After Arithmetic operator expression the result will be promoted to int type so, to hold the result minimum 32 bit data is required.

Unary minus operator :

```
public class Test
{
        public static void main(String [] args)
        {
                int x = 15;
                System.out.println(-x);  //-15

        }

}
```

```
public class Test
{
        public static void main(String [] args)
        {
                byte b = 2;
                byte c = -b; //error

                System.out.println(c);

        }

}
```

In Arithmetic operator OR Unary minus operator, the result will be promated to int type (32 bits) so to hold the result int data type is reqd.

```
public class Test
{
        public static void main(String [] args)
        {
                /*byte b = 1;
```

```
            b = b + 2;
            System.out.println(b); */

            byte b = 2;
                b += 2; //It is valid because shorthand operator
        System.out.println(b);


    }

}
```

In the above program we are using short hand operator so we will get the result in byte format also.

```
public class Test
{
        public static void main(String [] args)
        {
                int z = 5;
                if(++z > 5 || ++z > 6)   //Logical OR
                {
                        z++;
                }
                System.out.println(z);  //7

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

                z = 5;
                if(++z > 5 | ++z > 6)   //Boolean OR
                {
                        z++;
                }
                System.out.println(z); //8

        }

}
```

```
public class Test
{
        public static void main(String [] args)
        {
```

```java
            int z = 5;
            if(++z > 6 & ++z> 6)
            {
                    System.out.println("Inside If");
                    z++;
            }
            System.out.println(z);

    }

}
```

28-09-2024

Working with Bitwise AND(&), Bitwise OR(|) and Bitwise X-OR (^) :

Program on Bitwise X-OR :

```java
public class Test
{
        public static void main(String [] args)
        {
                System.out.println(false ^ false); //false
        }

}
```

Here we will get the output based on the input i.e Same input output will be zero.

```java
public class Test
{
        public static void main(String [] args)
        {
                System.out.println(5 & 6); //4
                System.out.println(5 | 6); //7
                System.out.println(5 ^ 6); //3
        }

}
```

Bitwise Complement Operator (~)

It will not work with boolean.

```java
public class Test
{
        public static void main(String [] args)
        {
                System.out.println(~ true); //error

        }

}
```

```java
public class Test
{
        public static void main(String [] args)
        {
                System.out.println(~ 5); //-6
                System.out.println(~ -4); //3

        }

}
```

Member Operator (.) :

It is also known as Dot operator OR Period.
It is used to access the member of the class.
If we want to access the variables OR methods which are available in the class then we should
use member access Operator.

```java
class Welcome
{
        static int x = 100;

        public static void greet()
        {
                System.out.println("Hello Batch 38");
        }

}

public class Test
{
```

```java
        public static void main(String [] args)
        {
                System.out.println(Welcome.x);
                Welcome.greet();


        }

}
```

Note : Welcome class contains static variable and static method so, we can directly call static veraible and static method with the help of class name using dot operator.
----
new Operator :

It is also a keyword.
It is used to create the object as well as it is responsible to
initialize all the non static data member with default value.

```java
package com.ravi.new_keyword;

class Welcome
{
        int x = 100; //non static variable

        public void greet() //non static method
        {
                System.out.println("Hello Batch 38");
        }
}


public class NewKeywordDemo
{
        public static void main(String[] args)
        {
                Welcome w = new Welcome();
                System.out.println(w.x);
                w.greet();


        }

}
```

Here Welcome class contains, non static member so Object is reqd.

Limitation of if else :

What is drawback of if condition :-

The major drawback with if condition is, it checks the condition again and again so It increases the burdon over CPU so we introduced switch-case statement to reduce the overhead of the CPU.

Switch case statement in java :

It is a selective statement so, we can select one statement among the available statements.

break is optional but if we use break then the control will move from out of the switch body.

We can write default so if any statement is not matching then default will be executed.

In switch case we can't pass long, float and double abd boolean value.

[long we can pass in switch case from java 14v]

We can pass String from JDK 1.7v and we can also pass enum from JDK 1.5v.

```java
import java.util.*;
public class SwitchDemo
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
        System.out.print("Please Enter a Character :");

                char colour = sc.next().toLowerCase().charAt(0);

                switch(colour)
                {
                case 'r' : System.out.println("Red") ; break;
                case 'g' : System.out.println("Green");break;
                case 'b' : System.out.println("Blue"); break;
                case 'w' : System.out.println("White"); break;
                default : System.out.println("No colour");
                }
                System.out.println("Completed") ;
```

```
            }
}
-----------------------------------------------------------
import java.util.*;
public class SwitchDemo1
{
public static void main(String args[])
  {
                System.out.println("\t\t**Main Menu**\n");
                System.out.println("\t\t**100 Police**\n");
                System.out.println("\t\t**101 Fire**\n");
                System.out.println("\t\t**102 Ambulance**\n");
                System.out.println("\t\t**139 Railway**\n");
                System.out.println("\t\t**181 Women's Helpline**\n");

                System.out.print("Enter your choice :");
                Scanner sc = new Scanner(System.in);
                int choice = sc.nextInt();

                switch(choice)
                {
                case 100:
                System.out.println("Police Services");
                break;
                case 101:
                System.out.println("Fire Services");
                break;
                case 102:
                System.out.println("Ambulance Services");
                break;
                case 139:
                System.out.println("Railway Enquiry");
                break;
                case 181:
                System.out.println("Women's Helpline ");
                break;
                default:
                System.out.println("Your choice is wrong");
                }
    }
}
-----------------------------------------------------------
import java.util.*;
```

```java
public class SwitchDemo2
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the name of the season :");
                String season = sc.next().toLowerCase();

                switch(season)  //String allowed from 1.7
                {
                        case "summer" :
                                System.out.println("It is summer Season!!");
                        break;

                        case "rainy" :
                                System.out.println("It is Rainy Season!!");
                        break;
                }
        }
}
```

```java
public class Test2
{
        public static void main(String[] args)
        {
                double val = 1;
                switch(val)  //Error, can't pass long, float and double
                {
                        case 1:
                                System.out.println("Hello");
                        break;
                }
        }
}
```

```java
public class Test
{
        public static void main(String[] args)
        {
          float I = 12;

          switch(I)
```

```
        {
   case 12 :
                        System.out.println("It is case 12");
               break;
               }
        }

}
```
Note : We can't pass float and double value.

```
public class Test
{
        public static void main(String[] args)
        {
            int x = 12;
               int y = 12;

          switch(x)
               {
            case y :  //error
                        System.out.println("It is case 12");
               break;
               }
        }

}
```

Note : In the label of switch we should take constant value.

```
public class Test
{
        public static void main(String[] args)
        {
            int x = 12;
               final int y = 12;

          switch(x)
               {
      case y :
                        System.out.println("It is case 12");
               break;
               }
        }
```

```java
}
```

```java
public class Test
{
        public static void main(String[] args)
        {
            int x = 12;
                final int y = 12;

          switch(x)
                {
        case y :
                        System.out.println("It is case 12");
                    break;
                }
        }

}
```

```java
public class Test
{
        public static void main(String[] args)
        {
          byte b = 90;

          switch(b)
                {
        case 128 :  //error
                        System.out.println("It is case 127");
                    break;
                }
        }

}
```

Note : Value 128 is out of the range of byte and same applicable for short data type
----
Loops in java :

A loop is nothing but repeatation of statements based on the
specified condition.

In java we have 4 types of loops :

----

1) do-while loop
2) while loop
3) for loop
4) for each loop

//Program on do while loop :

```java
public class DoWhile
{
        public static void main(String[] args)
        {
                do
                {
                        int x = 1;  //Local Variable (block Level)
                        System.out.println("x value is :"+x);
                        x++;
                }
                while (x<=10); //error
        }
}
```

Note : x is a block level variable because It is declared inside do block so the scope of this x variable will be within the do block only.

```java
package com.ravi.loop;

public class DoWhile
{
        public static void main(String[] args)
        {
                int x = 1;  //Local Variable
                do
                {
                        System.out.println("x value is :"+x);
                        x++;
                }
                while (x<=10); //error
        }
}
```

----

package com.ravi.loop;

```java
public class WhileLoop
{
        public static void main(String[] args)
        {
                int x = 1;

                while(x>=-10)
                {
                        System.out.println(x);
                        x--;
                }
        }
}
```

```java
package com.ravi.loop;

public class ForLoop
{
        public static void main(String[] args)
        {
                for(int i=1; i<=10; i++)
                {
        System.out.println(i);
                }
        }
}
```
==
30-09-2024

For Each loop :

It is an enhanced for loop.
It is introduced from JDK 1.5v.
It is used to retrieve the values one by one from the Collection like array.

How to fech an array value using for each loop :

```java
package com.ravi.for_each_loop;

public class ForEachLoopDemo1
{
        public static void main(String[] args)
```

```
        {
                int []values = {89, 78, 56, 34, 12, 44, 34};

                for(int x : values)
                {
                  System.out.println(x);
                }

        }

}
```

//Retrieving the value through command line argument
package com.ravi.for_each_loop;

```
public class FoEachLoopDemo2
{
   public static void main(String[] args)
   {
        for(String x : args)
        {
                System.out.println(x);
        }
   }
}
```

How to sort Array data :

In java.util pacakge, there is a predefined class called Arrays which has various static methods
to sort the array in descending or alphabetical order.

Example :
        Arrays.sort(int []arr); //For sorting int array
          Arrays.sort(Object []arr) //For sorting String array

----
How to sort an integer array :
----
package com.ravi.for_each_loop;

import java.util.Arrays;

public class ForEachLoopDemo3
```

```
{
        public static void main(String[] args)
        {
                int arr[] = {40,50,30,90,10,20};

                Arrays.sort(arr);

                for(int x : arr)
                {
                        System.out.println(x);
                }

        }

}
```
---------------------------------------------------------------
How to sort String array :
---------------------------
package com.ravi.for_each_loop;

import java.util.Arrays;

public class ForEachLoopDemo4
{
        public static void main(String[] args)
        {
                String fruits[] = {"Orange", "Mango","Apple"};

                Arrays.sort(fruits);

                for(String fruit : fruits)
                {
                        System.out.println(fruit);
                }

        }

}
---------------------------------------------------------------
In java, Can we hold hetrogeneous types of data using array ?
---------------------------------------------------------------
Yes, by using Object array we can hold hetrogeneous type of data but we can't peform sorting
operation using Arrays.sort(), It will generate

java.lang.ClassCastException

```java
package com.ravi.for_each_loop;

import java.util.Arrays;

public class ForEachLoopDemo5 {

    public static void main(String[] args)
    {
        Object []arr = {12, "NIT", 34.78, 'A', false};

        // Arrays.sort(arr); [Invalid becoz data is hetro]

        for(Object x : arr)
        {
            System.out.println(x);
        }

    }

}
```

What is BLC and ELC class in java ?

BLC :

It stands for Business Logic class.
We should always write the logic in BLC class.
BLC class will never contain main method, It will only contain the logic.

class BLC
{
}

ELC :

It stands for Executable logic class.
We should never write the actual logic in the ELC class.
It is always contain main method from where the execution of the program will start.

class ELC

```
{
  public static void main(String [] args)
  {
  }
}
```
==
How to reuse a class in java ?

The slogan of java is "WORA" write once run anywhere.

A public class created in one package can be reuse from different packages also by using import statement.

In a single java file, we can declare only one public class and that class can be reusable to all the packages.

*In a single java file, we can write only one public class and multiple non-public classes but it is not a recommended approach because the non public class we can use within the same package only.

So the conclusion is, we should declare every java class in a separate file to enhance the reusability of the BLC classes.
[Note we have 10 classes -> 10 java files]

How many .class file will be created in the above approach :

For a public class in a single file, Only 1 .class file will be created.

For a public class in a single file which contains n number of non public classes then compiler will generate n (number of .java) number of .class file.

Example :

Test.java (BLC)
----
public class Test
{

}
class A
{
}
```

```
class B
{
}
```

Here file name must be Test.java but after compilation 3 .class files will be created i.e Test.class, A.class and B.class

----

Program that describes how to reuse Welcome.java(BLC) file into different packages :

Welcome.java(BLC) [It is available in com.nit package]

```
package com.nit;

//BLC
public class Welcome  //10 java classes    1 java files
{
        public static void greet()
        {
                System.out.println("Welcome Batch 38");
        }
}
```

Main.java(ELC) [It is available in com.nit package]

```
package com.nit;

//ELC
public class Main
{
        public static void main(String[] args)
        {
    Welcome.greet();

        }
}
```

ELC.java(ELC) [It is available in another pacakge i.e com.ravi package so import keyword is required]

```
package com.ravi;
```

```java
import com.nit.Welcome; //Importing the Welcome class
                //from com.nit pacakge
//ELC
public class ELC
{
        public static void main(String[] args)
        {
                Welcome.greet();

        }

}
```

Working with static method and method return type :

If a static method is declared in the ELC class then we can directly call the static method from the main method as shown in the program.

```java
//A static method can be directly call within the same class
package com.ravi.pack1;
```

Test1.java

```java
public class Test1
{
        public static void main (String[] args)
        {
                square(5);
        }

        public static void square(int x)
        {
          System.out.println("Square is :"+(x*x));
        }
}
```
----

2 files :

GetSquare.java

```java
package com.ravi.pack2;

//BLC
```

```java
public class GetSquare
{
   public static void getSquareOfNumber(int num)
   {
           System.out.println("Square of "+num+" is :"+(num*num));
   }
}
```

Test2.java

```java
package com.ravi.pack2;

import java.util.Scanner;

//ELC
public class Test2
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the side :");
                int side = sc.nextInt();
                GetSquare.getSquareOfNumber(side);
                sc.close();

        }
}
```

Note : In the above program, there is no communication between
     BLC and ELC class, ELC class is sending the value of side
     of the square, BLC class is accepting the value, perforing
     some operation but not returing the value.
----
2 files :

FindSquare.java

```java
//A static method returning integer value
package com.ravi.pack3;

//BLC
public class FindSquare
```

```java
{
        public static int getSquare(int x)
        {
                return (x*x);
        }
}
```

Test3.java

```java
package com.ravi.pack3;

import java.util.Scanner;

//ELC
public class Test3
{
        public static void main (String[] arg)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the value of side :");
                int side = sc.nextInt();

            int square = FindSquare.getSquare(side);
            System.out.println("Square of "+side+" is :"+square);
            sc.close();
        }
}
```

Note : Here there is a communication between BLC and ELC class, ELC class is sending the
value, BLC class is receiving, after
processing, It is sending the result back to ELC class.
----
2 files :

Calculate.java

```
/*Program to find out the square and cube of
  the number by following criteria
 *
a) If number is 0 or Negative it should return -1
b) If number is even It should return square of the number
c) If number is odd It should return cube of the number
*/
```

```java
package com.ravi.pack4;

//BLC
public class Calculate
{
  public static int getSquareAndCube(int num)
  {
        if(num <=0)
        {
                return -1;
        }
        else if(num % 2==0)
        {
                return (num*num);
        }
        else
        {
                return (num*num*num);
        }
  }

}

Test4.java
-----------
package com.ravi.pack4;

import java.util.Scanner;

public class Test4
{
  public static void main(String[] args)
  {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number :");
        int num = sc.nextInt();

        int result = Calculate.getSquareAndCube(num);
        System.out.println("Result is :"+result);

        sc.close();
```

```
  }

}
```

Note : Whenever we receive the outer world data from our end client, we should always validate the data with different test cases.

----

2 files :

Rectangle.java

----

package com.ravi.pack5;

//BLC
public class Rectangle
{
  public static double getAreaOfRectangle(double length, double breadth)
  {
        return (length * breadth);
  }

}

Test5.java

----

package com.ravi.pack5;

import java.util.Scanner;

public class Test5
{
        public static void main(String[] args)
        {
          Scanner sc = new Scanner(System.in);
          System.out.print("Enter the length of the Rect :");
          double length = sc.nextDouble();

          System.out.print("Enter the breadth of the Rect :");
          double breadth = sc.nextDouble();

          double areaOfRectangle = Rectangle.getAreaOfRectangle(length, breadth);
```

```
        System.out.println("Area of Rectangle is :"+areaOfRectangle);
        sc.close();


    }
}
-------------------------------------------------------------
2 files :
---------
EvenOrOdd.java
---------------
package com.ravi.pack6;

//BLC
public class EvenOrOdd
{
        public static boolean isEven(int num)
    {
      return (num % 2 == 0);
    }
}

Test.java
-----------
package com.ravi.pack6;

import java.util.Scanner;

//ELC
public class Test6
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter a Number :");
                int num = sc.nextInt();

            boolean isEven = EvenOrOdd.isEven(num);
            System.out.println(num+" is Even ?:"+isEven);

            System.out.print("Enter another Number :");
                num = sc.nextInt();
```

```java
        isEven = EvenOrOdd.isEven(num);
        System.out.println(num+" is Even ?:"+isEven);
        sc.close();
    }

}
```
----
2 files :

```java
//Area of Circle
//If the radius is 0 or Negative then return -1.

package com.ravi.pack7;
public class Circle
{
  public static String getAreaOfCircle(double radius)
  {
        if(radius <=0)
        {
                return ""+(-1);
        }
        else
        {
                final double PI = 3.14;
                double areaOfCircle = PI * radius * radius;
                return ""+areaOfCircle;
        }

  }
}
```

```java
package com.ravi.pack7;

import java.util.Scanner;

public class Test7
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the radius :");
                double radius = sc.nextDouble();
```

```java
                String areaOfCircle = Circle.getAreaOfCircle(radius);

                float area = Float.parseFloat(areaOfCircle);

                System.out.printf("Area of circle is  : %.2f ",area);
                sc.close();

        }
}
```

Note : Instead of using System.out.printf() we can also use DecimalFormat class for decimal number formatting as shown in the below program.

How to provide formatting for Decimal number :

In java.text package, there is a predefined class called DecimalFormat through which we can provide formatting for Decimal number.

DecimalFormat df = new DecimalFormat("00.00"); //format [String pattern]
System.out.println(df.format(double d));

Note :- format is non static method of DecimalFormat class which accpts double as a parameter, and return type of this method is
String.

public String format(double number)

```java
package com.ravi.pack7;
public class Circle
{
  public static String getAreaOfCircle(double radius)
  {
        if(radius <=0)
        {
                return ""+(-1);
        }
        else
        {
                final double PI = 3.14;
                double areaOfCircle = PI * radius * radius;
                return ""+areaOfCircle;
        }
```

```java
    }
}

package com.ravi.pack7;

import java.text.DecimalFormat;
import java.util.Scanner;

public class Test7
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the radius :");
                double radius = sc.nextDouble();

                String areaOfCircle = Circle.getAreaOfCircle(radius);

                double area = Double.parseDouble(areaOfCircle);

                DecimalFormat df = new DecimalFormat("000.000");
                System.out.println("Area of Circle is :"+df.format(area));

                sc.close();

        }
}
```

----
2 files :

Student.java

```java
package com.ravi.pack8;

//BLC
public class Student
{
 public static String getStudentDetails(int roll, String name, double fees)
  {
        //[Student name is : Ravi, roll is : 101, fees is :1200.90]

         return "[Student name is :"+name+", roll is :"+roll+", fees is :"+fees+"]";
```

```
    }
}

Test8.java
-----------
package com.ravi.pack8;

public class Test8
{
        public static void main(String[] args)
        {
          String studentDetails = Student.getStudentDetails(101, "Scott", 12000.90);
          System.out.println(studentDetails);
        }

}
-------------------------------------------------------------
2 files :
---------
Table.java
-----------
package com.ravi.pack9;

//BLC
public class Table
{
  public static void printTable(int num) //[5 X 1 = 5]
  {
        for(int i =1; i<=10; i++)
        {
                System.out.println(num+" X "+i+" = "+(num*i));
        }
        System.out.println(".......................");

  }
}

Test9.java
-----------
package com.ravi.pack9;

//ELC
```

```java
public class Test9
{
        public static void main(String[] args)
        {
                for(int i=1; i<=20; i++)
                {
                        Table.printTable(i);
                }
        }

}
```

02-10-2024

Types of Variable in java :

Based on the data type we have only 2 types of variable in java :

  1) Primitive Variables
  2) Reference Variables

1) Primitive Variables :

If a variable is declared with primitive data type like byt, short, int, long and so on then it is called
Primitive Variables.

Example :
        int x = 100;
          boolean y = true;

2) Reference Variable :

If a variable is declared with class name, interface name, enum , record and so on then it is
called reference variable.

Example :
        Student s; //s is reference variable
           Scanner sc = new Scanner(System.in); //sc is reference variable
        Integer y = null; //y is reference variable

Note : With primitive variable we can't call a method as well as we can't assign null literal.

      int x = null; //Invalid

```
        int y = 45;
        y.m1(); //Invalid
```

Based on declaration position Variables are divided into 4 types :

  a) Class Variables OR Static Field
  b) Instance Variables OR Non Static Field
  c) Local /Stack/temporary/Automatic Variable
  d) Parameter Variables

Program on Primitive Variables :

PrimitiveVariables.java

```java
package com.ravi.variables;

class Test
{
        static int a = 100;  //Static Field
        int b = 200; //Non Static Field

        public void acceptData(int c) //Parameter Variable
        {
                int d = 400; //Local Variable

                System.out.println("Static Field :"+a);
                System.out.println("Non static Field :"+b);
                System.out.println("Parameter Variable :"+c);
                System.out.println("Local Variable : "+d);
        }
}

public class PrimitiveVariables
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                t1.acceptData(300);

        }
```

}

//Program on Reference Variable :
----
package com.ravi.variables;

class Student
{
        Student st1 = new Student(); //st1 is a non static field
        static Student st2 = new Student(); //st2 is a static field

        public void acceptData(Student st3) //st3 is parameter variable
        {
                Student st4 = new Student(); //st4 is local variable
        }
}

public class ReferenceVariables {

        public static void main(String[] args)
        {
                System.out.println("Reference Variable");

        }

}

Object Oriented Programming (OOPs)

What is an Object?

An object is a physical entity which exist in the real world.
Example :- Pen, Car, Laptop, Mouse, Fan and so on

An Object is having 3 characteristics :

   a) Identification of the Object (Name of the Object)
   b) State of the Object (Data OR Properties OR Variable of Object)
   c) Behavior of the Object (Functionality of the Object)

OOP is a technique through which we can design or develop the programs using class and
object.

Writing programs on real life objects is known as Object Oriented Programming.

Here in OOP we concentrate on objects rather than function/method.

Advantages of OOP :
----
1) Modularity (Dividing the bigger task into smaller task)
2) Reusability (We can reuse the component so many times)
3) Flexibility (Easy to maintain [By using interface])

Features of OOP :
----
1) Class
2) Object
3) Abstraction
4) Encapsulation
5) Inheritance
6) Polymorphism

03-10-2024

What is a class?

A class is model/blueprint/template/prototype for creating the object.

A class is a logical entity which does not take any memory.

A class is a user-defined data type which contains data member and member function.

public class Employee
{
    Employee Data (Properties)
        +
    Employee behavior (Function/Method)
}

A CLASS IS A COMPONENT WHICH IS USED TO DEFINE OBJECT PROPERTIES AND OBJECT BEHAVIOR.

WAP to initialize the Object properties using Object reference ?
----
2 files : [Diagram]

Student.java
----
package com.ravi.oop;

//BLC
public class Student
{
   //Object Properties (non static variable)
   String name;
   int age;
   int rollNumber;

   //Object Behavior
   public void talk()
   {
           System.out.println("My name is :"+name);
           System.out.println("My Age is :"+age);
           System.out.println("And My rollNumber is :"+rollNumber);
   }

   public void writeExam()
   {
           System.out.println(name+" is writing exam");
   }

}

StudentDemo.java
----
package com.ravi.oop;

//ELC
public class StudentDemo
{
        public static void main(String[] args)
        {
                Student raj = new Student();

                //Initialize the Object properties through Object reference
                raj.name = "Raj Gourav";
                raj.age = 19;
                raj.rollNumber = 111;

```
                //Calling the behavior through Object reference
                raj.talk();
                raj.writeExam();

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

                Student priya = new Student();
                //Initialize the Object properties through Object reference
                priya.rollNumber = 222;
                priya.name = "Priya";
                priya.age = 19;

                //Calling the behavior through Object reference
                priya.talk();
                priya.writeExam();

        }

}
```

Steps for creating Object Orineted Programming

Step 1 :- Create the Object based on the BLC class inside ELC
        class

Step 2 :- Define all the object properties and behavior inside
        the BLC class based on your imagination/thinking.

Step 3 :- Initialize all the object properties with user friendly value by using reference variable.

step 4 :- call the behavior (calling the methods)

Another Program on OOP to initialize the Object properties through Object Reference.

2 files :

Lion.java

```java
package com.ravi.oop;

public class Lion
{
   String name;
```

```java
    double height;
    String color;

    public void roar()
    {
            System.out.println("I can roar so don't come near to me");
    }

    public String getLionInformation()
    {
            return "[Lion Name is :"+name+", height is :"+height+", color is :"+color+"]";
    }
}
```

LionDemo.java

```java
package com.ravi.oop;

public class LionDemo {

        public static void main(String[] args)
        {
                Lion lion = new Lion();
                lion.name = "Tiger";
                lion.height = 4.4;
                lion.color = "Brown";

                lion.roar();
                System.out.println(lion.getLionInformation());
        }

}
```

Initialize the Object properties through Methods :
----
2 files :

Customer.java

```java
package com.ravi.oop;

public class Customer
{
```

```java
    int customerId;
    String customerName;
    double customerBill;

    //Initializing the object properties through Method
    public void setCustomerData()
    {
            customerId = 999;
            customerName = "Mr. Smith";
            customerBill = 24897.89;
    }

    public void getCustomerData()
    {
            System.out.println("Customer Id is :"+customerId);
            System.out.println("Customer Name is :"+customerName);
            System.out.println("Customer Bill is :"+customerBill);
    }

}
```

CustomerDemo.java

```java
package com.ravi.oop;

public class CustomerDemo
{
        public static void main(String[] args)
        {
                Customer smith = new Customer();
                smith.setCustomerData();
                smith.getCustomerData();
        }
}
```

Note : Upto here, we have learned the following two ways to initialize the Object properties
  a) By using Object Reference (raj.name = "Raj Gourav")
  b) By using Methods (smith.setCustomerData())
==
04-10-2024

What is an instance variable OR Non static variable ?

It is a class level variable.

If a non static variable defined inside a class but outside of a method then it is called Instance Variable.

Example :

```java
public class Customer
{
   int cutsomerId; //Instance variable OR non static field

   public void setCustomerData()
   {
   }
}
```

An instance variable life will autonmtically start during object creation.[We can't think about instance variable without object creation]

Every instance variable must be some defualt value.

Instance variable scope is within the same class as well as the depends upon the access modifier we have applied on instance variable.

Every time we create an object in java, a separate copy of all the instance variables will be created.

How to initialize the Object properties through Method Parameter :

We can also initialize the Object properties using Method parameter as shown below :

2 files :

Employee.java

```java
package com.ravi.oop;
//BLC
public class Employee
{
   int employeeId;  //instance variable and parameter variable
   String employeeName;

   public void setEmployeeData(int eid, String ename)
```

```java
    {
            employeeId = eid;
            employeeName = ename;
    }

    public void getEmployeeData()
    {
            System.out.println("Employee Id is :"+employeeId);
            System.out.println("Employee Name is :"+employeeName);
    }
}
```

EmployeeDemo.java

```java
package com.ravi.oop;

public class EmployeeDemo
{
        public static void main(String[] args)
        {
                Employee raj = new Employee();
                raj.setEmployeeData(111, "Raj");
                raj.getEmployeeData();

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

                Employee sneha = new Employee();
                sneha.setEmployeeData(222, "Sneha");
                sneha.getEmployeeData();

        }

}
```

Constructor [Introduction Only]

If the name of the class and name of the method both are exactly same and it does not contain
any return type then it is called
Constructor.

Example :

```
public class Sample
{
    public Sample()  //Constructor
    {

    }
}
```

In java, Whenever we write a class and if we don't write any type of constructor then by default compiler will add one constructor in the class known as Default constructor.

Every java class must contain at-least one constructor either explicitly written by user or implicitly added by compiler (default constructor).

Test.java

```
public class Test
{

}
```

javac Test.java (Compilation)

Test.class

```
public class Test
{
    public Test() //Default constructor added by compiler
    {
    }
}
```

The default constructor (Added by compiler) access modifier depends upon class access modifier that means if class is public
then the default constructor added by compiler will also be public.

Why compiler adds default constructor ?

Compiler adds default constructor in the class due to following two reasons

1) Without default constructor Object creation is not possible in Java.

2) With the help of new keyword, constructor is responsible to provided default values for the

non static variables(using new Keyword) and by using user-defined constructor we can re-initialize our non static variables with user defined values.

Data type - Default value
  byte - 0
  short - 0
  int   - 0
  long - 0
  float - 0.0
  double - 0.0
  char - (space) '\u0000'
  boolean - false
  String - null
  Object - null (For any class i.e reference variable the default value is null)

Program that describes default values are provided by new keyword with the help of constructor:

```
package com.ravi.oop;

public class Player
{
    int playerId;
    String playerName;

    public void show()
    {
        System.out.println(playerId);
        System.out.println(playerName);
    }

        public static void main(String[] args)
        {
                Player p1 = new Player();
                p1.show();
        }

}
```

Output : 0 and null.

What is Variable Shadow in java ?

Variable shadowing in Java occurs when a variable declared within a certain scope (like a

method or a block or Constructor) has the same name as a variable declared in an outer scope (class Level).

In variable Shadow, the variable in the inner scope hides the variables in Outer scope so known as variable shadowing.

This means that within the inner scope, when we refer to the variable by name, We are actually referring to the inner variable, not the outer one.

Student.java

```java
package com.ravi.variable_shadow;

public class Student
{
    int rollNumber = 111;
    String studentName = "Scott";

    public void printStudentData()
    {
            int rollNumber = 222;
            String studentName = "Smith";

            System.out.println("Roll Number is :"+rollNumber);
            System.out.println("Name  is :"+studentName);

    }


}
```

VariableShadow.java

```java
package com.ravi.variable_shadow;

public class VariableShadow
{
        public static void main(String[] args)
        {
                Student s1 = new Student();
                s1.printStudentData();
```

```
        }

}
```

Output is : 222 Smith

Note : From the above program it is clear that Method level variables are having more priority than class level variable inside the method, block Or constructor.

If we want to access non static variable of the class (class level variable) then we should use this keyword.

```java
package com.ravi.this_keyword;

class Employee
{
        int employeeId = 111;
        String employeeName = "Raj";

        public void getData()
        {
                int employeeId = 222;
                String employeeName = "Smith";

                System.out.println("Employee Id is :"+this.employeeId);
                System.out.println("Employee Name is :"+this.employeeName);
        }
}
public class ThisDemo
{
        public static void main(String[] args)
        {
                Employee e1 = new Employee();
                e1.getData();

        }

}
```

Method local search algorithm :

Whenever we use any variable inside the method body, block or constructor body then compiler will search that variable inside the method body first then in method parameter, if the variable

DECLARATION is not available in the method then it is not a method level variable, actually it may be class level variable so now the compiler will search the variable at class lavel.
[05-OCT-24]

```java
package com.ravi.this_keyword;

class Test
{
        static int a = 100;
        int b = 200;

        public void acceptData(int c)
        {
                int d = 400;
                System.out.println("Static Field :"+Test.a);
                System.out.println("Non Static Field :"+this.b);
                System.out.println("Parameter Variable :"+c);
                System.out.println("Local varaible :"+d);
        }
}

public class MethodSerachAlgorithm
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                t1.acceptData(300);

        }

}
```
----
this keyword in java :

Whenever instance variable name and parameter variable name both are same then at the time of instance variable initialization our runtime environment will provide more priority to parameter variable, parameter variables are hiding the instance variables (Due to variable shadow)

To avoid the above said problen, Java software people introduced "this" keyword.

this keyword always refers to the current object and instance variables are the part of the object so by using this keyword we can refer to instance variable.

We cannot use this (non static member) keyword from static area (Static context).
----
2 files :

Customer.java
----
package com.ravi.this_ex;

public class Customer
{
    int customerId;
    String customerName;

    public void setCustomerData(int customerId, String customerName)
    {
            this.customerId = customerId;
            this.customerName = customerName;

    }

    public void getCustomerData()
    {
            System.out.println("Customer Id is :"+this.customerId);
            System.out.println("Customer Name is :"+this.customerName);
    }

}

CustomerDemo.java
----
package com.ravi.this_ex;

public class CustomerDemo
{
        public static void main(String[] args)
    {
                Customer raj = new Customer();
                raj.setCustomerData(111, "Raj");
                raj.getCustomerData();

        }

}

07-10-2024

How to print object properties by using toString() method :

If we want to print our object properties (Instance Variable) then we should generate(override) toString() method in our class from Object class.

Now with the help of toString() method we need not to write any display kind of method to print the object properties i.e instance variable.

In order to generate the toString() method we need to follow the steps
Right click on the program -> source -> generate toString()

In order to call this toString() method, we need to print the corresponding object reference by using System.out.println() statement.

Manager m = new Manager();
System.out.println(m); //Calling toString() method of Manager class

Employee e = new Employee();
System.out.println(e); //Calling toString() method of Employee class.

2 files :

Manager.java

```java
package com.ravi.printing_objeect_properties;

public class Manager
{
        int managerId;
        String managerName;

        public void setManagerData(int managerId, String managerName)
        {
                this.managerId = managerId;
                this.managerName = managerName;
        }

        @Override
        public String toString() {
                return "Manager [managerId=" + managerId + ", managerName=" +
```

```
managerName + "]";
        }



}

ManagerDemo.java
-----------------
package com.ravi.printing_objeect_properties;

public class ManagerDemo
{
        public static void main(String[] args)
        {
                Manager m1 = new Manager();
                m1.setManagerData(111, "Scott");
                System.out.println(m1); //toSting() method of manager class
        }

}
------------------------------------------------------------
How to initialize the instance variable through parameter variable as per requirement.

2 files :
---------
Employee.java
--------------
package com.ravi.printing_objeect_properties;

public class Employee
{
        int employeeId;
        String employeeName;
        double employeeSalary;
        char employeeGrade;

        public void setEmployeeData(int employeeId, String employeeName, double
employeeSalary) {
                this.employeeId = employeeId;
                this.employeeName = employeeName;
                this.employeeSalary = employeeSalary;
        }
```

```java
        @Override
        public String toString()
        {
                return "Employee [employeeId=" + employeeId + ", employeeName=" +
employeeName + ", employeeSalary="
                                        + employeeSalary + ", employeeGrade=" + employeeGrade + "]";
        }

        public void calculateEmployeeGrade()
        {
                if(this.employeeSalary >= 90000)
                {
                        this.employeeGrade = 'A';
                }
                else if(this.employeeSalary >= 75000)
                {
                        this.employeeGrade = 'B';
                }
                else if(this.employeeSalary >= 50000)
                {
                        this.employeeGrade = 'C';
                }
                else
                {
                        this.employeeGrade = 'D';
                }
        }

}

EmployeeDemo.java
----------------
package com.ravi.printing_objeect_properties;

public class EmployeeDemo
{
        public static void main(String[] args)
        {
                Employee emp = new Employee();
                emp.setEmployeeData(111, "Scott", 99000);
                emp.calculateEmployeeGrade();
                System.out.println(emp);
        }
```

```
}
```

Role of instance variable while creating the Object :

Whenever we create an objet in java, a separate copy of all the instance variables will be created with each and every object, which is the part HEAP memory as shown in the Program.

Test.java

```java
package com.ravi.instance_variable_copy;

public class Test
{
        int x = 100;

        public static void main(String[] args)
        {
    Test t1 = new Test();
    Test t2 = new Test();

    ++t1.x;    --t2.x;

    System.out.println(t1.x); //101
    System.out.println(t2.x); //99
        }
}
```

What is a static field ?

It is a class level variable.

If a variable is declared with static modifier inside a class then it is called class variable OR static field.

A static field variable will be automatically initialized with default values and memory will be allocated (even the variable is final) AT THE TIME OF LOADING THE CLASS INTO JVM MEMORY.

In order to access the static member, we need not to create an object, here class name is required.

Role of static variable in Object creation :

Whenever we create an object in java then a single copy of static variables will be created in Method area OR class area and it is sharable by all the objects as shown in the program.

```java
package com.ravi.static_variable_demo;

public class Demo
{
    static int x = 100;

        public static void main(String[] args)
        {
    Demo d1 = new Demo();
    Demo d2 = new Demo();

    ++d1.x;  ++d2.x;

    System.out.println(d1.x); //102
    System.out.println(d2.x); //102

        }

}
```

So, The conclusion is :

   Instance Variable = Multiple Copies with each and every object
   Static Field = Single Copy is sharable by all the Objects.

When we should decalre a variable as static variable and when we should declare as variable as a non static variable ?

Instance Variable :

If the value of the variable is different with respect to objects then we should declare a variable as a instance variable.

Static Variable :

If the value of the variable is common to all the objects then we should declare a variable as a static variable.

2 files :

Student.java

```java
package com.ravi.static_variable_role;

public class Student {
        int studentId;
        String studentName;
        String studentAddress;
        static String collegeName = "NIT";
        static String courseName = "Java";

        public void setStudentData(int studentId, String studentName, String studentAddress) {
                this.studentId = studentId;
                this.studentName = studentName;
                this.studentAddress = studentAddress;
        }

        @Override
        public String toString()
        {
                return "Student [studentId=" + studentId + ", studentName=" + studentName + ",
studentAddress=" + studentAddress+", College Name is :"+Student.collegeName+", Course
Name is "+Student.courseName
                                        + "]";
        }

}
```

StudentDemo.java

```java
package com.ravi.static_variable_role;

public class StudentDemo {

        public static void main(String[] args)
        {
                Student raj = new Student();
                raj.setStudentData(1, "Raj", "Ameerpet");
                System.out.println(raj);

                Student smith = new Student();
                smith.setStudentData(2, "Smith", "S.R Nagar");
                System.out.println(smith);
```

```
        }

}
```

What is Data Hiding ?

Data hiding is nothing but declaring our data members with private access modifier so our data will not be accessible from outer world that means no one can access our data directly from outside of the class.

*We should provide the accessibility of our data through methods so we can perform VALIDATION ON DATA which are coming from outer world.

2 files :

Customer.java

```java
package com.ravi.data_hiding;

public class Customer
{
    private double balance = 10000;  //Data Hiding

    public void deposit(double amount)
    {
            if(amount <=0)
            {
                    System.err.println("Amount can't be deposited");
            }
            else
            {
            this.balance = this.balance + amount;
            System.out.println("Balance After deposit :"+this.balance);
            }
    }

    public void withdraw(double amount)
    {
            this.balance = this.balance - amount;
            System.out.println("Balance after withdraw is :"+this.balance);
    }
```

```
    }

BankingApplication.java
-----------------------
package com.ravi.data_hiding;

public class BankingApplication
{
        public static void main(String[] args)
        {
                Customer scott = new Customer();
                scott.deposit(10000);
                scott.withdraw(5000);


        }
}
```

Note : Our balance data is private (Data hiding) so it is not accessible outside of the class, we can accesss via method method with proper data (Data Validation)

What is Constructor ?

What is the advantage of writing constructor in our class ?

If we don't write a constructor in our program then variable initialization and variable re-initialization both are done in two different lines.

If we write constructor in our program then variable initialization and variable re-initialization both are done in the same line i.e at the time of Object creation. [08-OCT]

With Constructor approach, we need not to depend on method to re-initialize our instance variable with user value.

Defination of Constructor :

It is called constructor because it is used to construct the object at runtime.

If the name of the class and name of the method both are exactly same without any return value then it is called constructor.

If a user will specify explicit return type then it will become method and we need to call explicitly.

The main purpose of constructor to initialize the instance variable of the class with user value.

A constructor never contain any return type even void.

Every java class must contain at-least one constructor, either implicitly added by java compiler OR explicitly written by user.

Every time we create an object by using new keyword, at-least one constructor must be invoked.

Constructors are automatically called and executed at the time of creating the object. [No need to call explicitly]

//Constructor containing return statement.

package com.ravi.data_hiding;

public class Test
{
        public Test()
        {
                System.out.println("Constructor");
                return; //valid
        }

        public static void main(String[] args)
        {
    Test t1 = new Test();
        }

}

If we put return type for Constructor then it will become as a method and we need to call explicitly.

package com.ravi.data_hiding;

public class Test
{
        public void Test()
        {

```java
            System.out.println("Method");
            return;
        }

        public static void main(String[] args)
        {
          System.out.println("Main");
      Test t1 = new Test();
      t1.Test(); //Calling Explicitly
        }
}
```

Types of Constructor in java :

Java suports 3 types of Constructor

1) Default no argument constructor

2) No Argument OR Parameter less OR non Parameterized OR Zero
   argument constructor.

3) Parameterized Constructor.
----
1) Default no argument constructor :

If a user does not provide any type of constructor in the class
then automatically compiler will add constructor in the class
known as Default no argument constructor.

Test.java

```java
public class Test
{
}
```

javac Test.java

Test.class
```java
public class Test
{
  public Test() //Default no argument constructor added by the
  {          //compiler
  }
```

}

2) No Argument OR Parameter less OR non Parameterized OR Zero
   argument constructor.

If a constructor is written by user without any parameter then it is called No Argument OR
Parameter less OR non Parameterized OR Zero argument constructor.

```java
public class Student
{
   private int rollNumber;
   public Student() //No Argument Constructor
   {
      rollNumber = 111;
   }
}
```

No argument constructor is not recommended to initialize our object properties because due to
no argument constructor all the object properties will be initialized with SAME VALUE as shown
in the program.

2 files :

Person.java

```java
package com.ravi.no_arg;

public class Person
{
        private int personId;
        private String personName;

        public Person() //No Argument constructor
        {
                personId = 111;
                personName = "Scott";
        }

        @Override
        public String toString() {
                return "Person [personId=" + personId + ", personName=" + personName + "]";
        }
```

```
        }

NoArgumentConstructor.java
----------------------------
package com.ravi.no_arg;

public class NoArgumentConstructor
{
        public static void main(String[] args)
        {
                Person scott = new Person();
                System.out.println(scott);

                Person smith = new Person();
                System.out.println(smith);

        }

}
```

Note : Here we have created two objects scott and smith but both the objects will be initialized with scott data so to initialize all the objects with different types of data we should use parameterized constructor.

Parameterized Constructor :

If we pass one or more argument to the constructor then it is called parameterized constructor.

By using parameterized constructor all the objects will be initialized with different values.

Example :

```
public class Employee
{
  int id;
  String name;

  public Employee(int id, String name)
  {
    this.id = id;
    this.name = name;
  }
}
```

2 files :

Lion.java

```java
package com.ravi.parameterized;

public class Lion
{
        private String lionName;
        private double lionHeight;
        private String lionColor;


        public Lion(String lionName, double lionHeight, String lionColor)
        {
                super();
                this.lionName = lionName;
                this.lionHeight = lionHeight;
                this.lionColor = lionColor;
        }



        @Override
        public String toString() {
                return "Lion [lionName=" + lionName + ", lionHeight=" + lionHeight + ",
lionColor=" + lionColor + "]";
        }

}
```

ParameterizedConstructor.java

```java
package com.ravi.parameterized;

public class ParameterizedConstructor {

        public static void main(String[] args)
        {
                Lion tiger = new Lion("Tiger",4.2,"White");
                System.out.println(tiger);
```

```java
            Lion simba = new Lion("Simba", 4.4, "Brown");
            System.out.println(simba);
        }

}
```

How to write a setter and getter :

```java
public class Customer
{
   private double customerBill;

   //Initialize the customerBill using parameterized constructor
   public Customer(double customerBill)
   {
      this.customerBill = customerBill;
   }

   //Writing setter to modify the existing customerBill
     public void setCustomerBill(double customerBill)
     {
       this.customerBill = customerBill;
     }

   //Writing getter to retrieve the private data value outside of BLC class
   public double getCustomerBill()
   {
      return this.customerBill;
   }
}
```

FINAL CONCLUSION :

Parameterized Constructor : To initialize the Object properties with user values.

Setter : To modify the existing object data.[Only one data at a time] OR Writing Operation

Getter : To read/retrieve private data value outside of BLC class. [Reading Operation]

Program on setter and getter :

Employee.java

```java
package com.ravi.setter_getter;

public class Employee
{
    private String employeeName;
    private double employeeSalary;

    public Employee(String employeeName, double employeeSalary) {
        super();
        this.employeeName = employeeName;
        this.employeeSalary = employeeSalary;
    }

    public String getEmployeeName() //getter
    {
        return employeeName;
    }

    public void setEmployeeName(String employeeName) //setter
    {
        this.employeeName = employeeName;
    }

    public double getEmployeeSalary() //getter
    {
        return employeeSalary;
    }

    public void setEmployeeSalary(double employeeSalary) //setter
    {
        this.employeeSalary = employeeSalary;
    }

    @Override
    public String toString()
    {
        return "Employee [employeeName=" + employeeName + ", employeeSalary=" +
employeeSalary + "]";
    }

}
```

EmployeeDemo.java

```java
package com.ravi.setter_getter;

public class EmployeeDemo {

        public static void main(String[] args)
        {
                Employee scott = new Employee("Scott", 50000);
                System.out.println(scott);



                scott.setEmployeeSalary(scott.getEmployeeSalary()+30000);
                System.out.println(scott);

                /*Based on following salary criteria we will decide whether
                scott is a developer OR Designer OR Tester
                salary >= 75000  [Developer]
                salary >= 40000  [Designer]
                salary >=250000  [Tester]
        */

                double empSal = scott.getEmployeeSalary();
                String empName = scott.getEmployeeName();

                if(empSal >=75000)
                {
                        System.out.println(empName+" is a Developer");
                }
                else if(empSal >=40000)
                {
                        System.out.println(empName+" is a Designer");
                }
                else
                {
                        System.out.println(empName+" is a Tester");
                }
        }

}
```

*** What is Encapsulation

[Accessing our private data with public methods like setter and getter]

Binding the private data with its associated method in a single unit is called Encapsulation.

Encapsulation ensures that our private data (Object Properties) must be accessible via public methods like setter and getter.

It provides security because our data is private (Data Hiding) and it is only accessible via public methods WITH PROPER DATA VALIDATION.

In java, class is the example of encapsulation.

How to achieve encapsulation in a class :

In order to achieve encapsulation we should follow the following two
techniques :

1) Declare all the data members with private access modifiers (Data Hiding OR Data Security)

2) Write public methods to perform read(getter) and write(setter) operation on these private data like setter and getter.

Note : If we decalre all our data with private access modifier then it is called TIGHTLY ENCAPSULATED CLASS. On the other hand if we declare our data other then private access modifier then it is called Loosely Encapsulated class.

Example of Tightly Encapsulated class :
----
public class Student
{
  private int rollNumber;
  private String studentName;
  private int studentMarks;
}

Example of Loosely Encapsulated class :
----
public class Emplouee
{
  private int employeeNumber;

```
   protected String employeeName;
   protected double employeeSalary;
}
```
==
14-10-2024

Method return type as a class :

While declaring a method in java, return type is compulsory.
As a method return type we have following options

1) void as a retutn type of the Method

2) Any primitive data type as a return type of the method.

3) Any class name/interface / enum / record we can take as a return type of the method.

```
package com.ravi.method_return_type;

public class Demo
{

        public Demo m1()
        {
                return null;
                   OR
                return this;
                   OR
                return new Demo();
        }
}
```

Program that describes, If the method return type is class then we depnd upon the constructor
of the class.


```
package com.ravi.method_return_type;

public class Demo
{
        public Demo(int x)
        {
```

```
                }

                public Demo m1()
                {
                        return new Demo(5); //It depends upon class constructor
                }


}

Note : Here the return value depends upon the available constructor in the class.
================================================================
What is a Factory Method :
--------------------------
If a method return type is class name menas it is returning the Object of the class then it is
called Factory Method.
---------------------------------------------------------------
2 files :
----------
Book.java
---------
package com.ravi.method_return_type;

public class Book
{
        private String bookTitle;
        private String authorName;

        public Book(String bookTitle, String authorName)
        {
                super();
                this.bookTitle = bookTitle;
                this.authorName = authorName;
        }

        public static Book getBookObject()
        {
                return new Book("Java", "James Gosling");
        }

        @Override
        public String toString() {
```

```
                return "Book [bookTitle=" + bookTitle + ", authorName=" + authorName + "]";
        }

}

BookDemo.java
--------------
package com.ravi.method_return_type;

public class BookDemo {

        public static void main(String[] args)
        {
                Book object = Book.getBookObject();
                System.out.println(object);

        }

}
```

Note : By using getBookObject() method we can get the Object of Book class but methods are meant for Re-usability purpose as shown in the program below.
----
2 files :

Employee.java

```
package com.ravi.method_return_type;

import java.util.Scanner;

public class Employee
{
        private int employeeId;
        private String employeeName;
        private double employeeSalary;

        public Employee(int employeeId, String employeeName, double employeeSalary) {
                super();
                this.employeeId = employeeId;
                this.employeeName = employeeName;
                this.employeeSalary = employeeSalary;
        }
```

```java
        @Override
        public String toString() {
                return "Employee [employeeId=" + employeeId + ", employeeName=" +
employeeName + ", employeeSalary="
                                + employeeSalary + "]";
        }

        public static Employee getEmployeeObject()
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter Employee Id :");
                int id = sc.nextInt();

                System.out.print("Enter Employee Name :");
                String name = sc.nextLine();
                name = sc.nextLine();

                System.out.print("Enter Employee Salary :");
                double salary = sc.nextDouble();

                return new Employee(id, name, salary);
        }

}
```

EmployeeDemo.java

```java
package com.ravi.method_return_type;

import java.util.Scanner;

public class EmployeeDemo
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("How many objects :");
                int noOfObj = sc.nextInt();

                for(int i=1; i<=noOfObj; i++)
                {
                        Employee employeeObject = Employee.getEmployeeObject();
```

```
                        System.out.println(employeeObject);
                }
                sc.close();
        }


}
-------------------------------------------------------------
Constructor Overloading :
-------------------------
In the same class if we write more than one constructor where parameter must be different (If
same, compilation error will be generated) then it is called constructor Overloading.

In order to call overloaded constructor we need not to create multiple objects, we can call all the
overloaded constructors with one object only by using this() [this of]

this() is used to call current class overloaded constrcutor and it must be FIRST STATEMENT OF
THE CONSTRUCTOR BODY.

2 files :
----------
package com.ravi.method_return_type;

public class Calculate
{
  public Calculate(double x, double y)
  {
        this(10,20,30);
        System.out.println("Addition of two double is :"+(x+y));

  }

   public Calculate(int x, int y, int z)
   {
        this();
        System.out.println("Addition of three integer is :"+(x+y+z));

   }

   public Calculate()
   {
        System.out.println("Non parameterized Constructor...");
   }
```

}

ConstructorOverloading.java
----
package com.ravi.method_return_type;

public class ConstructorOverloading {

        public static void main(String[] args)
        {
                new Calculate(3.2, 4.1); //Nameless OR Anonymous object
        }
}

Note : In order to call overloaded constructor, we need only 1 object.

What is Shallow and Deep copy in java :

Shallow Copy :

In Shallow copy, Only one Object will be created but the same object will be refered by multiple
reference variables.

If we modify the object properties by any of the reference variable then original object will be
modified as shown in the program.

2 files :

Laptop.java
----
package com.ravi.shallow_copy;

public class Laptop
{
        private String laptopBrand;
        private double laptopPrice;

        public Laptop(String laptopBrand, double laptopPrice)
        {
                super();
                this.laptopBrand = laptopBrand;
                this.laptopPrice = laptopPrice;
        }

```java
        public String getLaptopBrand() {
                return laptopBrand;
        }

        public void setLaptopBrand(String laptopBrand) {
                this.laptopBrand = laptopBrand;
        }

        public double getLaptopPrice() {
                return laptopPrice;
        }

        public void setLaptopPrice(double laptopPrice) {
                this.laptopPrice = laptopPrice;
        }

        @Override
        public String toString() {
                return "Laptop [laptopBrand=" + laptopBrand + ", laptopPrice=" + laptopPrice +
"]";
        }

}
```

ShallowCopy.java

```java
package com.ravi.shallow_copy;

public class ShallowCopy {

        public static void main(String[] args)
        {
                Laptop laptop1 = new Laptop("Dell", 80000);
                Laptop laptop2 = laptop1;

                System.out.println(laptop1 +" : "+laptop2);
                System.out.println("After Modification");

                laptop2.setLaptopBrand("HP");
                laptop2.setLaptopPrice(120000);
                System.out.println(laptop1 +" : "+laptop2);
```

```
        }

}
```

Deep Copy :

In deep copy two different objects will be created, the 2nd object will copy the content of first object.

If we modify the object by using reference variable then only one object will be modified as shown below.

2 files :

Customer.java

```java
package com.ravi.deep_copy;

public class Customer
{
        private int customerId;
        private String customerName;

        public Customer()
        {
                customerId = 0;
                customerName = null;
        }

        public Customer(int customerId, String customerName)
        {
                super();
                this.customerId = customerId;
                this.customerName = customerName;
        }




        public int getCustomerId() {
                return customerId;
```

```java
        }

        public void setCustomerId(int customerId) {
                this.customerId = customerId;
        }

        public String getCustomerName() {
                return customerName;
        }

        public void setCustomerName(String customerName) {
                this.customerName = customerName;
        }

        @Override
        public String toString() {
                return "Customer [customerId=" + customerId + ", customerName=" +
customerName + "]";
        }




}
```

DeepCopy.java

```java
package com.ravi.deep_copy;

public class DeepCopy {

        public static void main(String[] args)
        {
                Customer c1 = new Customer(111, "Virat");
                Customer c2 = new Customer();
                c2.setCustomerId(c1.getCustomerId());
                c2.setCustomerName(c1.getCustomerName());

                System.out.println(c1);
                System.out.println(c2);

                System.out.println("After Modification");
```

```
                c2.setCustomerId(999);
                c2.setCustomerName("Rohit");
                System.out.println(c1);
                System.out.println(c2);



        }

}
```

==
15-10-2024

**Pass by Value :

Java does not support pointers concept so, java works with pass by value only.

Pass by value means we are passing the copy of original data to the method so the method will get only the copy of the data.

```
package com.ravi.pass_by_value;

public class PassByValueDemo1
{
        public static void main(String[] args)
        {
                int x = 100;
                accept(x);
                System.out.println(x);
        }

        public static void accept(int y)
        {
                y = 200;
        }

}
```

Here x will provide the output as 100 because we are only passing the copy of x to y variable.

```
package com.ravi.pass_by_value;

public class PassByValueDemo2
```

```java
{
        public static void main(String[] args)
        {
                int a = 100;
                a = accept(a);
                System.out.println(a);
        }

        public static int accept(int x)
        {
                x = 500;
                return x;
        }

}
```

```java
package com.ravi.pass_by_value;

class Employee
{
        private double employeeSalary = 40000;

        public double getEmployeeSalary()
        {
                return employeeSalary;
        }

        public void setEmployeeSalary(double employeeSalary)
        {
                this.employeeSalary = employeeSalary;
        }
}


public class PassByValueDemo3 {

        public static void main(String[] args)
        {
                Employee e1 = new Employee();
                modifySalary(e1);
                System.out.println(e1.getEmployeeSalary());
        }
```

```java
        public static void modifySalary(Employee emp)
        {
                emp.setEmployeeSalary(90000);
        }
}
```

Note : Same object is refered by multiple reference variable so it is a shallow copy concept
hence we will get the output 90000

```java
package com.ravi.pass_by_value;

class Product
{
        private double productPrice = 2000;

        public double getProductPrice()
        {
                return productPrice;
        }

        public void setProductPrice(double productPrice)
        {
                this.productPrice = productPrice;
        }
}

public class PassByValueDemo4 {

        public static void main(String[] args)
        {
                Product p1 = new Product();
                accept(p1);
                System.out.println(p1.getProductPrice());

        }

        public static void accept(Product prod)
        {
                prod = new Product();
                prod.setProductPrice(15000);
        }
}
```

Note : Here another object is created and it is assigned to prod variable (deep copy) so the changes will done in the
newly created object but not the existing object so, the output is 2000.

What is Grabge Collection in Java ?

In C++ langauge, Developer is responsible to allocate the memory as well as de-allocate the memory otherwise OutOfMemoryError will encounter.

In java, Developer is only responsible to allocate the memory, memory de-allocation is not the responsibility of the developer because java provides automatic memory management technique.

Garbage Collector(Component of JVM) provides automatic memory management because It scans the heap area, identify which objects are eligible for Garbage Collector(The objects which does not contain any references) and delete those objects from the heap memory so we can re-use heap memory for another object.

Garbage Collector is a deamon thread which uses Mark and sweep algorithm to delete the un-used object from the Heap memory.

How many ways we can make an object eligible for GC(Garbage

Collection)

There are 3 ways we can make an object eligible for GC.

1) Assigning null literal to existing reference variable :
   Employee e1 = new Employee(111,"Ravi");
        e1 = null;

2) Creating an Object inside a method :

   public void createObject()
   {
     Employee e2 = new Employee();
   }
Here we are creating Employee object inside the method so, once the method execution is over then e2 will be deleted from the Stack Frame and the employee object will become eligible for GC.

3) Assigning new Object to the old existing reference variable:

Employee e3 = new Employee();
    e3 = new Employee();

Earlier e3 variable was poting to Employee object after that a new Employee Object is created
which is pointing to another memory location so the first object is eligible for GC.

Memory in java :

In java, whenever we create an object then Object and its content (properties and behavior) are
stroed in a special memory called HEAP Memory. Garbage collector visits heap memory only.

All the local variables and parameters variables are executed in Stack Frame and available in
Stack Memory.

16-10-2024

HEAP and STACK Diagram :

Create a HEAP and STACK Digram for CustomerDemo.java
----
class Customer
{
        private String name;
        private int id;

        public Customer(String name , int id)
        {
                super();
                this.name=name;
                this.id=id;
        }

        public void setId(int id) //setter
        {
                this.id=id;
        }

        public int getId() //getter
        {
                return this.id;
        }
}

```java
public class CustomerDemo
{
        public static void main(String[] args)
        {
                int val = 100;

                Customer c = new Customer("Ravi",2);

        m1(c);

                //GC [1 Object i.e 3000x is eligible for GC]

                System.out.println(c.getId());
        }

        public static void m1(Customer cust)
        {
                cust.setId(5);

            cust = new Customer("Rahul",7);

                cust.setId(9);
                System.out.println(cust.getId());
        }
}

//Output 9 5
```

Note : String objects are not eligible for GC.

HEAP and STACK Digram for Sample.java

```java
public class Sample
{
        private Integer i1 = 900;

        public static void main(String[] args)
        {
                Sample s1 = new Sample();

                Sample s2 = new Sample();

            Sample s3 = modify(s2);
```

```
                s1 = null;

        //GC [4 objects, 1000x, 2000x, 5000x and 6000x eligible 4 GC]

                        System.out.println(s2.i1);
                }
    public static Sample modify(Sample s)
            {
                        s.i1=9;
                        s = new Sample();
                        s.i1= 20;
            System.out.println(s.i1);
                        s=null;
                        return s;
                }
}

//20 9


------------------------------------------------------------
public class Employee
{
        int id = 100;

        public static void main(String[] args)
        {
                        int val = 200;

                        Employee e1 = new Employee();

                        e1.id = val;

                        update(e1);

                        System.out.println(e1.id);

            Employee e2 = new Employee();

                        e2.id = 900;

                        switchEmployees(e2,e1); //3000x and 1000x
```

```java
                    //GC [2 objects, 2000x and 4000x are eligible for GC]

                        System.out.println(e1.id);
                    System.out.println(e2.id);
            }

        public static void update(Employee e)
        {
    e.id = 500;
                e = new Employee();
                e.id = 400;
                System.out.println(e.id);
        }

        public static void switchEmployees(Employee e1, Employee e2)
        {
                int temp = e1.id;
                e1.id = e2.id; //500
                e2 = new Employee();
                e2.id = temp;
        }
  }
```

//Output 400 500 500 500

HEAP and STACK Diagram for Test.java

```java
public class Test
{
        Test t;
        int val;

        public Test(int val)
        {
                this.val = val;
        }

        public Test(int val, Test t)
        {
                this.val = val;
                this.t = t;
        }
```

```java
        public static void main(String[] args)
        {
                Test t1 = new Test(100);

                Test t2 = new Test(200,t1);

                Test t3 = new Test(300,t1);

                Test t4 = new Test(400,t2);

                t2.t = t3;
                t3.t = t4;
                t1.t = t2.t;
                t2.t = t4.t;

        System.out.println(t1.t.val);
        System.out.println(t2.t.val);
        System.out.println(t3.t.val);
        System.out.println(t4.t.val);
        }

}
```
Output :
300
200
400
200

Heap and Stack Diagram for Test.java

```java
class Test
{
int x;
int y;

void m1(Test t)
{
x=x+1;
y=y+2;
t.x=t.x+3;
t.y=t.y+4;
}
public static void main(String[] args)
```

```
{
Test t1=new Test();  // x and y  m1()
Test t2=new Test();  // x and y  m1()

t1.m1(t2);

System.out.println(t1.x+"... "+t1.y); // 1 ... 2
System.out.println(t2.x+"... "+t2.y); // 3 ... 4

t2.m1(t1);
System.out.println(t1.x+"... "+t1.y);
System.out.println(t2.x+"... "+t2.y);

t1.m1(t1);
System.out.println(t1.x+"... "+t1.y);
System.out.println(t2.x+"... "+t2.y);

t2.m1(t2);
System.out.println(t1.x+"... "+t1.y);
System.out.println(t2.x+"... "+t2.y);
}
}
```

17-10-2024

Passing an Object reference to the Constructor :(Copy Constructor)

We can pass an Object reference to the constructor, to copy the content of one object to another object.

```
public class Manager
{
        private int managerId;
        private String managerName;

        public Manager(Employee emp)
        {
                this.managerId = emp.getEmployeeId();
                this.managerName = emp.getEmployeeName();
        }
}
```

By using copy constructor we can copy the properties of one object to another object, In the

above example we have initialized the properties of Manager class with the help of Employee class properties.

Program on Copy Constructor :

----

3 files :

Employee.java

```java
package com.ravi.copy_constructor;

public class Employee
{
        private int employeeId;
        private String employeeName;

        public Employee(int employeeId, String employeeName)
        {
                super();
                this.employeeId = employeeId;
                this.employeeName = employeeName;
        }

        @Override
        public String toString() {
                return "Employee [employeeId=" + employeeId + ", employeeName=" +
employeeName + "]";
        }

        public int getEmployeeId() {
                return employeeId;
        }

        public String getEmployeeName() {
                return employeeName;
        }
}


Manager.java

-------------

package com.ravi.copy_constructor;
```

```java
public class Manager
{
        private int managerId;
        private String managerName;

        public Manager(Employee emp)  //emp = e1
        {
                this.managerId = emp.getEmployeeId();
                this.managerName = emp.getEmployeeName();
        }

        @Override
        public String toString() {
                return "Manager [managerId=" + managerId + ", managerName=" +
managerName + "]";
        }

}
```

CopyConstructorDemo.java

```java
package com.ravi.copy_constructor;

public class CopyConstructorDemo {

        public static void main(String[] args)
        {
                Employee e1 = new Employee(111, "Scott");
                Manager m1 = new Manager(e1);
                System.out.println(m1);
        }
}
```

In the above program we have initialized manager class properties by using employee class
properties, In the next program we will initialize same class properties in another constructor.

2 files :

Product.java

```java
package com.ravi.copy_constructor;
```

```java
public class Product
{
        private int productId;
        private String productName;

        public Product(int productId, String productName)
        {
                super();
                this.productId = productId;
                this.productName = productName;
        }

        public Product(Product prod) //Parameterized Constructor
        {
                this.productId = prod.productId;
                this.productName = prod.productName;
        }

        @Override
        public String toString() {
                return "Product [productId=" + productId + ", productName=" + productName +
"]";
        }

}

ProductCopyConstructor.java
----------------------------
package com.ravi.copy_constructor;

public class ProductCopyConstructor {

        public static void main(String[] args)
        {
                Product p1 = new Product(111, "Laptop");
                System.out.println(p1);

                Product p2 = new Product(p1);
                System.out.println(p2);
        }

}
--------------------------------------------------------------
```

Lab Program :

Lab Program :(Method return type as a class + Passing Object ref)

A class called Customer is given to you.

The task is to find the applicable Credit card Type and create CardType object based on the Credit Points of a customer.

Define the following for the class.

Attributes :
        customerName : String,private
        creditPoints: int, private

Constructor :
        parameterizedConstructor: for both cusotmerName & creditPoints in that order.

Methods :
        Name of the method : getCreditPoints
        Return Type : int
        Modifier   : public
        Task : This method must return creditPoints

        Name of the method : toString, Override it,
        Return type : String
        Task : return only customerName from this.

Create another class called CardType. Define the following for the class

Attributes :
        customer : Customer, private
        cardType : String, private
Constructor :
        parameterizedConstructor: for customer and cardType attributes in that order

Methods :
        Name of the method : toString Override this.
        Return type : String
        Modifier : public
        Task : Return the string in the following format.
                The Customer 'Rajeev' Is Eligible For 'Gold' Card.

Create One more class by name CardsOnOffer and define the following for the class.

Method :

       Name Of the method : getOfferedCard
       Return type : CardType
       Modifiers: public,static
       Arguments: Customer object

       Task : Create and return a CardType object after logically finding cardType from creditPoints as per the below rules.

creditPoints		cardType
100 - 500	-	Silver
501 - 1000	-	Gold
1000 >	-	Platinum
< 100	-	EMI

Create an ELC class which contains Main method to test the working of the above.

4 files :

Customer.java
----
package com.ravi.lab_credit_card;

public class Customer
{
       private String customerName;
       private int creditPoints;

       public Customer(String customerName, int creditPoints)
       {
              super();
              this.customerName = customerName;
              this.creditPoints = creditPoints;
       }

       public int getCreditPoints()
       {
              return this.creditPoints;
       }

       @Override

```java
        public String toString()
        {
                return this.customerName;
        }


}

CardType.java
--------------
package com.ravi.lab_credit_card;

public class CardType
{
        private Customer customer; // HAS-A relation
        private String cardType;

        public CardType(Customer customer, String cardType)
        {
                super();
                this.customer = customer;
                this.cardType = cardType;
        }

        @Override
        public String toString()
        {
                //The Customer 'Rajeev' Is Eligible For 'Gold' Card.
                return "The Customer '"+this.customer+"' Is Eligible For '"+this.cardType+"' Card";
        }

}

CardsOnOffer.java
------------------
package com.ravi.lab_credit_card;

public class CardsOnOffer
{
    public static CardType getOfferedCard(Customer obj)
    {
        int creditPoint = obj.getCreditPoints();
```

```java
        if(creditPoint >=100 && creditPoint <=500)
        {
                return new CardType(obj, "Silver");
        }
        else if(creditPoint >500 && creditPoint <=1000)
        {
                return new CardType(obj, "Gold");
        }
        else if(creditPoint > 1000)
        {
                return new CardType(obj, "Platinum");
        }
        else
        {
                return new CardType(obj, "EMI");
        }

    }
}


package com.ravi.lab_credit_card;

import java.util.Scanner;

public class CreditCardDemo
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter Customer Name :");
                String name = sc.nextLine();
                System.out.print("Enter Customer Credit Points  :");
                int crediPoint = sc.nextInt();

                Customer cust = new Customer(name, crediPoint);

                CardType offeredCard = CardsOnOffer.getOfferedCard(cust);
                System.out.println(offeredCard);
                sc.close();


        }
```

```
}
--------------------------------------------------------------
Modifiers on Constructor :
--------------------------
Constructor can accept all types of access modifiers which are apllicable for accessibility level
like private, default, protected and public.

package com.ravi.constructor;

public class Sample
{
        private int x,y;

        private Sample(int x, int y) //Private Constructor
        {
                this.x = x;
                this.y = y;
        }


        @Override
        public String toString() {
                return "Sample [x=" + x + ", y=" + y + "]";
        }

        public static void main(String[] args)
        {
                Sample s = new Sample(12, 24);
                System.out.println(s);

        }

}
--------------------------------------------------------------
package com.ravi.constructor;

public class Demo
{
    public Demo()
    {
        System.out.println("No Argument Constructor");
        return;  //Valid
    }
```

```
        public static void main(String[] args)
        {
                new Demo(); //Nameless OR Anonymous Object
        }

}
```

Note : We can't apply static, final, abstract and synchronized modifier on constructor.

IQ :

Why we should declare a constructor with private Access modifier ?

We can declare a constructor with private access modifier due to the following two reasons :

1) If we want to declare only static variables and static methods inside a class then we should decalre constructor with private access modifier [Object is not required]

Example : java.util.Arrays and Math class has private constructor because It contains only static method.

2) If we want to develop singleton class, for singleton class we should create only one object by the class itself then we should declare the constructor with private access modifier.

```
package com.ravi.constructor;

public class Foo
{
  private Foo() //Private Constructor
  {
        System.out.println("It is a private Constructor");
  }

  public static void main(String [] args)
  {
        new Foo();
  }

}
```

18-10-2024

What is an instance block OR instance initializer in java ?

It is a special block in java which is automatically executed at the time of creating the Object.

Example :

```
{
   //Instance OR Non static block
}
```

If a constructor contains first line as a super() statement then only compiler will add instance block in the 2nd line of constructor otherwise it will not be added by the compiler.

If constructor contains super() then non static block will be executerd before the body of the constructor.

The main purpose of instance block to initialize the instance variables (So It is called Instance Iniatilizer) of the class OR to write a common logic which will be applicable to all the objects.

If a class contains multiple non static block then it will be executed according to the order [Top to bottom]

Instance initializer must be executed normally that menas we can't interrupt the execution flow of any initializer hence we can't write return statement inside non static block.

If a user defines non static block after the body of the constructor then compiler will not placed in the 2nd line of the constructor. It will be executed as it is because compiler will search the NSB in the class level.
----
package com.ravi.instance_demo;

```
class Test
{
        public Test()
        {
                System.out.println("No Argument Constructor...");
        }


        {
                System.out.println("Non static Block1");
        }


        {
```

```
                System.out.println("Non static Block2");
        }

}

public class InstanceBlockDemo1
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                Test t2 = new Test();
        }

}
```

Note : NSB will be exected before the constructor body.

```
package com.ravi.instance_demo;

class Demo
{
        public Demo()
        {
                this(10);
                System.out.println("No Argument Constructor");
        }

        public Demo(int x)
        {
                System.out.println("Parameterized Constructor :"+x);
        }

        {
                System.out.println("Non static Block");
        }
}


public class InstanceBlockDemo2
{
        public static void main(String[] args)
        {
                new Demo();
```

```
        }

}
```

Note : Compiiler will add NSB in the constructor which contains
       super()
----
package com.ravi.instance_demo;

```java
class Sample
{
        int x;

        public Sample()
        {
                System.out.println("x value is :"+x);
        }

        {
                x = 100;
                 System.out.println("Object creation in Progress....");
        }

}

public class InstanceBlockDemo3 {

        public static void main(String[] args)
        {
                Sample s1 = new Sample();
                Sample s2 = new Sample();

        }

}
```

Note : Can initialize the instance variable using NSB
----
package com.ravi.instance_demo;

```java
class Foo
{
        int x;
```

```java
        public Foo()
        {
                x = 400;
                System.out.println("x value is :"+x);
        }


        {
                x = 100;
                System.out.println("x value is :"+x);
        }


        {
                x = 200;
                System.out.println("x value is :"+x);
        }


        {
                x = 300;
                System.out.println("x value is :"+x);
        }
}

public class InstanceBlockDemo4
{
        public static void main(String[] args)
        {
                new Foo();
        }

}
```

Note : Execution of the NSB from top to bottom

```java
package com.ravi.instance_demo;

public class InstanceBlockDemo5
{
        {
                //return;  //Invalid [Must be executed Normally]
        }
```

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

    }

}
```

Note : Initializer must be executed normally

```
package com.ravi.instance_demo;

class MyDemo
{
    public MyDemo()
    {
        System.out.println("My Demo Constructor");

        {
            System.out.println("Non static Block");
        }
    }
}

public class InstanceBlockDemo6
{
    public static void main(String[] args)
    {
        new MyDemo();
    }

}
```

Note : If we wtite NSB in the constructor body then it will be executed as it is.
==
Order of instance variable initialization in the program life cycle :

All the instance variables are initialized in the following order during the life cycle :

1) It will initialized with default value at the time of Object
   creation. [new Demo(); Demo class instance variable will be initialized, init method is working
internally]

2) Now control will verify whether, we have initailized at the time of variable declaration or not.

3) Now control will verify whether, we have initailized
   inside non static block or not. [If present]

4) Now control will verify whether, we have initailized in the
   body of the constructor or not.

5) Now control will verify whether, we have initailized in the method body or not but it is not recommended because Object is already created, we need to call the method explicitly, It is not the part of the object.

Default value => At the time of declaration => in the body of non static block => in the body of constructor => Inside method
body [Not Recommended]

```java
package com.nit.oop;

class Sample
{
        int x = 90;  //1st STEP

        public Sample()
        {
                super();
                x = 900; //STEP 3
        }


        {
                x = 120;  //STEP 2
        }

}


public class InstanceVeriableInitializationOrder {

        public static void main(String[] args)
        {
                Sample s1 = new Sample();
                System.out.println("x value is :"+s1.x);


        }
```

}

19-10-2024

What is blank final field in java ?

If a final instance variable is not initialized at the time of declaration then it is called blank final field.

final int A ; //Blank final field

A final variable must have user-defined value.

A blank final field can't be initialized by default constrcutor.

A blank final field must be explicitly initialized by the user till the execution of constructor body.[Till Object creation].

It can be iniatialized in the following two places :
   a) Inside a non static block [If available]
   b) Inside the constructor body

A blank final field can't be initialized by method.[Object creation is already Over]

*A blank final field can also have default value.

A blank final must be iniatilized explicitly by user in all the constructors available in the class.

```java
class Test
{
        final int A;
}

public class BlankFinalField
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                System.out.println(t1.A);
        }
}
```

Note : It can't be initialized by default constructor.

```
class Test
{
        final int A;


        {
                A = 100;
        }
}


public class BlankFinalField
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                System.out.println(t1.A);
        }
}
```

Note : We can initialize the blank final field inside the non static block.

```
class Test
{
        final int A;



        public Test()
        {
                A = 200;
        }

}


public class BlankFinalField
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                System.out.println(t1.A);
        }
}
```

Note : We can initialize the blank final field inside the Constructor body [till object creation]

```java
class Test
{
        final int A;


        {
                m1();
                A = 200;
        }

        public void m1()
        {
                System.out.println("Default Value is :"+A);
        }

}

public class BlankFinalField
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                System.out.println("User value is :"+t1.A);
        }
}
```

Note : A blank final field also have default value.

```java
class Test
{
        final int A;

        public Test()
        {
          A = 100;
        }

        public Test(int x)
        {
                A = x;
        }
}
```

```
public class BlankFinalField
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                System.out.println(t1.A);

                Test t2 = new Test(200);
                System.out.println(t2.A);
        }
}
```

Blank final field nust be initialized by all the constructors available in the class.

Relationship between the classes :

In java, We have 2 types of relationship between the classes :

  1) IS-A Relation
  2) HAS-A Relation

Example of IS-A Relation :

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

Note : Car is a Vehicle (In between Car and Vehicle we have IS-A
       relation)

Example of HAS-A relation :

```
class Car
{
   private Engine engine;
}
```

IS-A relation we can achieve by using Inheritance Concept.
HAS-A relation we can achieve by Association Concept.
==
Inheritance (IS-A Relation) :

Deriving a new class (child class) from existing class (parent class) in such a way that the new class will acquire all the properties and features (except private) from the existing class is called inheritance.

It is one of the most imporatnt feature of OOPs which provides "CODE REUSABILITY".

Using inheritance mechanism the relationship between the classes is parent and child.
According to Java the parent class is called super class and the child class is called sub class.

In java we provide inheritance using 'extends' keyword.

*By using inheritance all the feature of super class is by default available to the sub class so the sub class need not to start the process from begning onwards.

Inheritance provides IS-A relation between the classes. IS-A relation is tightly coupled relation (Blood Relation) so if we modify the super class content then automatically sub class content will also modify.

Inheritance provides us hierarchical classification of classes, In this hierarchy if we move towards upward direction more generalized properties will occur, on the other hand if we move towards downwand more specialized properties will occur.

21-10-2024

Types of Inheritance :

Java supports 5 types of Inheritance :

 1) Single Level Inheritance.
 2) Multilevel Inheritance.
 3) Hierarchical Inheritance.
 4) Multiple Inheritance (Not Support by class)
 5) Hybrid Inheritance.
==
Program on Single Level Inheritance :

SingleLevelDemo.java

```
package com.ravi.single_level_inheritance;

class Father
{
        public void house()
        {
                System.out.println("3 BHK House");
        }
}
class Son extends Father
{
        public void Car()
        {
                System.out.println("Audi Car");
        }
}

public class SingleLevelDemo
{
        public static void main(String[] args)
        {
                Son raj = new Son();
                raj.house();
                raj.Car();
        }

}
```

Note : Inheritance follows top to bottom approach, In inheritance we should always create the object for more specialized class.

SingleLevelDemo1.java[Encapsulation]
----
package com.ravi.single_level_inheritance;

```
class Super
{
  private int x;  //Data Hiding
  private int y;

  public void setData(int x, int y)
  {
        this.setX(x);
```

```java
                this.setY(y);
    }

        public int getX()
        {
                return x;
        }

        public void setX(int x)
        {
                this.x = x;
        }

        public int getY() {
                return y;
        }

        public void setY(int y) {
                this.y = y;
        }
}
class Sub extends Super
{
        public void showData()
        {
                System.out.println("x value is :"+getX());
                System.out.println("y value is :"+getY());
        }
}

public class SingleLevelDemo1
{
        public static void main(String[] args)
        {
                Sub sub = new Sub();
                sub.setData(100, 200);
                sub.showData();
        }

}
```

How to initialize the super class properties :

In order to initialize the super class properties we should use super keyword in the sub class as a first line of constructor.

super keyword always referes to its immediate super class.

Just like this keyword, super keyword (non static member) also we can't use inside static context.

super keyword we can use 3 ways in java :
----
1) To access super class variable
2) To access super class method
3) To access super class constructor.

1) To access the super class variable (Variable Hiding) :

Whenever super class variable name and sub class variable name both are same then it is called variable Hiding, Here sub class variable hides super class variable.

In order to access super class variable i.e super class memory, we should use super keyword as shown in the program.

package com.ravi.single_level_inheritance;

class Father
{
        protected double balance = 50000;
}
class Son extends Father
{
        protected double balance = 18000; //Variable Hiding

        public void showBalance()
        {
                System.out.println("Son balance is :"+this.balance);
                System.out.println("Father Balance is :"+super.balance);
        }
}
public class VariableHiding
{
        public static void main(String[] args)
        {

```
                Son s1 = new Son();
                s1.showBalance();


        }

}
```

22-10-2024

```java
package com.ravi.single_level_inheritance;

class Vehicle
{
        protected String engineType = "Petrol";
}

class Car extends Vehicle
{
        protected String typeOfEngine = "Battery";

        public void getEngineDetails()
        {
                System.out.println("Vehicle Engine type :"+this.engineType);
                System.out.println("Car Engine type :"+this.typeOfEngine);
        }

}
public class VariableHiding
{
        public static void main(String[] args)
        {
                Car car = new Car();
                car.getEngineDetails();
        }

}
```

Note : Here Vehicle class property (engineType) is by default available to Car class due to Inheritance so we can access using this keyword.

```java
package com.ravi.single_level_inheritance;

class Vehicle
```

```java
{
        protected String engineType = "Petrol";
}

class Car extends Vehicle
{
        protected String engineType = "Battery"; //Variable Hiding

        public void getEngineDetails()
        {
                System.out.println("Vehicle Engine type :"+super.engineType);
                System.out.println("Car Engine type :"+this.engineType);
        }
}
public class VariableHiding
{
        public static void main(String[] args)
        {
                Car car = new Car();
                car.getEngineDetails();
        }

}
```

Note : In the above Program we have Variale Hiding so super keyword is required.
----
2) To call super class method :

If the super class non static method name and sub class non static method name both are same
(Method Overriding) and if we create an object for sub class then sub class method will be
executed (bootom to top), if we want to call super class method from sub class method body
then we we should use super keyword as shown in the program.

```java
package com.ravi.multi_level;

class Alpha
{
        public void show()
        {
                System.out.println("Show Method in Alpha Class");
        }
}
```

```java
class Beta extends Alpha
{
        public void show()
        {
                super.show();
                System.out.println("Show Method in Beta Class");
        }

}
class Gamma extends Beta
{
        public void show()
        {
                super.show();
                System.out.println("Show Method in Gamma Class");
        }
}

public class MethodOverrding
{
        public static void main(String[] args)
        {
                Gamma g = new Gamma();
                g.show();


        }


}
```
----
*3) To call super class Constructor [Constructor Chaining]:

Whenever we write a class in java and we don't write any kind of constructor to the class then
the java compiler will automatically add one default constructor to the class.

THE FIRST LINE OF ANY CONSTRUCTOR IS RESERVERD EITHER FOR super() or this()
keyword that means first line of any constructor is used to call another constructor of either
same class OR super class.

In the first line of any constructor if we don't specify either super() or this() then the compiler will
automatically add super() to the first line of constructor.

Now the purpose of this super() [added by java compiler], to call the default constructor or

No-Argument constructor of the super class.

In order to call the constructor of super class as well as same class, we have total 4 cases.

Case 1:
----
super() : Automatically added by compiler to maintain the
        hierarchy in the first line of the Constructor. It
          is used to call default OR no argument constructor
          of super class.

ConstructorChainingDemo1.java
----
package com.ravi.constructor_chaining;

class A
{
        A()
        {
                System.out.println("A");
        }
}
class B extends A
{
        B()
        {
                System.out.println("B");
        }
}

public class ConstructorChainingDemo1
{
        public static void main(String[] args)
        {
                B b1 = new B();

        }

}

Case 2 :
----
super("Scott") : Must be explicitly written by user in the

first line of constructor. It is used to call
the parameterized constructor of super class.

ConstructorChaningDemo2.java

```java
package com.ravi.constructor_chaining;

class Alpha
{
        public Alpha(String name)
        {
                System.out.println("Your Name is :"+name);
        }
}
class Beta extends Alpha
{
        Beta()
        {
                super("Scott");
                System.out.println("No Argument constructor of Beta class");
        }
}
public class ConstructorChaningDemo2
{
        public static void main(String[] args)
        {
                new Beta();

        }

}
```

```java
package com.ravi.constructor_chaining;

class AA
{
        public AA()
        {
                System.out.println("No Argument Constructor of AA class");
        }
}
class BB extends AA
{
```

```
}
class CC extends BB
{
        public CC()
        {
                System.out.println("No Argument Constructor of CC class");
        }
}

public class IQ {

        public static void main(String[] args)
        {
                new CC();

        }

}
```

Output :
No Argument Constructor of AA class
No Argument Constructor of CC class

Case 3 :
----
this() : Must be explicitly written by user in the
        first line of constructor. It is used to call
         no argument constructor of current class.

```
package com.ravi.constructor_chaining;

class Super
{
        public Super()
        {
                System.out.println("No Argument Constructor of super class");
        }
        public Super(int x)
        {
                this();
                System.out.println("Parameterized Constructor of super class :"+x);
        }
}
class Sub extends Super
```

```java
{
        public Sub(int val)
        {
                super(val);
                System.out.println("Parameterized Constructor of Sub class ");
        }
}

public class ConstructorChaningDemo3 {

        public static void main(String[] args)
        {
                new Sub(12);

        }

}
```

Case 4 :
----
this("java") : Must be explicitly written by user in the
        first line of constructor. It is used to call
        parameterized constructor of current class.

```java
package com.ravi.constructor_chaining;

class Base
{
        public Base()
        {
                this(15);
                System.out.println("No Argument Constructor of Base class");
        }

        public Base(int x)
        {
                System.out.println("Parameterized Constructor of Base class :"+x);
        }
}

class Derived extends Base
{
        public Derived()
```

```java
        {
                System.out.println("No Argument Constructor of Derived class");
        }
}
public class ConstructorChaningDemo4 {

        public static void main(String[] args)
        {
                new Derived();

        }

}
```

===

```java
package com.ravi.constructor_chaining;

class Shape
{
        protected int x;

        public Shape(int x)
        {
                super();
                this.x = x;
                System.out.println("x value is :"+x);
        }
}
class Square extends Shape
{
        public Square(int side)  //10
        {
                super(side);
        }

        public void getAreaOfSquare()
        {
                double area = super.x * super.x;
                System.out.println("Area of Square is :"+area);
        }
}

public class SingleLevelDemo
{
```

```java
        public static void main(String[] args)
        {
                Square ss = new Square(10);
                ss.getAreaOfSquare();
        }


}
```

```java
package com.ravi.multi_level;

class Shape
{
        protected int x;

        public Shape(int x)
        {
                super();
                this.x = x;
                System.out.println("x value is :"+x);
        }
}
class Rectangle extends Shape
{
        protected int breadth;
        public Rectangle(int length, int breadth) //2   5
        {
                super(length);
                this.breadth = breadth;
        }

        public void getAreaOfRectangle()
        {
                double area = super.x * this.breadth;
                System.out.println("Area of Rectangle is :"+area);
        }
}
class Circle extends Shape
{
        final double PI = 3.14;
        public Circle(int radius)
        {
                super(radius);
        }
```

```java
        public void getAreaOfCircle()
        {
                double area = PI * super.x * super.x;
                System.out.println("Area of Circle is :"+area);
        }
}



public class MultiLevelDemo {

        public static void main(String[] args)
        {
                Rectangle rr = new Rectangle(2, 5);
                rr.getAreaOfRectangle();

                Circle cc = new Circle(4);
                cc.getAreaOfCircle();

        }

}
```

23-10-2024

Program on Single Level Inheritance :
----
3 files :

Employee.java

```java
package com.ravi.single_level_inheritance;

public class Employee
{
        protected int employeeId;
        protected String employeeName;
        protected String employeeAddress;

        public Employee(int employeeId, String employeeName, String employeeAddress)
        {
                super();
```

```java
                this.employeeId = employeeId;
                this.employeeName = employeeName;
                this.employeeAddress = employeeAddress;
        }

}
```

PermanentEmployee.java

```java
package com.ravi.single_level_inheritance;

public class PermanentEmployee extends Employee {
        protected String department;
        protected String designation;

        public PermanentEmployee(int employeeId, String employeeName, String
employeeAddress, String department,        String designation)
        {
                super(employeeId, employeeName, employeeAddress);
                this.department = department;
                this.designation = designation;
        }

        @Override
        public String toString() {
                return "PermanentEmployee [employeeId=" + employeeId + ", employeeName="
+ employeeName + ", employeeAddress="
                                + employeeAddress + ", department=" + department + ",
designation=" + designation + "]";
        }

}
```

Main.java

```java
package com.ravi.single_level_inheritance;

public class Main {

        public static void main(String[] args)
        {
                PermanentEmployee p = new PermanentEmployee(101, "Scott", "S R nagar",
"Computer", "Programmer");
```

```
                System.out.println(p);
        }

}
------------------------------------------------------------
Program on Multilevel Inheritance :
-----------------------------------
package com.ravi.multilevel_ex;

class Student {
        protected int studentId;
        protected String studentName;
        protected String studentAddress;

        public Student(int studentId, String studentName, String studentAddress) {
                super();
                this.studentId = studentId;
                this.studentName = studentName;
                this.studentAddress = studentAddress;
        }
}

class Science extends Student {
        protected int physics;
        protected int chemistry;

        public Science(int studentId, String studentName, String studentAddress, int physics, int
chemistry) {
                super(studentId, studentName, studentAddress);
                this.physics = physics;
                this.chemistry = chemistry;
        }

}

class PCM extends Science
{
        protected int math;

        public PCM(int studentId, String studentName, String studentAddress, int physics, int
chemistry, int math) {
                super(studentId, studentName, studentAddress, physics, chemistry);
                this.math = math;
```

```java
        }

        public void calculateMarks()
        {
                int total = this.physics + this.chemistry + this.math;
                System.out.println("Total Marks is :"+total);
        }



        @Override
        public String toString()
        {
                return "PCM [studentId=" + studentId + ", studentName=" + studentName + ",
studentAddress=" + studentAddress
                                + ", physics=" + physics + ", chemistry=" + chemistry + ", math=" +
math + "]";
        }

}

public class MultilevelDemo {

        public static void main(String[] args)
        {
          PCM p = new PCM(1,"Raj","Koti",67,90,78);
          p.calculateMarks();
          System.out.println(p);
        }
}
```

Program on Hierarchical Inheritance :

HierarchicalDemo.java

```java
package com.ravi.hierarchical_demo;

class Employee
{
        protected double salary;

        public Employee(double salary)
        {
```

```java
                super();
                this.salary = salary;
        }
}
class Developer extends Employee
{
        public Developer(double salary)
        {
                super(salary);
        }

        @Override
        public String toString() {
                return "Developer [salary=" + salary + "]";
        }
}
class Designer extends Employee
{
        public Designer(double salary)
        {
                super(salary);
        }

        @Override
        public String toString() {
                return "Designer [salary=" + salary + "]";
        }
}

public class HierarchicalDemo
{
        public static void main(String[] args)
        {
                Developer d = new Developer(55000);
                System.out.println(d);

                Designer d1 = new Designer(30000);
                System.out.println(d1);


        }

}
```

Assignment :

Program on Hybrid Inheritance :

Vehicle

  Car

Ford Maruti
==
**Why java does not support multiple Inheritance ?

Multiple Inheritance is a situation where a sub class wants to inherit the properties two or more than two super classes.

In every constructor we have super() or this(). When compiler will add super() to the first line of the constructor then we have an ambiguity issue that super() will call which super class constructor as shown in the diagram [23-OCT-24]

It is also known as Diamond Problem in java so the final conclusion is we can't achieve multiple inheritance using classes but same we can achieve by using interface [interface does not contain any constructor]

==
Access modifiers in java :

In order to define the accessibility level of the class as well as member of the class we have 4 access modifiers :

1) private (Within the same class)
2) default (Within the same package)
3) protected (Within the same package Or even from another
          package by using Inheritance)
4) public (No Restriction)

private :

It is the most restrictive access modifier because the member declared as private can't be accessible from outside of the class.
In Java we can't declare an outer class as a private or protected. Generally we should declare the data member(variables) as private.

In java outer class can be declared as public, abstract, final, sealed and non-sealed only.

default :-

It is an access modifier which is less restrictive than private. It is such kind of access modifier whose physical existance is not avaialble that means when we don't specify any kind of access modifier before the class name, variable name or method name then by default it would be default.

As far as its accessibility is concerned, default members are accessible within the same folder(package) only. It is also known as private-package modifier.

protected :

It is an access modifier which is less restrictive than default because the member declared as protected can be accessible from the outside of the package (folder) too but by using inheritance concept.

Program on protected access modifier :

----

2 files :

Test.java [Available in com.ravi.m1 package]

```
package com.ravi.m1;

public class Test
{
   protected int x = 500;  //default AM
}
```

ELC.java [Available in com.ravi.m2 package]

```
package com.ravi.m2;

import com.ravi.m1.Test;

public class ELC extends Test
{
        public static void main(String[] args)
        {
                ELC e = new ELC();
                System.out.println(e.x);
```

```
        }

}
```

Note : Both the classes are available in two different packages.
----
public :

It is an access modifier which does not contain any kind of restriction that is the reason the member declared as public can be accessible from everywhere without any restriction.

According to Object Oriented rule we should declare the classes and methods as public where as variables must be declared as private or protected according to the requirement.

Note : If a method is used for internal purpose only (like validation)
then we can declare that method as private method.It is called
Helper method.

HOW MANY WAYS WE CAN INITIALIZE THE OBJECT PROPERTIES ?

The following are the ways to initialize the object properties
----
```
public class Test
{
    int x,y;
}
```

1) At the time of declaration :

   Example :

```
 public class Test
  {
    int x = 10;
    int y = 20;
  }
```

```
  Test t1 = new Test();   [x = 10  y = 20]
  Test t2 = new Test();   [x = 10  y = 20]
```

Here the drawback is all objects will be initialized with same value.
----

2) By using Object Reference :

```
public class Test
{
   int x,y;
}


Test t1 = new Test();   t1.x=10;   t1.y=20;
Test t2 = new Test();   t2.x=30;   t2.y=40;
```

Here we are getting different values with respect to object but here the program becomes more complex.
----
3) By using methods :

A) First Approach (Method without Parameter)

```
public class Test
{
   int x,y;

   public void setData()
   {
      x = 100;  y = 200;
   }
}


Test t1 = new Test();  t1.setData();  [x = 100   y = 200]
Test t2 = new Test();  t2.setData();  [x = 100   y = 200]
```

Here also, all the objects will be iniatilized with same value.

B) Second Approach (Method with Parameter)

```
public class Test
{
 int x,y;

 public void setData(int x, int y)
 {
   this.x = x;
      this.y = y;
```

```
     }
   }

   Test t1 = new Test();  t1.setData(12,78);  [x = 12  y = 78]
   Test t2 = new Test();  t2.setData(15,29);  [x = 15  y = 29]
```

   Here the Drawback is initialization and re-initialization both are done in two different lines so Constructor introduced.

----

4) By using Constructor

  A) First Approach (No Argument Constructor)

```
  public class Test
  {
     int x,y;

     public Test()  //All the objects will be initialized with
     {                                    same value
       x = 0;  y = 0;
     }

  }

   Test t1 = new Test();    [x = 0   y = 0]
   Test t2 = new Test();    [x = 0   y = 0]
```

  B) Second Approach (Parameterized Constructor)

```
   public class Test
   {
    int x,y;

     public Test(int x, int y)
     {
       this.x = x;
         this.y = y;
     }
   }

   Test t1 = new Test(12,78);    [x = 12   y = 78]
   Test t2 = new Test(15,29);    [x = 15   y = 29]
```

This is the best way to initialize our instance variable because variable initialization and variable re-initialization both will be done in the same line as well as all the objects will be initialized with different values.

C) Third Approach (Copy Constructor)

```java
public class Manager
{
   private int managerId;
      private String managerName;

      public Manager(Employee emp)
      {
       this.managerId = emp.getEmployeeId();
       this.managerName = emp.getEmployeeName();
   }
 }
```

Here with the help of Object reference (Employee class) we are initializing the properties of Manager class. (Copy Constructor)

d) By using instance block (Instance Initializer)

```java
 public class Test
 {
   int x,y;

   public Test()
    {
       System.out.println(x); //100
       System.out.println(y); //200
    }

   //Instance block
    {
       x = 100;
       y = 200;
    }
  }
```

----
5) By using super keyword :

```
  class Super
  {
    int x,y;

    public Super(int x , int y)
    {
      this.x = x;
          this.y = y;
    }
  }
  class Sub extends Super
  {
    Sub()
    {
      super(100,200); //Initializing the properties of super class
    }
  }

  new Sub();
```

25-10-2024

JVM Architecture with class loader sub system :

The entire JVM Architecture is divided into 3 sections :

1) Class Loader sub system
2) Runtime Data areas (Memory Areas)
3) Execution Engine

Class Loader Sub System :

The main purpose of Class Loader sub system to load the required .class file into JVM Memory
from different memory loactions.

In order to load the .class file into JVM Memory, It uses an algorithm called "Delegation
Hierarchy Algoroithm".

Internally, Class Loader sub system performs the following Task

1) LOADING
2) LINKING
3) INITIALIZATION

LOADING :

In order to load the required .class file, JVM makes a request to class loader sub system. The class loader sub system follows delegation hierarchy algorithm to load the required .class files from different areas.

To load the required .class file we have 3 different kinds of class loaders.

1) Bootstrap/Primordial class loader

2) Extension/Platform class loader

3) Application/System class loader

Bootstrap/Primordial class Loader :-

It is responsible for loading all the predefined .class files that means all API level predefined classes are loaded by Bootstrap class loader.

It has the highest priority becuase Bootstrap class loader is the super class for Platform class loader.

It loads the classes from the following path
C -> Program files -> Java -> JDK -> lib -> jrt-fs.jar

Platform/Extension class loader :

It is responsible to load the required .class file which is given by some 3rd party in the form of jar file.

It is the sub class of Bootstrap class loader and super class of Application class loader so it has more priority than Application class loader.

It loads the required .class file from the following path.
C -> Program files -> Java -> JDK -> lib -> ext -> ThirdParty.jar

command to create the jar file :

jar cf FileName.jar FileName.class   [*.class]

[If we want to compile more than one java file at a time then the command is : javac *.java]

Application/System class loader :

It is responsible to load all userdefined .class file into JVM memory.

It has the lowest priority because it is the sub class Platform class loader.

It loads the .class file from class path level or environment
variable.

Note :
1) If all the class loaders are failed to load the required
.class file then we will get an exception i.e java.lang.ClassNotFoundException.

2) java.lang.Object is the first class to be loaded into JVM Memory.

What is Method Chaning in java ?

It is a technique through we call multiple methods in a single
statement.

In this method chaining, always for calling next method we depend upon last method return
type.

The final return type of the method depends upon last method call as shown in the program.

package com.ravi.method_chaining;

public class MethodChainingDemo1
{
        public static void main(String[] args)
        {
                String str = "india";
                char ch = str.concat(" is great").toUpperCase().charAt(7);
        System.out.println(ch);
        }

}

```java
package com.ravi.method_chaining;

public class MethodChainingDemo2
{
        public static void main(String[] args)
        {
                String str = "india";
                int length = str.concat(" is great").length();
                System.out.println(length);
        }

}
```

Role of java.lang.Class class in class loading :

There is a predefined class called Class available in java.lang pacakge.

In JVM memory whenever we load a class then it is loaded in special memory called Method
Area and retutn type is java.lang.Class.

   java.lang.Class cls = AnyClass.class

 java.lang.Class class contains a predefined non static method
 called getName() through which we can get the fully qualified
 name [Package Name + class Name]

 public String getName() : Provide fully qualified name of the class.

WAP that describes java.lang.Class will hold any class .class file into JVM Memory.

MethodAreaDemo.java

```java
package com.ravi.method_chaining;

class Employee{}

class Student{}

class Sample{}

public class MethodAreaDemo {

        public static void main(String[] args)
```

```
        {
                Class cls =   Employee.class;
                System.out.println(cls.getName());

                cls = Student.class;
                System.out.println(cls.getName());

                cls = Sample.class;
                System.out.println(cls.getName());

        }

}
```

WAP that describes Application class loader is responsible to
load the user defined .class file

java.lang.Class class has provided a predefined non static method called getClassLoader(), the
return type of this method
is ClassLoader class.[Factory Method]

This method will provide the name of the class loader which is responsible to load the .class file
into JVM Memory.

public ClassLoader getClassLoader()

```
package com.ravi.class_loader_description;

public class Test
{
        public static void main(String[] args)
        {
                System.out.println("Test.class file will be loaded by
:"+Test.class.getClassLoader());
        }

}
```

Note : Test.class statement, return type is java.lang.Class so further we can call any method of
java.lang.Class class.

WAP that describes Platform class Loader is the super class for
Application class loader

getClassLoader() method return type is ClassLoader so further we can call any method of ClassLoader class, ClassLoader class
has provided a method called getParent() whose return type is again ClassLoader only

```
    public ClassLoader getParent();
```

package com.ravi.class_loader_description;

```java
public class Test
{
        public static void main(String[] args)
        {
                System.out.println("Super class of Application class loader is
:"+Test.class.getClassLoader().getParent());
        }

}
```

Output : PlatformClassLoader

----

package com.ravi.class_loader_description;

```java
public class Test
{
        public static void main(String[] args)
        {
                ClassLoader parent = Test.class.getClassLoader().getParent().getParent();
                System.out.println(parent);
        }

}
```

Note :- Here we will get the output as null because it is built in class loader for JVM which is used for internal purpose (loading only predefined .class file) so implementation is not provided hence we are getting null.

----

verify :-

It ensures the correctness of the .class files, If any suspicious activity is there in the .class file then It will stop the execution immediately by throwing a runtime error i.e java.lang.VerifyError.

There is something called ByteCodeVerifier(Component of JVM), responsible to verify the loaded .class file i.e byte code. Due to this verify module JAVA is highly secure language.

java.lang.VerifyError is the sub class of java.lang.linkageError

----
prepare :

[Static variable memory allocation + static variable initialization with default value]

It will allocate the memory for all the static data members, here all the static data member will get the default values so if we have static int x = 100; then for variable x memory will be allocated (4 bytes) and now it will initialize with default value i.e 0, even the variable is final.

static Test t = new Test();

Here, t is a static reference variable so for t variable (reference variable) memory will be allocated as per JVM implementation i.e for 32 bit JVM (4 bytes of Memory) and for 64 bit (8 bytes of memory) and initialized with null.

Resolve :

All the symbolic references will be converted into direct references OR actual references.

javap -verbose FileName.class

Note :- By using above command we can read the internal details of .class file.
==
28-10-2024

Initialization :

Here class initialization will take place. All the static data member will get their actual value and we can also use static block for static data member initialization.

Here, In this class initialization phase static variable and static block is having same priority so it will executed according to the order.(Top to bottom)

Static Block in java :

It is a special block in java which is automatically executed at the time of loading the .class file.

Example :

static
{

}

Static blocks are executed only once because in java we can load the .class files only once.

If we have more than one static block in a class then it will be executed according to the order [Top to bottom]

The main purpose of static block to initialize the static data member of the class so it is also known as static initializer.

In java, a class is not loaded automatically, it is loaded based on the user request so static block will not be executed everytime, It depends upon whether class is loaded or not.

static blocks are executed before the main or any static method.

When we try to load sub class first of all super class will be loaded.

A static blank final field must be initialized inside the static block only.

static final int A;   //static blank final field

static
{
    A = 100;
}

A static blank final field also have default value.

We can't write any kind of return statement inside static block.

If we don't declare static variable before static block body execution then we can perform write operation(Initialization is possible due to prepare phase) but read operation is not possible directly otherwise we will get an error Illegal forward reference, It is possible with class name bacause now compiler knows that it is coming from class area OR Method area.

//static block
class Foo
{

```java
        Foo()
        {
                System.out.println("No Argument constructor..");
        }


        {
                System.out.println("Instance block..");
        }


        static
        {
                System.out.println("Static block...");
        }

}
public class StaticBlockDemo
{
        public static void main(String [] args)
        {

    System.out.println("Main Method Executed ");
        }

}
```

Here Foo.class file is not loaded into JVM Memory so static block of Foo class will not be executed.

```java
class Test
{
        static int x;

        static
        {
                x = 100;
                System.out.println("x value is :"+x);
        }

        static
        {
                x = 200;
                System.out.println("x value is :"+x);
        }
```

```java
        static
        {
                x = 300;
                System.out.println("x value is :"+x);
        }


}
public class StaticBlockDemo1
{
        public static void main(String[] args)
        {
                System.out.println("Main Method");
                System.out.println(Test.x);
        }
}
```

Note : If a class contains more than 1 static block then it will be executed from top to bottom.

```java
class Foo
{
        static int x;

    static
        {
                System.out.println("x value is :"+x);
        }
}

public class StaticBlockDemo2
{
        public static void main(String[] args)
        {
                 new Foo();
        }
}
```
Note : static variables are also having default value.

```java
class Demo
{

        final static int a ;            //Blank static final field
```

```java
        static
        {
                a = 100;
                System.out.println(a);
        }
}
public class  StaticBlockDemo3
{
        public static void main(String[] args)
                {
                System.out.println("a value is :"+Demo.a);
            }
}
```

Note : A blank static final field must be initialized through static block only.

```java
class  A
{
        static
        {
                System.out.println("A");
        }

        {
                System.out.println("B");
        }

        A()
        {
                System.out.println("C");
        }
}
class B extends A
{
        static
        {
                System.out.println("D");
        }


        {
                System.out.println("E");
```

```
        }

        B()
        {
                System.out.println("F");
        }

}
public class StaticBlockDemo4
{
        public static void main(String[] args)
        {
                new B();
        }
}
```

Note : Always Parant class will be loaded first then only Child class will be loaded.

//illegal forward reference

```
class Demo
{
    static
        {
                i = 100;
        }

    static int i;
}

public class StaticBlockDemo5
{
    public static void main(String[] args)
        {
      System.out.println(Demo.i);
        }
}
```

Note : For static variable i, already memory is allocated in the prepare phase so we can initialize
(can perform write operation)
in the static block without pre-declaration.

```java
class Demo
{
    static
        {
                i = 100;
                //System.out.println(i); //Invalid
                System.out.println(Demo.i);
        }

    static int i;
}

public class StaticBlockDemo6
{

    public static void main(String[] args)
        {
                System.out.println(Demo.i);
        }
}
```

Note : Without declaring the static variable if we try to access(read Operation) static variable value in the static block directly then we will get compilation error, we can access with the help of class name (Class Area)

```java
class StaticBlockDemo7
{
        static
        {
    System.out.println("Static Block");
          return;
        }

        public static void main(String[] args)
        {
                System.out.println("Main Method");
        }
}
```

Note : We can't write return statement in static and non static block
----

```java
public class StaticBlockDemo8
{
```

```java
        final static int x; //Blank static final field

    static
        {
                m1();
                x = 15;
        }

    public static void m1()
        {
                System.out.println("Default value of x is :"+x);
        }

        public static void main(String[] args)
        {
                System.out.println("After initialization :"+StaticBlockDemo8.x);
        }
}
```

A blank static final field also has default value.

```java
class Test
{
  public static final Test t1 = new Test();   //t1 = null

   static
        {
           System.out.println("static block");
        }

        {
           System.out.println("Non static block");
        }

        Test()
        {
                System.out.println("No Argument Constructor");
        }



}
```

```java
public class StaticBlockDemo9
{
        public static void main(String[] args)
        {
                new Test();
        }
}
```

Note : First non static block, constructor then only static block will be executed.

29-10-2024

Variable Memory Allocation and Initialization :
----
1) static field OR Class variable :

Memory allocation done at prepare phase of class loading and initialized with default value even variable is final.

It will be initialized with Original value (If provided by user at the time of declaration) at class initialization phase.

When JVM will shutdown then during the shutdown phase class will be un-loaded from JVM memory so static data members are destroyed. They have long life.

2) Non static field OR Instance variable

Memory allocation done at the time of object creation using new keyword (Instantiation) and initialized as a part of Constructor with default values even the variable is final. [Object class-> at the time of declaration -> instance block -> constructor]

When object is eligible for GC then object is destroyed and all the non static data memebers are also destroyed with corresponding object. It has less life in comparison to static data members becuase they belongs to object.

3) Local Variable

Memory allocation done at stack area (Stack Frame) and developer is responsible to initialize the variable before use. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

4) Parameter variable

Memory allocation done at stack area (Stack Frame) and end user is responsible to pass the value at runtime. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

Note : We can done validation only one parameter variables.
==
Can we write a Java Program without main method ?

```
class WithoutMain
{
   static
   {
      System.out.println("Hello User!!");
      System.exit(0);
   }
}
```

It was possible to write a java program without main method till JDK 1.6V. From JDK 1.7v onwards, at the time of loading the .class file JVM will verify the presence of main method in the .class file. If main method is not available then it will generate a runtime error that "main method not found in so so class".

How many ways we can load the .class file into JVM memory :

There are so many ways to load the .class file into JVM memory but the following are the common examples :

1) By using java command

```
   public class Test
   {
   }

   javac Test.java
   java Test
```

   Here we are making a request to class loader sub system to load Test.class file into JVM memory

2) By using Constructor (new keyword at the time of creating object).

3) By accessing static data member of the class.

4) By using inheritance

5) By using Reflection API

//Program that describes we can load a .class file by using new keyword (Object creation) OR by accessing static data member of the class.

```java
class Demo
{
        static int x = 10;
        static
        {
                System.out.println("Static Block of Demo class Executed!!! :"+x);
        }
}
public class ClassLoading
{
        public static void main(String[] args)
        {
                System.out.println("Main Method");
                //new Demo();
            System.out.println(Demo.x);
        }
}
```

//Program that describes whenever we try to load sub class, first of all super class will be loaded. [before parent, child can't exist]

```java
class Alpha
{
        static
        {
                System.out.println("Static Block of super class Alpha!!");
        }
}
class Beta extends Alpha
{
        static
        {
                System.out.println("Static Block of Sub class Beta!!");
        }
```

```
}
class InheritanceLoading
{
        public static void main(String[] args)
        {
                new Beta();
        }
}
```

Loading the .class file by using Reflection API :

----

java.lang.Class class has provided a predefined static and factory method called forName(String className), It is mainly used to load the given .class file at runtime, The return type of this method is java.lang.Class

public static java.lang.Class forName(String className)

Note : This method throws a checked execption i.e ClassNotFoundException

```
package com.ravi.dynamic_loading;

class Test
{
        static
        {
                System.out.println("Class Loaded....");
        }

}
public class DynamicLoading
{
        public static void main(String[] args) throws Exception
        {
                Class.forName("com.ravi.dynamic_loading.Test");
        }
}
```

Note : From the above program it is clear that Class.forName(String className) is used to load the given .class file dynamically at runtime.

Eclipse IDE always accept Fully Qualified Name (FQN) [Package Name + Class Name]

IQ :

** What is the difference between java.lang.ClassNotFoundException and java.lang.NoClassDefFoundError

java.lang.ClassNotFoundException :-

It occurs when we try to load the required .class file at RUNTIME by using Class.forName(String className) statement or loadClass() static of ClassLoader class and if the required .class file is not available at runtime then we will get an exception i.e java.lang.ClassNotFoundException

Note :- It does not have any concern at compilation time, at run time, JVM will simply verify whether the required .class file is available or not available.

```
class Foo
{
        static
        {
                System.out.println("static block of Foo class");
        }
}
public class ClassNotFoundExceptionDemo
{
        public static void main(String[] args) throws Exception
        {
                Class.forName("Player");
        }
}
```

Note : Here Player.class file is not available at runtime so, we will get java.lang.ClassNotFoundException.

java.lang.NoClassDefFoundError :

It occurs when the class was present at the time of COMPILATION but at runtime the required .class file is not available(manualy deleted by user) Or it is not available in the current directory (Misplaced) then we will get a runtime error i.e java.lang.NoClassDefFoundError.

```
class Welcome
{
```

```java
        public void greet()
        {
                System.out.println("Hello Batch 38!");
        }
}
public class NoClassDefFoundErrorDemo
{
        public static void main(String[] args)
        {
                Welcome w = new Welcome();
                w.greet();
        }
}
```

After compilation either delete Welcome.class or put this .class file in another folder[remove from the current folder]

Why a static method does not allow to access instance variable directly ?

All the static members (static variable, static block, static method, static nested inner class) are loaded/executed at the time of loading the .class file into JVM Memory.

At class loading phase object is not created because object is created in the 2nd phase i.e Runtime data area so at the TIME OF EXECUTION OF STATIC METHOD AT CLASS LOADING PAHSE, NON STATIC VARIABLE WILL NOT BE AVAILABLE henec we can't access non static variable from static context[static block, static method and static nested inner class]

===

```java
public class Test
{
        int x = 900;

        public static void main(String[] args)
        {
                System.out.println(" x value is "+x); //error
        }
}
```

```java
class Sample
{
        private int x;

        public Sample(int x)
```

```
		{
			this.x = x;
		}
		public static void access()
		{
			System.out.println(x); //error
		}

}
public class Test
{
	public static void main(String[] args)
	{
		Sample s = new Sample(10);
		Sample.access();
	}
}
```
==
Runtime Data Areas :

It is also known as Memory Area.

Once a class is loaded then based on variable type method type it is divided into different
memory areas which are as follows :

1) Method Area
2) HEAP Area
3) Stack Area
4) PC Register
5) Native Method Stack

Method Area :

Whenever a class is loaded then the class is dumpped inside method area and returns
java.lang.Class class.

It provides all the information regarding the class like name of the class, name of the package,
static and non static fields available in the class, methods available in the class and so on.

We have only one method area per JVM that means for a single JVM we have only one Method
area.

This Method Area OR Class Area is sharable by all the objects.

Program to Show From Method Area we can get complete information of the class. (Reflection API)

2 files :

Test.java

```java
package com.ravi.method_area;

import java.util.Scanner;

public class Test
{
  int x = 100;
  static int y = 200;

  Integer a = 500;
  static Scanner b = new Scanner(System.in);

  public void accept() {}

  public void input() {}

  public void show() {}

  public void display() {}

  public void m1() {}

  public void m2() {}

}
```

ClassDescription.java

```java
package com.ravi.method_area;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ClassDescription {
```

```java
public static void main(String[] args) throws Exception
{
        Class cls = Class.forName(args[0]);

        System.out.println("Class Name :"+cls.getName());
        System.out.println("Package Name :"+cls.getPackageName());

        System.out.println("Available Methods :");
          int count = 0;

          Method[] methods = cls.getDeclaredMethods();

          for(Method method : methods)
          {
                  count++;
                  System.out.println(method.getName());
          }

        System.out.println("Total number of methods are :"+count);

        System.out.println("Available Fields :");
        count = 0;
        Field[] fields = cls.getDeclaredFields();

        for(Field field : fields)
        {
                count++;
                System.out.println(field.getName());
        }
        System.out.println("Total number of fields are :"+count);

    }

}
```

Note :- getDeclaredMethods() is a predefined non static method available in java.lang.Class class , the return type of this method is Method array where Method is a predefined class available in java.lang.reflect sub package.

getDeclaredFields() is a predefined non static method available in java.lang.Class class , the return type of this method is Field array where Field is a predefined class available in

java.lang.reflect sub package.

Field and Method both the classes are providing getName() method to get the name of the field and Method.

HEAP AREA :

Whenever we create an object in java then the properties and behavior of the object are strored in a special memory area called HEAP AREA.

We have only one HEAP AREA per JVM.

STACK Area :

All the methods are executed as a part of Stack Area.

Whenever we call a method in java then internally one stack Frame will be created to hold method related information.

Every Stack frame contains 3 parts :
1) Local Variable arrays
2) Frame Data
3) Operand Stack.

We have multiple stack area for a single JVM.

JVM creates a separate stack Memory for each individual thread.

HEAP and STACK Diagram for Beta.java

```
class Alpha
{
        int val;

        static int sval = 200;
        static Beta b = new Beta();

        public Alpha(int val)
        {
    this.val = val;
        }
}
```

```java
public class Beta
{
        public static void main(String[] args)
        {
                Alpha am1 = new Alpha(9);
                Alpha am2 = new Alpha(2);

                Alpha []ar = fill(am1, am2);

                ar[0] = am1; //2000x
        System.out.println(ar[0].val);
        System.out.println(ar[1].val);
            }

        public static Alpha[] fill(Alpha a1, Alpha a2)
        {
                a1.val = 15;

        Alpha fa[] = new Alpha[]{a2, a1};

                return fa;
            }
}
```

PC Register :

It stands for Program counter Register.

In order to hold the current executing instruction of running thread we have separate PC register
for each and every thread.

Native Method Stack :

Native method means, the java methods which are written by using native languages like C and
C++. In order to write native method we need native method library support.

Native method stack will hold the native method information in a separate stack.

Interpreter

In java, JVM contains an interpreter which executes the program line by line. Interpreter is slow

in nature because at the time of execution if we make a mistake at line number 9 then it will throw the execption at line number 9 and after solving the execption again it will start the execution from line number 1 so it is slow in execution that is the reason to boost up the execution java software people has provided JIT compiler.

JIT Compiler :

It stands for just in time compiler. The main purpose of JIT compiler to boost up the execution so the execution of the program will be completed as soon as possible.

JIT compiler holds the repeated instruction like method signature, variables, native method code and make it available to JVM at the time of execution so the overall execution becomes very fast.

HAS-A Relation :

If we use any class as a property to another class then it is called HAS-A relation.

class Engine
{
}

class Car
{
    private Engine engine; //HAS-A relation
}

Association :

Association is a connection between two separate classes that can be built up through their Objects.

The association builds a relationship between the classes and describes how much a class knows about another class.

This relationship can be unidirectional or bi-directional. In Java, the association can have one-to-one, one-to-many, many-to-one and many-to-many relationships.

Example:-
One to One: A person can have only one PAN card
One to many: A Bank can have many Employees
Many to one: Many employees can work in single department
Many to Many: A Bank can have multiple customers and a customer can have multiple bank

accounts.

----

3 files :

Student.java

```java
package com.ravi.association;

public class Student
{
        private int studentId;
        private String studentName;
        private int studentMarks;

        public Student(int studentId, String studentName, int studentMarks) {
                super();
                this.studentId = studentId;
                this.studentName = studentName;
                this.studentMarks = studentMarks;
        }

        public int getStudentId() {
                return studentId;
        }

        public String getStudentName() {
                return studentName;
        }

        public int getStudentMarks() {
                return studentMarks;
        }

        @Override
        public String toString() {
                return "Student [studentId=" + studentId + ", studentName=" + studentName + ",
studentMarks=" + studentMarks
                                        + "]";
        }

}

Trainer.java
```

```java
package com.ravi.association;

import java.util.Scanner;

public class Trainer
{
  public static void viewStudentProfile(Student obj)
  {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Student id :");
        int id = sc.nextInt();

        if(id == obj.getStudentId())
        {
                System.out.println(obj);
        }
        else
        {
                System.err.println("Sorry! Id is not available");
        }
        sc.close();
  }
}
```

AssociationDemo.java

```java
package com.ravi.association;

public class AssociationDemo {

        public static void main(String[] args)
        {
                Student s1 = new Student(101, "Scott", 80);

                Trainer.viewStudentProfile(s1);

        }

}
```

Composition : [Strong Reference]

Composition in Java is a way to design classes such that one class contains an object of another class. It is a way of establishing a "HAS-A" relationship between classes.

Composition represents a strong relationship between the containing class and the contained class.If the containing object (Car object) is destroyed, all the contained objects (Engine object) are also destroyed.

A car has an engine. Composition makes strong relationship between the objects. It means that if we destroy the owner object, its members will be also destroyed with it. For example, if the Car is destroyed the engine will also be destroyed as well.

Program Guidelines :

One object can't exist without another object
We will not create two separate objects.

3 files :

Engine.java

```java
package com.ravi.composition;

public class Engine
{
        private String engineType;
        private int horsePower;

        public Engine(String engineType, int horsePower)
        {
                super();
                this.engineType = engineType;
                this.horsePower = horsePower;
        }

        @Override
        public String toString()
        {
                return "Engine [engineType=" + engineType + ", horsePower=" + horsePower +
"]";
        }
}
```

Car.java
----
package com.ravi.composition;

public class Car
{
        private String carName;
        private int carModel;
        private final Engine engine; // HAS-A Relation [Blank final field]

        public Car(String carName, int carModel)
        {
                super();
                this.carName = carName;
                this.carModel = carModel;
                this.engine = new Engine("Battery", 1400); //Composition
        }

        @Override
        public String toString() {
                return "Car [carName=" + carName + ", carModel=" + carModel + ", engine=" +
engine + "]";
        }

}

CompositionDemo.java
----
package com.ravi.composition;

public class CompositionDemo {

        public static void main(String[] args)
        {
                Car car = new Car("Ford", 2024);
                System.out.println(car);

        }

}

Aggregation :

Aggregation (Weak Referance) :

Aggregation in Java is another form of association between classes that represents a "HAS-A" relationship, but with a weaker bond compared to composition.

In aggregation, one class contains an object of another class, but the contained object can exist independently of the container. If the container object is destroyed, the contained object can still exist.

3 files :

College.java

```java
package com.ravi.aggregation;

public class College
{
        private String collegeName;
        private String collegeLocation;

        public College(String collegeName, String collegeLocation)
        {
                super();
                this.collegeName = collegeName;
                this.collegeLocation = collegeLocation;
        }

        @Override
        public String toString()
        {
                return "College [collegeName=" + collegeName + ", collegeLocation=" +
collegeLocation + "]";
        }
}
```

Student.java

```java
package com.ravi.aggregation;

public class Student
{
```

```java
        private int studentId;
        private String studentName;
        private College college; // HAS-A Relation

        public Student(int studentId, String studentName, College college)
        {
                super();
                this.studentId = studentId;
                this.studentName = studentName;
                this.college = college;
        }

        @Override
        public String toString()
        {
                return "Student [studentId=" + studentId + ", studentName=" + studentName + ",
college=" + college + "]";
        }

}
```

AggregationDemo.java

```java
package com.ravi.aggregation;

public class AggregationDemo {

        public static void main(String[] args)
        {
                College clg = new College("JNTU","Hyderabad");

                Student s1 = new Student(1, "Scott", clg);
                System.out.println(s1);

                Student s2 = new Student(2, "Smith", clg);
                System.out.println(s2);

                Student s3 = new Student(3, "John", clg);
                System.out.println(s3);


        }
```

}

Note :- IS-A relation is tightly coupled relation so if we modify the content of super class, sub class content will also modify but in HAS-A realtion we are accessing the properties of another class so we are not allowed to modify the content, we can access the content or Properties.
==
Description of System.out.println() :
----
public class System
{
   public final static java.io.PrintStream out = null; //HAS-A Relation
}

System.out.println();

Internally System.out.println() creates HAS-A relation because System class contains a predefined class called java.io.PrintStream as shown in the above example.

The following program describes that how System.out.println() works internally :

2 files :

Demo.java
----
package has_a_relation;

public class Demo
{
   public static final String out = "india";

}

Test.java
----
package has_a_relation;

public class Test
{
 public static void main(String[] args)
 {
      System.out.println(Demo.out.length());
 }

}

***Polymorphism :

Poly means "many" and morphism means "forms".

It is a Greek word whose meaning is "same object having different behavior".

In our real life a person or a human being can perform so many task, in the same way in our programming languages a method or a constructor can perform so many task.

Eg:-

void add(int a, int b)

void add(int a, int b, int c)

void add(float a, float b)

void add(int a, float b)

Types of Polymorphism :

Polymorphism can be divided into two types :

1) Static polymorphism OR Compile time polymorphism OR Early binding

2) Dynamic Polymorphism OR Runtime polymorphism OR Late binding

----

1) Static Polymorphism :

The polymorphism which exist at the time of compilation is called Static OR compile time polymorphism.

In static polymorphism, compiler has very good idea that which method is invoked depending upon METHOD PARAMETER.

Here the binding of the method is done at compilation time so, it is known as early binding.

We can achieve static polymorphism by using Method Overloading concept.

Example of static polymorphism : Method Overloading.

2) Dynamic Polymorphism

The polymorphism which exist at runtime is called Dynamic polymorphim Or Runtime
Polymorphism.

*Here compiler does not have any idea about method calling, at runtime JVM will decide which
method will be invoked depending upon CLASS TYPE OBJECT.

Here method binding is done at runtime so, it is also called Late Binding.

We can achieve dynamic polymorphism by using Method Overriding.

Example of Dynamic Polymorphism : Method Overriding

12-11-2024

Method Overloading :

Writing two or more methods in the same class or even in the super and sub class in such a
way that the method name must be same but the argument must be different.

While Overloading a method we can change the return type of the method.

If parameters are same but only method return type is different then it is not an overloaded
method.

Method overloading is possible in the same class as well as super and sub class.

While overloading the method the argument must be different otherwise there will be ambiguity
problem.

Method Overloading allows two methods with same name to differ
in:
1. Number of parameters
2. Data type of parameters
3. Sequence of data type of parameters(int -long and long int)

IQ :

Can we overload the main method/static method ?

Yes, we can overload the main method OR static method but the execution of the program will start from main method which accept String [] array as a parameter.

Note :- The advantage of method overloading is same method name we can reuse for different functionality for refinement of the method.

Note :- In System.out.println() or System.out.print(), print()
        and println() methods are best example for Method Overloading.

Example :

```
public void makePayment(Cash c)
{
}
public void makePayment(UPI c)
{
}
public void makePayment(CreditCard c)
{
}
```
----
2 files :

Addition.java

```
//Program on Constructor Overloading
package com.ravi.constructor_overloading;

public class Addition
{
   public Addition(int x, int y)
   {
          this(10,20,30);
          System.out.println("Sum of two integer is :"+(x+y));
   }

   public Addition(int x, int y, int z)
   {
          this(2.3F, 8.9f);
          System.out.println("Sum of three integer is :"+(x+y+z));
   }

   public Addition(float x, float y)
```

```java
	{
		super();
		System.out.println("Sum of two float is :"+(x+y));
	}

}
```

Main.java

```java
package com.ravi.constructor_overloading;

public class Main {

	public static void main(String []  args)
	{
		new Addition(12,90);
	}

}
```

//Program on Constructor Overloading

2 files :

Addition.java

```java
package com.ravi.constructor_overloading2;

public class Addition
{
  public Addition()
  {
	this(100);
	System.out.println("No Argument Constructor");
  }

  public Addition(int x)
  {
	 this(1000,2000);
	 System.out.println("One Argument Constructor :"+x);

  }
  public Addition(int x, int y)
```

```java
    {
            super();
            System.out.println("Two Argument Constructor :"+x+":"+y);
    }

    {
                System.out.println("Instance Block");
    }
}
```

Test.java

```java
package com.ravi.constructor_overloading2;

public class Test {

        public static void main(String[] args)
        {
                new Addition();

        }
}
```

Note : Non static block will be added to the constructor which contains super() [Here in only one constructor]

The following program explains, we can change the return type of the method while overloading a method

2 files :

Sum.java

```java
package com.ravi.method_overload;

public class Sum
{
 public int add(int x, int y)
 {
        return x + y;
 }
```

```java
    public String add(String x, String y)
    {
            return x + y;
    }

    public double add(double x, double y)
    {
            return x + y;
    }



}
```

Main.java

```java
package com.ravi.method_overload;

public class Main
{
        public static void main(String[] args)
        {
                Sum s = new Sum();
                double add1 = s.add(2.6, 7.8);
                System.out.println("Addition of two double is :"+add1);

                String add2 = s.add("Data", "base");
                System.out.println("Concatenation of two String :"+add2);

                int add3 = s.add(12, 24);
                System.out.println("Addition of two integer :"+add3);
        }
}
```
==
Var-Args :

It was introduced from JDK 1.5 onwards.

It stands for variable argument. It is an array variable which can hold 0 to n number of parameters of same type or different type by using Object class.

It is represented by exactly 3 dots (...) so it can accept any number of argument (0 to nth) that means now we need not to define method body again and again, if there is change in method parameter value.

var-args must be only one and last argument.

We can use var-args as a method parameter only.

===
2 files :

Test.java
----
package com.ravi.var_args;

public class Test
{
  public void accept(int ...x)
  {
        System.out.println("Var args executed");
  }
}

Main.java
----
package com.ravi.var_args;

public class Main
{
   public static void main(String ...args)
   {
        Test t1 = new Test();
            t1.accept();
            t1.accept(12);
            t1.accept(12,89);
            t1.accept(12,78,56);
            t1.accept(10,20,30,40);
   }

}

Var args can accept 0 to n number of parameters.

//Program to add sum of parameters passes inside a method using var args

2 files :

Test.java

----

package com.ravi.var_args1;

public class Test
{
    public void sumOfParameters(int... values)
    {
        int sum = 0;

        for(int value : values)
        {
            sum = sum + value;
        }

        System.out.println("Sum of parameters :"+sum);
    }
}

Main.java

----

package com.ravi.var_args1;

public class Main
{
        public static void main(String[] args)
        {
                Test t1 = new Test();
                t1.sumOfParameters(12,24,24);
                t1.sumOfParameters(100,200,300);
                t1.sumOfParameters(1,1,1,1,1,1);

        }

}

==
Program that describes var args must be onlt one and last argument.

Test.java
----
package com.ravi.var_args2;

public class Test
{
        // All commented codes are invalid

        /*
         * public void accept(float ...x, int ...y) { }
         *
         * public void accept(int ...x, int y) { }
         *
         * public void accept(int...x, int ...y) {}
         */

        public void accept(int x, int... y) // Valid
        {
                System.out.println("x value is :" + x);

                for (int z : y)
                {
                        System.out.println(z);
                }
        }
}

Main.java
----
package com.ravi.var_args2;

public class Main {

        public static void main(String[] args)
        {
                new Test().accept(12, 10,20,30,40);
```

```
        }
}
------------------------------------------------------------
2 files :
---------
Test.java
----------
package com.ravi.var_args3;

public class Test
{
   public void acceptHetro(Object ...obj)
   {
           for(Object o : obj)
           {
                   System.out.println(o);
           }
   }
}


Main.java
-----------

package com.ravi.var_args3;

public class Main {

        public static void main(String[] args)
        {
          new Test().acceptHetro(true,45.90,12,'A', new String("Ravi"));



        }

}
================================================================

Wrapper classes in java :
-------------------------
We have 8 primitive data types in java i.e byte, short, int, long and so on.

Except these 8 primitive data types, everything in java is an object.
```

If we remove these 8 primitive data types then only java can become pure object oriented language.

On these primitive data types, we can't assign null or we can't invoke a method.

These primitive data types are unable to move in the network, only objects are moving in the network.

We can't perform serialization and object cloning on primitive data types.

To avoid the above said problems, From JDK 1.5v, java software people has provided the following two concepts :

a) Autoboxing
b) Unboxing

Autoboxing

When we convert the primitive data types into corresponding wrapper object then it is called Autoboxing as shown below.

```
Primitive type          Wrapper Object
----------------        ------------------
byte            -       Byte
short           -       Short
int             -       Integer
long            -       Long
float           -       Float
double          -       Double
char            -       Chracter
boolean             -       Boolean
```

Note : ALL THE WRAPPER CLASSES ARE IMMUTABLE(UN-CHANGED) AS WELL AS equals(Object obj) and hashCode() method is overridden
in all the Wrapper classes.

```java
//Integer.valueOf(int);
public class AutoBoxing1
{
        public static void main(String[] args)
        {
                int a = 12;
```

```java
            Integer x = Integer.valueOf(a); //Upto 1.4 version
            System.out.println(x);


        int y = 15;
            Integer i = y;   //From 1.5 onwards compiler takes care
            System.out.println(i);
    }
}
-----------------------------------------------------------
public class AutoBoxing2
    {
            public static void main(String args[])
                    {
                        byte b = 12;
        Byte b1 = Byte.valueOf(b);
        System.out.println("Byte Object :"+b1);

        short s = 17;
        Short s1 = Short.valueOf(s);
        System.out.println("Short Object :"+s1);

        int i = 90;
        Integer i1 = Integer.valueOf(i);
        System.out.println("Integer Object :"+i1);

            long g = 12;
            Long h =  Long.valueOf(g);
            System.out.println("Long Object :"+h);

            float f1 = 2.4f;
            Float f2 = Float.valueOf(f1);
            System.out.println("Float Object :"+f2);

            double k = 90.90;
            Double l = Double.valueOf(k);
            System.out.println("Double Object :"+l);

            char ch = 'A';
            Character ch1 = Character.valueOf(ch);
            System.out.println("Character Object :"+ch1);

            boolean x = true;
```

```
                Boolean x1 = Boolean.valueOf(x);
                System.out.println("Boolean Object :"+x1);



      }
}
```
In the above program we have used 1.4 approach so we are converting primitive to wrapper object manually.

Overloaded valueOf() method :

----

1) public static Integer valueOf(int x) : It will convert the given int value into Integer Object.

2) public static Integer valueOf(String str) : It will convert
   the given String into Integer Object.
   [valueOf() method will convert the String into Wrapper object where as parseInt() method will convet the String into primitive type]

3) public static Integer valueOf(String str, int radix/base) :
   It will convert the given String number into Integer object
   by using the radix or base.

 Note :- We can pass base OR radix upto 36
i.e A to Z (26) + 0 to 9 (10) -> [26 + 10 = 36], It can be
calculated by using Character.MAX_RADIX.
Output will be generated on the basis of radix

```
//Integer.valueOf(String str)
//Integer.valueOf(String str, int radix/base)
public class AutoBoxing3
{
        public static void main(String[] args)
        {
                        Integer a = Integer.valueOf(15);

                        Integer b = Integer.valueOf("25");

        Integer c = Integer.valueOf("111",36); //Here Base we can take upto 36

                        System.out.println(a);
                        System.out.println(b);
                        System.out.println(c);
```

```
        }
}
-------------------------------------------------------------
public class AutoBoxing4
{
        public static void main(String[] args)
        {
                Integer i1 = new Integer(100);
                Integer i2 = new Integer(100);
                System.out.println(i1==i2);

                Integer a1 = Integer.valueOf(15);
                Integer a2 = Integer.valueOf(15);
                System.out.println(a1==a2);
        }
}


-------------------------------------------------------------
```

Converting primtive to String type :

Integer class has provided a static method toString() which will convert the int value into String type.

```
//Converting integer value to String
public class AutoBoxing5
{
        public static void main(String[] args)
        {
                int x = 12;
                String str = Integer.toString(x);
                System.out.println(str+2);
        }
}
-------------------------------------------------------------
```

15-10-2024

Unboxing :

Converting wrapper object to corresponding primitive type is called Unboxing.

Wrapper   Primitive
Object   type

```
   ----------            ----------
   Byte          -       byte

   Short         -       short

   Integer       -       int

   Long          -       long

   Float         -       float

   Double        -       double

   Chracter      -       char

   Boolean              -        boolean
```

We have total 8 Wrapper classes.

Among all these 8, 6 Wrapper classes are the sub class of Number class which represent numbers (either decimal OR non decimal)
so all the following six wrapper classes (Which are sub class of Number class) are providing the following common methods.

1) public byte byteValue()

2) public short shortValue()

3) public int intValue()

4) public long longValue()

5) public float floatValue()

6) public double doubleValue()

```java
//Converting Wrapper object into primitive
public class AutoUnboxing1
{
   public static void main(String args[])
         {
       Integer obj = 15;   //Upto 1.4
                     int x = obj.intValue();
```

```java
                        System.out.println(x);
            }
}
--------------------------------------------------------------
public class AutoUnboxing2
{
        public static void main(String[] args)
        {
                        Integer x = 25;
                        int y = x;        //JDK 1.5 onwards
                        System.out.println(y);
        }
}
--------------------------------------------------------------
public class AutoUnboxing3
{
        public static void main(String[] args)
        {
                        Integer i = 15;
                        System.out.println(i.byteValue());
                        System.out.println(i.shortValue());
                        System.out.println(i.intValue());
                        System.out.println(i.longValue());
                        System.out.println(i.floatValue());
                        System.out.println(i.doubleValue());
        }
}
--------------------------------------------------------------
public class AutoUnboxing4
{
        public static void main(String[] args)
        {
                Character c1 = 'A';
                char ch = c1.charValue();
                System.out.println(ch);
        }
}
--------------------------------------------------------------
public class AutoUnboxing5
{
        public static void main(String[] args)
        {
                Boolean b1 = true;
```

```
            boolean b = b1.booleanValue();
            System.out.println(b);
        }
}
```

Unlike primitive types we can't convert one wrapper type object to another wrapper object.

Example :

Long l = 12; //Invalid

Float f = 90; //Invalid

Double d = 123; //Invalid

package com.ravi.basic;

public class Conversion
{
        public static void main(String[] args)
        {
          long l = 12; //Implicit OR Widening
          byte b = (byte) 12L; //Explicit OR Narrowing

          Long a = 12L;
          Double d = 90D;
          Double d1 = 90.78;
          Float f = 12F;
        }

}

Ambiguity issue while overloading a method :

When we overload a method then compiler is selecting appropriate method among the available methods based on the following types.

1. Different number of parameters
2. Different data type of parameters
3. Different sequence(order) of data type of parameters

In order to solve the ambiguity issue while overloading a method compiler has provided the

following rules :

1) Most Specific Type :

Compiler alwyas provide more priority to most specific data type or class type.

double > float [Here float is the most specific type]
float > long
long > int
int > char
int > short
short > byte

2) WAV [Widening -> Autoboxing -> Var Args]

   Compiler gives the priority to select appropriate method by using
   the following sequence :
   Widening ---> Autoboxing ----> Var args

3) Nearest Data type or Nearest class (sub class)

   While selecting the appropriate method in ambiguity issue compiler provides priority to nearest
data type or nearest class i.e sub class
----
class Test
{
        public void accept(double d)
        {
                System.out.println("double");
        }
        public void accept(float d)
        {
                System.out.println("float");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(6);

        }

```
}
```

Here float will be executed because float is the nearest type.

```java
class Test
{
        public void accept(int d)
        {
                System.out.println("int");
        }
        public void accept(char d)
        {
                System.out.println("char");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(6);



        }
}
```

Here int will be executed because int is the nearest type.

```java
class Test
{
        public void accept(int ...d)
        {
                System.out.println("int");
        }
        public void accept(char ...d)
        {
                System.out.println("char");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
```

```
        {
                Test t = new Test();
                t.accept();
        }
}
```

Here char will be executed because char is the most specific type
----

```
class Test
{
        public void accept(short ...d)
        {
                System.out.println("short");
        }
        public void accept(char ...d)
        {
                System.out.println("char");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept();
        }
}
```

Here we will get compilation error because there is no relation between char and short based on the specific type rule.

```
class Test
{
        public void accept(short ...d)
        {
                System.out.println("short");
        }
        public void accept(byte ...d)
        {
                System.out.println("byte");
        }
}
public class AmbiguityIssue {
```

```
        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept();
        }
}
```

Here byte will be executed because byte is the specific type.

```
class Test
{
        public void accept(double ...d)
        {
                System.out.println("double");
        }
        public void accept(long ...d)
        {
                System.out.println("long");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept();
        }
}
```
Here long will be executed because long is the most specific type.

```
class Test
{
        public void accept(byte d)
        {
                System.out.println("byte");
        }
        public void accept(short s)
        {
                System.out.println("short");
        }
}
public class AmbiguityIssue {
```

```
        public static void main(String[] args)
        {
                Test t1 = new Test();
                t1.accept(15);
                t1.accept(byte(15));
        }
}
```

Here value 15 is of type int so, we can't assign directly to byte and short, If we want, explicit type casting is reqd.

```
class Test
{
        public void accept(int d)
        {
                System.out.println("int");
        }
        public void accept(long s)
        {
                System.out.println("long");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(9);

        }
}
```

Note : Here int will be executed because int is the nearest type

----

```
class Test
{
        public void accept(int d, long l)
        {
                System.out.println("int-long");
        }
        public void accept(long s, int i)
        {
                System.out.println("long-int");
        }
}
```

```java
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(9,9);
        }
}
```
Here We will get ambiguity issue.

```java
class Test
{
        public void accept(Object s)
        {
                System.out.println("Object");
        }
        public void accept(String s)
        {
                System.out.println("String");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(9);

        }
}
```
Here Object will be executed

```java
class Test
{
        public void accept(Object s)
        {
                System.out.println("Object");
        }
        public void accept(String s)
        {
                System.out.println("String");
        }
}
```

```java
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept("NIT");
        }
}
```
Here String will be executed

```java
class Test
{
        public void accept(Object s)
        {
                System.out.println("Object");
        }
        public void accept(String s)
        {
                System.out.println("String");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(null);

        }
}
```
String will executed because String is the nearest type.

```java
class Test
{
        public void accept(Object s)
        {
                System.out.println("Object");
        }
        public void accept(String s)
        {
                System.out.println("String");
        }
        public void accept(Integer i)
```

```
        {
                System.out.println("Integer");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(null);
    }
}
```
Here We will get compilation error

```
class Alpha
{
}
class Beta extends Alpha
{
}
class Test
{
        public void accept(Alpha s)
        {
                System.out.println("Alpha");
        }
        public void accept(Beta i)
        {
                System.out.println("Beta");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(null);


    }
}
```
Here Beta will be executed.

```java
class Test
{
        public void accept(Number s)
        {
                System.out.println("Number");
        }
        public void accept(Integer i)
        {
                System.out.println("Integer");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(12);


        }
}
```

Here Integer will be executed.

```java
class Test
{
        public void accept(long s)
        {
                System.out.println("Widening");
        }
        public void accept(Integer i)
        {
                System.out.println("Autoboxing");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(12);
    }
}
```
Here widening is having more priority

```
--------------------------------------------------------------
class Test
{
        public void accept(int ...s)
        {
                System.out.println("Var args");
        }
        public void accept(Integer i)
        {
                System.out.println("Autoboxing");
        }
}
public class AmbiguityIssue {

        public static void main(String[] args)
        {
                Test t = new Test();
                t.accept(12);

    }
}
```

Here Autoboxing will be executed.

13-11-2024

***Method Overriding :

Writing two or more non static methods in super and sub class in such a way that method name
along with method parameter (Method Signature) must be same is called Method Overriding.

Method Overriding is not possible without inheritance.

Generally we can't change the return type of the method while overriding a method but from
JDK 1.5v there is a concept called Co-variant (In same direction) through which we can change
the
return type of the method.

Example :

```
class Super
{
        public void m1()
        {
```

```
        }
}
class Sub extends Super
{
        public void m1() //Overridden Method
        {


        }
}
```

Method overriding is mainly used to replacing the implementation
of super class method by sub class method body.

Advantage of Method Overriding :

The advantage of Method Overriding is, each sub class is specifying its own specific behavior.

Upcasting :-

It is possble to assign sub class object to super class reference variable (up) using dynamic
polymorphism. It is known as Upcasting.

Example:-   Animal a = new Lion(); //valid [upcasting]

Downcasting :

By default we can't assign super class object to sub class reference variable.

      Lion l = new Animal(); //Invalid

  Even if we type cast Animal to Lion type then compiler will allow but at runtime JVM will not
convert Animal object (Generic type) into Lion object (Specific type) and it will throw an
exception java.lang.ClassCastException

  Lion l = (Lion) new Animal(); //At runtime we will get
                        java.lang.ClassCastException

Downcasting is not possible without upcasting. It is a technique to assign sub class object
(Reference is super type) to sub class reference variable.
Example :
        Animal a = new Lion();
          Lion l = (Lion) a;

l.roar();

Note : To avoid this ClassCastException we should use instanceof opertor.

14-11-2024

Programs :
----
package com.ravi.method_overriding;

class Bird
{
        public void fly()
        {
                System.out.println("Generic bird is flying..");
        }
}
class Parrot extends Bird
{
        public void fly()
        {
                System.out.println("Parrot bird is flying..");
        }
}
class Sparrow extends Bird
{
        public void fly()
        {
                System.out.println("Sparrow bird is flying..");
        }
}

public class MethodOverridingDemo1
{
        public static void main(String[] args)
        {
                Bird b = null;

                b = new Parrot(); b.fly(); //Dynamic Method Dispatch

                b = new Sparrow(); b.fly(); //Dynamic Method Dispatch

```
        }
}
```

@Override Annotation :

In Java we have a concept called Annotation, introduced from JDK 1.5 onwards. All the
annotations must be start with @ symbol.

@Override annotation is metadata (Giving information that method is overridden) and it is
optional but it is always a good practice to write @Override annotation before the Overridden
method so compiler as well as user will get the confirmation that the method is overridden
method and it is available in the super class.

If we use @Override annotation before the name of the overridden method in the sub class and
if the method is not available in the super class then it will generate a compilation error so it is
different from comments because comment will not generate any kind of compilation error if
method is not an overridden method, so this is how it is different from comment.

```java
package com.ravi.method_overriding;

class Animal
{
        public void sleep()
        {
                System.out.println("Generic Animal is sleeping");
        }
}

class Dog extends Animal
{
        public void sleep()
        {
                System.out.println("Dog Animal is sleeping");
        }
}
class Puppy extends Dog
{
        @Override
        public void sleep()
        {
                System.out.println("Puppy Animal is sleeping");
        }
}
```

```java
public class MethodOverridingDemo2
{
        public static void main(String[] args)
        {
                Animal a = new Puppy();
                a.sleep();
        }
}
```

Method Overloading is also possible in super and sub class as shown below

```java
package com.ravi.method_overriding;

class Bird
{
        public void fly()
        {
                System.out.println("Genric Bird is flying");
        }

        public void roam()
        {
                System.out.println("Generic Bird is roamig");
        }
}
class Parrot extends Bird
{
        //Method Overloading
        public double fly(double height)
        {
                System.out.println("Parrot is flying with "+height+" feet height");
                return height;
        }


        @Override
        public void roam()
        {
                System.out.println("Parrot Bird is roamig");

        }
}
public class MethodOverridingDemo3
```

```
{
        public static void main(String[] args)
        {
                Parrot p = new Parrot();
                p.fly(15.6);
                p.roam();
        }

}
```

Variable Hiding concept in upcasting :

```
class Super
{
   int x = 100;

}
class Sub exetnds Super
{
   int x = 200;  //Variable Hiding
}
```

Only non static methods are overridden in java but not the variables[variables are not overridden in java] because behavior will change but not the property(variable).

Note : In upcasting variable will be always executed besed on the current reference class variable.

```
package com.ravi.method_overriding;

class Animal
{
        protected String name = "Animal";

        public String roam()
        {
                return "Generic Animal is roaming";
        }
}

class Lion extends Animal
{
        protected String name = "Lion";  //Variable Hiding
```

```java
        @Override
        public String roam()
        {
                return "Lion is roaming";
        }
}

public class MethodOverridingDemo4
{
        public static void main(String[] args)
        {
                Animal a = new Lion();
                System.out.println(a.name+" : "+a.roam());

        }

}
```

Can we override private method ?

No, We can't override private method of super class because private methods are not visible
(not available) to the sub class hence we can't override.

We can't use @Override annotation on private method of sub class because it is not overridden
method, actually it is re-declared by sub class developer.

```java
class Vehicle
{
        private void digitalMeter()
        {
                System.out.println("Generic Digital Meter");
        }
}
class Car extends Vehicle
{
        //Re-declaration of Method [Not an Overridden Method]
        public void digitalMeter()
        {
                System.out.println("Car Digital Meter");
        }
}
```

```java
public class MethodOverridingDemo5
{
        public static void main(String[] args)
        {
                new Car().digitalMeter();


        }
}
```

Note :- private method of super class is not available or not inherited in the sub class so if the sub class declare the method with same signature then it is not overridden method, actually it is re-declared in the sub class.

Role of access modifier while overriding a method :

While overriding the method from super class, the access modifier of sub class method must be greater or equal in comparison to access modifier of super class method otherwise we will get compilation error.

In terms of accessibility, public is greater than protected, protected is greater than default (public > protected > default)
[default < protected < public]

**So the conclusion is we can't reduce the visibility of the method while overriding a method.

Note :- private method is not availble (visible) in sub class so it is not the part of method overriding.

```java
class Shape
{
        public void draw()
        {
                System.out.println("Generic Draw");
        }
}
class Square extends Shape
{
        @Override
        protected void draw()  //error
        {
                System.out.println("Drawing Square");
        }
```

```
}
public class MethodOverriding
{
        public static void main(String[] args)
        {
                Shape s = new Square();
                s.draw();
        }
}
```

15-11-2024

Co-variant in java :

In general we cann't change the return type of method while overriding a method. if we try to
change it will generate compilation error because in method overriding, return type of both the
methods must be compaitable as shown in the program below.

```
class Super
{
        public void show()
        {
                System.out.println("Super class show Method");
        }
}
class Sub extends Super
{
        @Override
        public int show()
        {
                System.out.println("Sub class show Method");
                return 0;
        }
}

public class IncompaitableReturnType
{
        public static void main(String[] args)
        {
                Super s = new Sub();
                s.show();
        }
}
```

Note : error, return type int is not compaitable with void.

But from JDK 1.5 onwards we can change the return type of the method in only one case that the return type of both the METHODS(SUPER AND SUB CLASS METHODS) MUST BE IN INHERITANCE RELATIONSHIP (IS-A relationship so it is compatible) called Co-Variant as shown in the program below.

Note :- Co-variant will not work with primitive data type, it will work only with classes.

Co-variant represents only one direction that meanse sub class method return type object we can assign super class method return object i.e in one direction.

```java
package com.ravi.upcasting_downcasting;

class A
{
}
class B extends A
{
}
class Super
{
        public A show()
        {
                System.out.println("Super class show Method");
                return null;
        }
}
class Sub extends Super
{
        @Override
        public B show()  //B class Object we can assign to A
        {
                System.out.println("Sub class show Method");
                return null;
        }
}
```

```java
public class CoVariant
{
        public static void main(String[] args)
        {
                Super s = new Sub();
                s.show();
        }
}
```

While working with co-variant (In the same direction), sub class method return type we can assign to super class method return then only it is compatible and it is co-variant as shown in the program.

```java
package com.ravi.upcasting_downcasting;

class Alpha
{
        public Alpha m1()
        {
                System.out.println("Alpha class m1 method");
                return this;
        }
}
class Beta extends Alpha
{
        public Beta m1()
        {
                System.out.println("Beta class m1 method");
                return this;
        }
}
public class CoVariantDemo1 {

        public static void main(String[] args)
        {
                Alpha a = new Beta();
                a.m1();

        }

}
```
----
package com.ravi.upcasting_downcasting;

```java
class AA
{
        public Object m1()
        {
                System.out.println("AA class m1 method");
                return this;
        }
}
class BB extends AA
{
        @Override
        public Integer m1()
        {
                System.out.println("BB class m1 method");
                return null;
        }
}

public class CoVariant {

        public static void main(String[] args)
        {
          AA a = new BB();
          a.m1();

        }

}
```

Progrm that describes Polymorphic behaviour of sub classes :

```java
class Animal
{
        public void sleep()
        {
                System.out.println("Generic Animal is sleeping");
        }
}
class Lion extends Animal
{
        @Override
        public void sleep()
```

```java
        {
                System.out.println("Lion is sleeping");
        }
}
class Dog extends Animal
{
        @Override
        public void sleep()
        {
                System.out.println("Dog is sleeping");
        }
}

public class MethodOverriding
{
        public static void main(String[] args)
        {
                Animal a = new Lion();
                accept(a);

                Animal a1 = new Dog();
                accept(a1);
        }

        public static void accept(Animal animal)
        {
                animal.sleep();
        }
}
```

Note : Based on the Object, JVM will call appropriate method.

```java
class Animal
{
        public void sleep()
        {
                System.out.println("Generic Animal is sleeping");
        }
}
class Lion extends Animal
{
        @Override
        public void sleep()
```

```java
        {
                System.out.println("Lion is sleeping");
        }

        public void roar()
        {
                System.out.println("Lion is roaring");
        }

}
class Dog extends Animal
{
        @Override
        public void sleep()
        {
                System.out.println("Dog is sleeping");
        }

        public void bark()
        {
                System.out.println("Dog is Barking");
        }
}

public class MethodOverriding
{
        public static void main(String[] args)
        {
                Animal a = new Lion();
                accept(a);

                Animal a1 = new Dog();
                accept(a1);
        }

        public static void accept(Animal animal)
        {
                animal.sleep();

                Lion lion = (Lion) animal; //Downcasting
                lion.roar();

        }
```

}

Note : Here we will get java.lang.ClassCastException because Dog object can't be converted into lion type.
----
instanceof operator :

It is a relational operator so it returns boolean value.
It is also a keyword.
It is used to describe whether a reference variable is holding an object of particular type or not.
There must be an IS-A relation between reference variable and partcular class OR interface type.
It is used to avoid java.lang.ClassCastException.

```java
//Programs
class Alpha
{
}
class Beta extends Alpha
{
}
public class InstanceOfOperator
{
        public static void main(String[] args)
        {
                Beta b = new Beta();

                if(b instanceof Alpha)
                {
                        System.out.println("b is pointing to Beta obj");
                }

        }


}
```

```java
package com.ravi.instance_of;

class Bird
{
}
```

```java
class Fish
{
}

public class InstanceOfDemo {

        public static void main(String[] args)
        {
                Fish f = new Fish();

                if(f instanceof Fish)
                {
                        System.out.println("f is pointing to Fish Object");
                }

                else if(f instanceof Bird)  //error
                {

                }


        }

}
```

```java
package com.ravi.instance_of;

class Test
{

}

public class InstanceOfDemo1
{
        public static void main(String[] args)
        {
           Test t1 = new Test();

           if(t1 instanceof Test)
           {
                   System.out.println("t1 is the instance of Test");
           }
```

```java
            else if(t1 instanceof Object)
            {
                    System.out.println("t1 is the instance of Object");
            }


        }

}
```

```java
package com.ravi.instance_of;

public class InstanceDemo2
{

    public static void main(String[] args)
        {
            Integer i = 45;

                if(i instanceof Integer)
                {
                        System.out.println("i is pointing Integer Object");
                }

                else if(i instanceof Number)
                {
                        System.out.println("i is pointing to Number Object");
                }

                else if(i instanceof Object)
                {
                        System.out.println("i is pointing to Object Object");
                }

}
```

```java
//Dynamic Polymorphism by using instanceof Operator :
-------------------------------------------------------
class Animal
{
        public void sleep()
        {
                System.out.println("Generic Animal is sleeping");
```

```java
        }
}
class Lion extends Animal
{
        @Override
        public void sleep()
        {
                System.out.println("Lion is sleeping");
        }

        public void roar()
        {
                System.out.println("Lion is roaring");
        }

}
class Dog extends Animal
{
        @Override
        public void sleep()
        {
                System.out.println("Dog is sleeping");
        }

        public void bark()
        {
                System.out.println("Dog is Barking");
        }
}

public class MethodOverriding
{
        public static void main(String[] args)
        {
                Animal a = new Lion();
                accept(a);

                Animal a1 = new Dog();
                accept(a1);
        }

        public static void accept(Animal animal) //Loose Coupling
        {
```

```java
                if(animal instanceof Lion)
                {
                        Lion lion = (Lion) animal;
                        lion.sleep();
                        lion.roar();
                }

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

                if(animal instanceof Dog)
                {
                        Dog dog = (Dog) animal;
                        dog.sleep();
                        dog.bark();
                }

        }
}
```

==

**What is Method Hiding in java ?
        OR
Can we override static method ?
        OR
Can we override main method ?

While working with method hiding we have all different cases :

Case 1 :

A public static method of super class by default available to sub class so, from sub class we can call super class static method with the help of Class name as well as object reference as shown in the below program :

```java
class Base
{
        public static void m1()
        {
                System.out.println("Base class static method");
        }
}
class Derived extends Base
```

```
{
}

public class StaticDemo
{
        public static void main(String[] args)
        {
          Derived.m1();

          Derived d = new Derived();
          d.m1();
        }
}
```

16-11-2024

Case 2 :

We can't override a static method with non static method because static method belongs to
class and non static method belongs to object, If we try to override static method with non static
method then it will generate an error i.e overridden method is static as shown below.

```
class Base
{
        public static void m1()
        {
                System.out.println("Static method of Base class");
        }
}
class Derived extends Base
{
        public void m1() //error
        {
                System.out.println("Non static method");

        }
}

public class StaticDemo
{
        public static void main(String[] args)
        {
```

```
        }
}
```

Case 3 :

We can't override any non static method with static method, If we try then it will generate an error, Overriding method is static.

```
class Base
{
        public void m1()
        {
                System.out.println("Static method of Base class");
        }
}
class Derived extends Base
{
        public static void m1() //error
        {
                System.out.println("Non static method");
        }
}
public class StaticDemo
{
        public static void main(String[] args)
        {
        }
}
```

So, the conclusion is we cannot overide static with non static method as well as non-static with static method because static method belongs to class and non-static method belongs to object.

Can we show that the following program is Method Hiding but not Overriding :

```
class A
{
        public static void m1()
        {
        }
}
class B extends A
{
        public static int m1() //Method Hiding
```

```
                {
                        return 0;
                }
        }
public class Test
{
        public static void main(String[] args)
        {


        }
}
```

From the above program it is clear that :
Method Hiding belons to static Method
Method Overriding belons to non-static Method

Case 4 :

We can't override static method because It belong to class but not object, If we write static
method in the sub class with same signature and compaitable return type then It is Method
Hiding but not Method Overriding here compiler will search the method of super class and JVM
will also execute the method of super class because method is not overridden.[Single copy and
belongs to class area, sharable by all the objects]

Note :- 1) We can't apply @Override annotation on static methods.

        2) Static methods can't be overridden so behavior is
             same for all the Objects hence it is Static Polymorphism.

StaticDemo.java

```
class Base
{
        public static void m1()
        {
                System.out.println("Static method of Base class");
        }
}
class Derived extends Base
{

        public static void m1() //Method Hiding
```

```java
        {
                System.out.println("Static method of Derived class");
        }
}

public class StaticDemo
{
        public static void main(String[] args)
        {
          Base b = new Derived();
          b.m1();
        }
}
```

StaticMethodHiding.java

```java
package com.ravi.copy;

class Animal
{
        public static void makeNoise()
        {
                System.out.println("Generic Animal is making Noise");
        }
}
class Dog extends Animal
{
        public static void makeNoise() //Method Hiding
        {
                System.out.println("Dog is making Noise");
        }
}

class Horse extends Animal
{
        public static void makeNoise() //Method Hiding
        {
                System.out.println("Horse is making Noise");
        }
}
public class StaticMethodHiding {

        public static void main(String[] args)
```

```
        {
                Animal a = new Horse();
                a.makeNoise();


        }

}
```
==
OCPJP Question :

```
package com.ravi.copy;

class Vehicle
{
        public int getHorsePower()
        {
                return 1000;
        }

        public void printHorsePower()
        {
                System.out.println(this.getHorsePower());
        }

}

class Car extends Vehicle
{
        @Override
        public int getHorsePower()
        {
                return 1200;
        }

        public void printHorsePower()
        {
                System.out.println(super.getHorsePower());
        }
}

public class MethodOverriding {

        public static void main(String[] args)
```

```java
		{
			Vehicle v = new Car();
			v.printHorsePower();


		}

}
```

```java
package com.ravi.copy;

class Bird
{
	protected int weathers = 2;

	public static void fly()
	{
		System.out.println("Bird is flying");
	}
}
class Parrot extends Bird
{
   protected int weathers = 4;

	public static void fly()
	{
		System.out.println("Parrot is flying");
	}
}

public class IQ {

	public static void main(String[] args)
	{
		Parrot p = new Parrot();
		Bird b = p;
		System.out.println(p.weathers);
		System.out.println(b.weathers);

		System.out.println("..........");
		p.fly();
		b.fly();

	}
```

}

*What is the limitation of 'new' keyword ?
                OR
What is the difference between new keyword and newInstance() method?
                OR
How to create the Object for the classes which are coming dynamically from the database or from some file at runtime.

The limitation with new keyword is, It demands the class name at the begning or at the time of compilation so new keyword is not suitable to create the object for the classes which are coming from database or files at runtime dynamically.

In order to create the object for the classes which are coming at runtime from database or files, we should use newInstance() method available in java.lang.Class class.

Methods :
----
public Object newInstance() : Predefined non static method of
                    java.lang.Class. It is used to
                        create the object for dynmacilly
                        loaded classes.

public native java.lang.Class getClass() :Predefined non static method of Object class.
----

class Employee{}

class Student{}

public class ObjectCreationAtRuntime
{
        public static void main(String[] args) throws Exception
        {
                Object obj = Class.forName(args[0]).newInstance();
                System.out.println("Object created for :"+obj.getClass().getName());
        }
}

Note : At runtime by using command line argument, we can pass
      class name so object will be created for the corresponding class.

```java
class Employee
{
  public void mySalary()
  {
        System.out.println("Employee Salary is 20K");
  }
}

class Developer
{
  public void mySalary()
  {
        System.out.println("Developer Salary is 70K");
  }
}
class Designer
{
  public void mySalary()
  {
        System.out.println("Designer Salary is 40K");
  }
}
public class ObjectCreationAtRuntime1
{
        public static void main(String[] args) throws Exception
        {
                Object obj = Class.forName(args[0]).newInstance();

                if(obj instanceof Employee)
                {
                        Employee emp = (Employee) obj;
                    emp.mySalary();
                }
                else if(obj instanceof Developer)
                {
                        Developer dev = (Developer) obj;
                    dev.mySalary();
                }
                else if(obj instanceof Designer)
                {
                        Designer des = (Designer) obj;
                    des.mySalary();
                }
```

}
}

Note : Creating the object by using newInstance() method and calling the non static method of the corresponding class.

18-11-2024
----
final keyword in java :

It is used to provide some kind of restriction in our program.
We can use final keyword in ways 3 ways in java.

1) To declare a class as a final. (Inheritance is not possible)
2) To declare a method as a final (Overriding is not possible)
3) To declare a variable (Field) as a final (Re-assignment is not possible)

1) To declare a class as a final :

Whenever we declare a class as a final class then we cann't extend or inherit that class otherwise we will get a compilation error.

We should declare a class as a final if the composition of the class (logic of the class) is very important and we don't want to share the feature of the class to some other developer to modify the original behavior of the existing class, In that situation we should declare a class as a final.

Declaring a class as a final does not mean that the variables and methods declared inside the class will also become as a final, only the class behavior is final that means we can modify the variables value as well as we can create the object for the final classes.

Note :- In java String and All wrapper classes are declared as final class.

class Ravi extends String
{

}

If we declare a class as final then only the class behavior is final, it variable can be modified.

final class A
{

```java
        private int x = 100;

        public void setData()
        {
                x = 120;
                System.out.println(x);
        }
}
class B extends A  //error [super class is final]
{
}
public class FinalClassEx
{
        public static void main(String[] args)
        {
                B b1 = new B();
                b1.setData();
        }
}
```

Here A class is final so, we can't inherit class A hence we will get compilation error.

```java
final class Test
{
        private int data = 100;

        public Test(int data)
        {
                this.data = data;
                System.out.println("Data value is :"+data);
        }
}
public class FinalClassEx1
{
        public static void main(String[] args)
        {
                Test t1 = new Test(200);

        }
}
```

We can create an object for final class as well as we can the modify the data.

Note : If a class contains private constructor then we should declare that class with final access modifier because we can't create sub class for the class which contains private constructor as shown in the program.

```
final class Sample
{
        private Sample() //Private Constructor
        {
        }
        public void m1()
        {
                System.out.println("Sample class m1 method");
        }
}
public class FinalClassEx2
{
        public static void main(String[] args)
        {

        }
}
```

Sealed class in Java :

It is a new feature introduced from java 15v (preview version) and become the integral part of java from 17v.

It is an improvement over final keyword.

By using sealed keyword we can declare classes and interfaces as sealed.

It is one kind of restriction that describes which classes and interfaces can extend or implement from Sealed class Or interface.

It is similar to final keyword with less restriction because here we can permit the classes to extend from the original Sealed class.

The class which is inheriting from the sealed class must be final, sealed or non-sealed.

The sealed class must have atleast one sub class.

We can also create object for Sealed class.

It provides the following modifiers :

1) sealed : Can be extended only through permitted class.

2) non-sealed : Can be extended by any sub class, if a user wants to give permission to its sub classes.

3) permits : We can provide permission to the sub classes, which are inheriting through Sealed class.

4) final : we can declare permitted sub class as final so, it cannot be extended further.

```java
package com.ravi.sealed;

sealed class Bird permits Parrot, Sparrow
{
        public void fly()
        {
                System.out.println("Generic Bird is flying");
        }
}
non-sealed class Parrot extends Bird
{
        @Override
        public void fly()
        {
                System.out.println("Parrot Bird is flying");
        }
}
final class Sparrow extends Bird
{
        @Override
        public void fly()
        {
                System.out.println("Sparrow Bird is flying");
        }
}

public class SealedDemo1
{
        public static void main(String[] args)
        {
```

```java
            Bird b = new Parrot();
            b.fly();
            b = new Sparrow();
            b.fly();


      }

}
```

```java
package com.ravi.sealed;

sealed class OnlineClass permits Mobile,Laptop
{
      public void attendOnlineJavaClass()
      {
            System.out.println("Online java class!!!");
      }
}
final class Mobile extends OnlineClass
{
      @Override
      public void attendOnlineJavaClass()
      {
            System.out.println("Attending Java class through mobile.");
      }
}
final class Laptop extends OnlineClass
{
      @Override
      public void attendOnlineJavaClass()
      {
            System.out.println("Attending Java class through Laptop.");
      }
}
public class SealedDemo2
{
      public static void main(String[] args)
      {
            OnlineClass c = null;
            c = new Mobile(); c.attendOnlineJavaClass();
            c = new Laptop(); c.attendOnlineJavaClass();
      }
```

}

---2) To declare a method as a final (Overriding is not possible)

Whenever we declare a method as a final then we can't override that method in the sub class otherwise there will be a compilation error.

We should declare a method as a final if the body of the method i.e the implementation of the method is very important and we don't want to override or change the super class method body by sub class method body then we should declare the super class method as final method.

```java
class A
{
        protected int a = 10;
        protected int b = 20;

    public final void calculate()
        {
                int sum = a+b;
                System.out.println("Sum is :"+sum);
        }
}
class B extends A
{
        @Override
        public void calculate() //final
        {
                int mul = a*b;
                System.out.println("Mul is :"+mul);
        }
}
public class FinalMethodEx
{
        public static void main(String [] args)
        {
                A a1 = new B();
                a1.calculate();
        }
}
```

```java
class Alpha
{
        private final void accept()
```

```java
        {
                System.out.println("Alpha class accept method");
        }
}
class Beta extends Alpha
{
        protected void accept()
        {
                System.out.println("Beta class accept method");
        }
}
public class FinalMethodEx1
{
        public static void main(String [] args)
        {
                new Beta().accept();
        }
}
```

Note : Here Program will compile and execute because private method of super class is not available to sub class.

----

3) To declare a variable/Field as a final :

In older langugaes like C and C++ we use "const" keyword to declare a constant variable but in java, const is a reserved word for future use so instead of const we should use "final" keyword.

If we declare a variable as a final then we can't perform re-assignment (i.e nothing but re-initialization) of that variable.

In java It is always a better practise to declare a final variable by uppercase letter according to the naming convention.

```java
class A
{
        final int A = 10;  //error
        public void setData()
        {
                A = 10;
                System.out.println("A value is :"+A);
        }
}
```

```
class FinalVarEx
{
        public static void main(String[] args)
        {
                final A a1 = new A();
                a1.setData();

                a1 = new A(); //error
                a1.setData();
        }
}
```

Common Topics (Sunday classes) :
----
 1) Nested class in java
 2) enum class (JDK 1.5)
 3) Object class and its Method (Online 9:30AM)
 4) Input Output and File Handling
 5) Wrapper classes in java

Abstraction : [Hiding the complexcity]

Showing the essential details without showing the background details is called abstraction.

In order to achieve abstraction we use the following two
concepts :

     1) Abstract class (we can achieve 0 - 100% abstraction)
     2) Interface (we can achieve 100 % abstraction)

Abstract class and abstract methods :

A class that does not provide complete implementation (partial implementation) is defined as an
abstract class.

An abstract method is a common method which is used to provide easiness to the programmer
because the programmer faces complexcity to remember the method name.

An abstract method observation is very simple because every abstract method contains abstract
keyword, abstract method does not contain any method body and at the end there must be a
terminator i.e ; (semicolon)

In java, whenever action is common but implementations are different then we should use

abstract method, Generally we declare abstract method in the super class and its implementation must be provided in the sub classes.

if a class contains at least one method as an abstract method then we should compulsory declare that class as an abstract class.

Once a class is declared as an abstract class we can't create an object for that class.

*All the abstract methods declared in the super class must be overridden in the sub classes otherwise the sub class will become as an abstract class hence object can't be created for the sub class as well.

In an abstract class we can write all abstract method or all concrete method or combination of both the method.

It is used to acheive partial abstraction that means by using abstract classes we can acheive partial abstraction(0-100%).

*An abstract class may or may not have abstract method but an abstract method must have abstract class.

Note :- We can't declare an abstract method as final, private and static (illegal combination of modifiers)

We can't declare an abstract class as a final.

//Program on abstract class and abstract method :

abstract class Shape
{
        public abstract void draw();
}

class Square extends Shape
{
        @Override
    public void draw()
        {
                System.out.println("Drawing Square");
        }
}
class Rectangle extends Shape
{

```java
        @Override
    public void draw()
        {
                System.out.println("Drawing Rectangle");
        }
}

public class AbstractMethodDemo1
{
        public static void main(String[] args)
        {
                Shape s = null;
                s = new Square(); s.draw();
                s = new Rectangle(); s.draw();
        }
}
```

```java
package com.ravi.abstraction;

abstract class Bike
{
        protected int speed = 60;

        public Bike()
        {
                System.out.println("Bike class Constructor...");
        }

        public void getBikeDetails()
        {
                System.out.println("It has two wheels....");
        }

        public abstract void run();
}

class Honda extends Bike
{

        @Override
        public void run()
        {
                System.out.println("Honda Bike is Running");
```

```
        }
}

public class AbstractDemo2
{
        public static void main(String[] args)
        {
                Bike obj = new Honda();
                System.out.println("Bike speed is :"+obj.speed);
                obj.getBikeDetails();
                obj.run();
        }

}
```

Note : Any constructor of abstract class will also be executed with sub class object using super keyword.

Program that describes abstract class may contain only concrete method (0 % abstraction)
----
package com.ravi.abstraction;

```
public abstract class Test
{
  public void m1()
  {
  }
}
```

IQ:

What is the advantage of writing constructor in the abstract class
----
As we know we can't create an object for abstract class but still
we can define instance variables to represent properties of an abstract class but these proprties
will be initialized through abstract class constructor (not through abstract class object) by using
sub class object via super keyword as shown in the program below.

AbstractDemo3.java
----

package com.ravi.abstract_demo;

```java
import org.w3c.dom.css.Rect;

abstract class Shape
{
        protected String shapeType;

        public Shape(String shapeType)
        {
                super();
                this.shapeType = shapeType;
        }

        public abstract void draw();
}

class Rectangle extends Shape
{
        public Rectangle(String shapeType)
        {
                super(shapeType);
        }

        @Override
        public void draw()
        {
                System.out.println("Drawing "+shapeType);
        }
}

class Circle extends Shape
{
        public Circle(String shapeType)
        {
                super(shapeType);
        }

        @Override
        public void draw()
        {
                System.out.println("Drawing "+shapeType);
        }
}
```

```java
public class AbstractDemo3
{
        public static void main(String[] args)
        {
                Shape ss = new Rectangle("Rectangle");
                ss.draw();

                ss = new Circle("Circle");
                ss.draw();

        }

}
```

//WAP that describes all the abstract methods must be overridden in the sub classes.

```java
package com.ravi.abstract_demo;

abstract class Alpha
{
        public abstract void show();
        public abstract void demo();
}
abstract class Beta extends Alpha
{
        @Override
        public void show() //+ demo();
        {
                System.out.println("Show method implemented in Beta class");
        }
}
class Gamma extends Beta
{
        @Override
        public void demo()
        {
                System.out.println("Demo method implemented in Gamma class");
        }
}
public class AbstractDemo4
{
   public static void main(String[] args)
   {
```

```java
        Gamma g = new Gamma();
        g.show();
        g.demo();
    }
}
```

Abstract class + Dynamic Poly + Array

```java
package com.ravi.abstract_demo;

abstract class Animal
{
        public abstract void checkup();
}
class Dog extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Dog is going for checkup");
        }
}
class Lion extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Lion is going for checkup");
        }
}

class Elephant extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Elephant is going for checkup");
        }
}

public class AbstractDemo5 {

        public static void main(String[] args)
```

```
        {
    Dog dogs[] = {new Dog(), new Dog(), new Dog() };
    Lion lions[] = {new Lion(), new Lion() };
    Elephant elephants[] = {new Elephant(), new Elephant()};

    animalChecking(dogs);
    animalChecking(lions);
    animalChecking(elephants);
        }

        public static void animalChecking(Animal ...animals)
        {
                for(Animal animal : animals)
                {
                        animal.checkup();
                }
        }


}
```

Anonymous inner class for overriding the method of abstract class
----
What is an anonumous inner class ?

An anonymous inner class is class defined inside a method without any name.

It is mainly used to extend a class OR implement an interface that means to create a sub type.

In anonymous inner class, class declaration and object creation
both are done in the same line i.e at the time of declaring the class.[; is compulsory]

Any inner class is represented by $ symbol. Anonymous inner class (which does not contain
any name) is represented by numeric value like 1,2,3 and so on.

```
class Super  //Super.class
{
        public void show()
        {
                System.out.println("Super class show method");
        }
}
```

```
public class AnonymousInnerClass //AnonymousInnerClass.class
{
        public static void main(String[] args)
        {
                //ANONYMOUS INNER CLASS DECLARATION AND OBJECT CREATION
                Super sub = new Super() //AnonymousInnerClass$1.class
                {
                        @Override
                        public void show()
                        {
                                System.out.println("Sub class show method");
                        }

                };
                sub.show();

                //ANONYMOUS INNER CLASS DECLARATION AND OBJECT CREATION
                Super sub1 = new Super() //AnonymousInnerClass$2.class
                {
                        @Override
                        public void show()
                        {
                                System.out.println("Sub1 class show method");
                        }

                };
                sub1.show();

        }

}
```

Anonymous inner class with abstract class :

```
package com.ravi.abstract_demo;

abstract class Vehicle
{
        public abstract void run();
}

public class AnonymousInnerClassDemo1
```

```java
{
        public static void main(String[] args)
        {
                Vehicle car = new Vehicle()
                {
                        @Override
                        public void run()
                        {
                                System.out.println("Car is running");

                        }

                };
                car.run();

                Vehicle bike = new Vehicle()
                {
                        @Override
                        public void run()
                        {
                                System.out.println("Bike is running");

                        }

                };
                bike.run();

        }

}
```
==
21-11-2024
----
interface
----
interface :
----
interface upto java 1.7

An interface is a keyword in java which is similar to a class which defines working functionality
of a class.

Upto JDK 1.7 an interface contains only abstract method that means there is a guarantee that

inside an interfcae we don't have concrete or general or instance methods.

From java 8 onwards we have a facility to write default and static methods.

By using interface we can achieve 100% abstraction concept because it contains only abstract methods.

In order to implement the member of an interface, java software people has provided implements keyword.

All the methods declared inside an interface is by default public and abstract so at the time of overriding we should apply public access modifier to sub class method.

All the variables declared inside an interface is by default public, static and final.

We should override all the abstract methods of interface to the sub classes otherwise the sub class will become as an abstract class hence object can't be created.

We can't create an object for interface, but reference can be created.

By using interface we can acheive multiple inheritance in java.

We can achieve loose coupling using interface.

Note :- inside an interface we can't declare any blocks (instance, static), instance variables (No properties) as well as we can't write constructor inside an interface.
----
package com.ravi.interface_ex;

sealed interface Moveable permits Car
{
        int SPEED = 90; //public + static + final
        void move();   //public + abstract
}
final class Car implements Moveable
{
        @Override
        public void move()
        {
                //SPEED = 120; [Invalid]
                System.out.println("Car is moving with :"+SPEED+ " Speed!!");
        }

```java
}
public class InterfaceDemo
{
        public static void main(String[] args)
        {
    Moveable m= new Car();
    m.move();
    System.out.println("Speed of the Car is :"+Moveable.SPEED);


        }

}
```

```java
package com.ravi.interface_ex;

interface Bank
{
        void deposit(double deposit);
        void withdraw(double withdraw);
        double getBalance();
}
class Customer implements Bank
{
        private double balance;

        public Customer(double balance)
        {
                super();
                this.balance = balance;
        }

        @Override
        public void deposit(double deposit)
        {
                if(deposit <=0)
                {
                 System.err.println("Amount can't be deposited");
                }
                else
                {
                        this.balance = this.balance + deposit;
                        System.out.println("Amount after deposit is :"+this.getBalance());
```

```java
            }
        }

        @Override
        public void withdraw(double withdraw)
        {
          if(this.balance < withdraw)
          {
                System.err.println("Insufficient Balance..");
          }
          else
          {
                this.balance = this.balance - withdraw;
                System.out.println("Amount after withdraw is :"+this.getBalance());
          }
        }

        @Override
        public double getBalance()
        {
                return this.balance;
        }

}

public class InterfaceDemo2 {

        public static void main(String[] args)
        {
          Customer raj = new Customer(10000);
          raj.deposit(1000);
          raj.withdraw(3000);

        }

}
```

Assignemnt :

```java
interface Calculator
{
  void sum(int x, int y);
  void sub(int x, int y);
```

```java
    void mul(int x, int y);
    void div(int x, int y);
}
class ArithmeticOperation implements Calculator
{
}
```

//Program on loose coupling using Dynamic Polymorphism :

Program on loose coupling :

Loose Coupling :- If the degree of dependency from one class object to another class is very
low then it is called loose coupling. [interface]

Tightly coupled :- If the degree of dependency of one class to another class is very high then it
is called Tightly coupled.

According to IT industry standard we should always prefer loose coupling so the maintenance of
the project will become easy.

High Cohesion [Encapsulation]:

Our private data must be accessible via public methods (setter and getters) so, in between data
and method we must have high cohesion (tight coupling) so, validation of outer data is possible.

6 files :

HotDrink.java(I)
----
package com.ravi.loose_coupling;

public interface HotDrink
{
    void prepare();
}

Tea.java
----
package com.ravi.loose_coupling;

public class Tea implements HotDrink {

        @Override

```java
        public void prepare()
        {
                System.out.println("Preparing Tea");

        }

}
```

Coffee.java

```java
package com.ravi.loose_coupling;

public class Coffee implements HotDrink {

        @Override
        public void prepare()
        {
                System.out.println("Preparing Coffee");

        }

}
```

Horlicks.java

```java
package com.ravi.loose_coupling;

public class Horlicks implements HotDrink
{
        @Override
        public void prepare()
        {
                System.out.println("Preparing Horlicks");

        }

}
```

Restaurant.java

```java
package com.ravi.loose_coupling;

public class Restaurant
```

```
{
  public static void acceptObject(HotDrink hd)  //hd = new Tea();
  {
        hd.prepare();
  }
}
```

Main.java

```
package com.ravi.loose_coupling;

public class Main
{
        public static void main(String[] args)
        {
                Restaurant.acceptObject(new Tea());
                Restaurant.acceptObject(new Coffee());
                Restaurant.acceptObject(new Horlicks());
        }

}
```

It is always better to take method return type as interface so we can return any implementer class object as shown in the example below

```
public HotDrink accept()
{
   return new Tea() OR new Coffee() OR new Horlicks() OR any future
   implementer class object..........................
}
```

Multiple Inheritance by using interface :

In a class we have a constructor so, it is providing ambiguity issue but inside an interface we don't have constructor so multiple inheritance is possible using interface.

The Implementer class constructor's super keyword will directly move to Object class constructor.[21-NOV]

MultipleInheritance.java

```
package com.ravi.multiple_Inheritance;
```

```java
interface A
{
        void m1();
}
interface B
{
        void m1();
}
class Implementer implements A,B
{
        @Override
        public void m1()
        {
                System.out.println("Multiple Inheritance is possible");
        }
}

public class MultipleInheritance {

        public static void main(String[] args)
        {
                Implementer i = new Implementer();
                i.m1();

        }

}
```

22-11-2024

Extending an interface :

One interface can extend another interface but it can't implement an interface as shown in the
program.

ExtendingInterface.java
----
package com.ravi.interface_ex;

interface A
{
        void m1();
}

```java
interface B extends A
{
        void m2();
}
class Implementer implements B
{
        @Override
        public void m1()
        {
                System.out.println("m1 method Implemented!!!");
        }

        @Override
        public void m2()
        {
                System.out.println("m2 method Implemented!!!");
        }
}

public class ExtendingInterface
{
        public static void main(String[] args)
        {
                Implementer i = new Implementer();
                i.m1();  i.m2();

        }

}
```
----
interface from JDK 1.8 onwards :

Limitation of abstract method OR Maintenance problem with interface in an Industry upto JDK 1.7

Limitation of abstract method :
        OR
Maintenance problem with interface in an Industry upto JDK 1.7

The major maintenance problem with interface is, if we add any new abstract method at the later stage of development inside an existing interface then all the implementer classes have to override that abstract method otherwise the implementer class will become as an abstract class

so it is one kind of boundation.

We need to provide implementation for all the abstract methods available inside an interface whether it is required or not required?

To avoid this maintenance problem java software people introduced default method inside an interface.

What is default Method inside an interface?
----
default method is just like concrete method which contains method body and we can write inside an interface from java 8 onwards.

default method is used to provide specific implementation for the implementer classes which are implmenting from interface because we can override default method inside the sub classes to provide our own specific implementation.

*By using default method there is no boundation to override the default method in the sub class, if we really required it then we can override to provide my own implementation.

by default, default method access modifier is public so at the time of overriding we should use public access modifier.

default method we can write inside an interface only but not inside a class.
----
4 files :

Vehicle.java(I)
----
package com.ravi.java_8_features;

public interface Vehicle
{
    void run();
    void horn();

    default void digitalMeter() //JDK 1.8
    {
        System.out.println("Digital Meter Facility is coming soon..");
    }

}

```
Car.java(C)
------------
package com.ravi.java_8_features;

public class Car implements Vehicle
{
        @Override
        public void run()
        {
                System.out.println("Car is Running");
        }

        @Override
        public void horn()
        {
                System.out.println("Car is having horn");
        }

        @Override
        public void digitalMeter()  //JDK 1.8
    {
        System.out.println("Car is having Digital Meter Facility");
    }
}

Bike.java(C)
------------
package com.ravi.java_8_features;

public class Bike implements Vehicle
{
        @Override
        public void run()
        {
                System.out.println("Bike is Running");
        }

        @Override
        public void horn()
        {
                System.out.println("Bike is having horn");
        }
}
```

Main.java(C)
----
package com.ravi.java_8_features;

public class Main
{
        public static void main(String[] args)
        {
          Vehicle v = null;
          v = new Car(); v.run(); v.horn(); v.digitalMeter();
          v = new Bike(); v.run(); v.horn();
        }

}

Priority of deafult and concrete method :

While working with class and interface, default method is having low
priority than concrete method, In the same way class is more powerfult than interface.

interface A
{
}
class B
{
}

class C extends B implements A {} //Valid

class C implements A extends B {} //Invalid

MethodPriority.java
----
package com.ravi.java_8_features;

interface Alpha
{
        default void m1() //JDK 1.8
        {
                System.out.println("Default Method of Alpha interface");
        }

```java
}

class Beta
{
        public void m1()
        {
                System.out.println("Concrete Method of Beta class");
        }
}

class Gamma extends Beta implements Alpha
{
}

public class MethodPriority
{
        public static void main(String[] args)
        {
                Gamma g = new Gamma();
                g.m1();
        }

}
```

Can we achieve multiple inheritance using default method :

Multiple inheritance is possible in java by using default method inside an interface, here we need to use super keyword to differenciate the super interface methods.
Before java 1.8, we have abstract method inside an interface but now we can write method body(default method) so, to execute the default method inside an interface we need to take super keyword with interface name(A.super.m1()).


```java
package com.ravi.java_8_features;

interface A
{
        default void m1()
        {
                System.out.println("Default method of interface A");
        }
}
```

```
interface B
{
        default void m1()
        {
                System.out.println("Default method of interface B");
        }
}

class C implements A,B
{
        @Override
        public void m1()
        {
          A.super.m1();
          B.super.m1();
          System.out.println("MI is possible");
        }
}

public class MultipleInheritanceUsingDefaultMethod
{
        public static void main(String[] args)
        {
                C c1 = new C();
                c1.m1();
        }

}
```

23-11-2024

What is static method inside an interface?

We can define static method inside an interface from java 1.8 onwards.

static method is only available inside the interface, It is not available to the implementer classes.

It is used to provide common functionality which we can apply/invoke from any BLC/ELC class.

By default static method of an inetrafce contains public access modifier.

2 files :

Calculator.java(I) [Available in com.ravi.static_method_demo]

```java
package com.ravi.static_method_demo;

public interface Calculator
{
   static double getSquare(double num)
   {
        return num*num;
   }


   static double getCube(double num)
   {
        return num*num*num;
   }
}
```

ELC.java(C) [Available in com.nit]

```java
package com.nit;

import com.ravi.static_method_demo.Calculator;

public class ELC
{
      public static void main(String[] args)
      {
            double result = Calculator.getSquare(4);
            System.out.println("Square is :"+result);

            result = Calculator.getCube(8);
            System.out.println("Cube is :"+result);

      }

}
```

Note : From the above program, It is clear that static method contains public access modifier so it is by default accessible from any package.

The following program explains that static method of an interface can be accessible through interface only.

```java
interface Printable
{
        static void print()
        {
                System.out.println("Static Method of interface");
        }
}

class Print implements Printable
{
}

public class StaticMethodDemo
{
        public static void main(String[] args)
        {
                //Print.print();  //Invalid
                Print p1 = new Print();
                //p1.print();  //Invalid

                Printable.print();
        }
}
```

Can we write main method inside an interface ?
----
interface is implicitly an abstract class, From java 8v onwards we can write static method so we can also write main method, Since it is a class so main method will be executed from interface body also.

Drawable.java(I)

```java
package com.ravi.static_method_demo;

public abstract interface Drawable
{
  public static void main(String[] args)
  {
          System.out.println("Main method inside an interface");
  }
}
```

Interface Static Method:
----
   a) Accessible using the interface name.
   b) Cannot be hidden by implementing classes.(Not Available)
   c) Can be called using the interface name only.

Class Static Method:
----
   a) Accessible using the class name.
   b) Can be hidden (not overridden) in subclasses by redeclaring a static method with the same signature.
   c) Can be called using the super class, sub class name as well as sub class object. also as shown in the program below.

```java
package com.ravi.interface_demo;

class A
{
        public static void m1()
        {
                System.out.println("Static method A");
        }
}
class B extends A
{

}
public class Demo
{
        public static void main(String [] args)
        {
                A.m1();
            B.m1();  //valid
               new B().m1(); //valid
        }
}
```

Introduction to Functional Programming ?

From JDK 8 onwards, Java also concentrated on function/method and introduced Functional Programming.

It is mainly used to write concise coding so the length of the method will be reduced.

What is a Functional interface ?

If an interface contains exactly one abstract method then it is called Functional interface.

```
interface Worker
{
   void work();  [SAM = Single abstract Method]
}
```

It may contain any number of default and static methods but it must contain only one abstract method.

We can provide @FunctionalInterface annotation to declare an interface as a Functional interface so the interface will not contain more than one abstract method.

```
@FunctionalInterface
interface Printable
{
        void print();  [SAM = Single abstract Method]

        default void m1()
        {
        }

        static void m2()
        {
        }

}
```

FunctionalInterfaceDemo.java

```
package com.ravi.functional_interface;

@FunctionalInterface
interface Payment
{
        double makePayment(double amount);
}

public class FunctionalInterfaceDemo
{
```

```java
        public static void main(String[] args)
        {
                Payment upi = new Payment()
                {
                        @Override
                        public double makePayment(double amount)
                        {
                                return amount + 50;
                        }
                };

                double payment = upi.makePayment(2000);
                System.out.println(payment);

        }

}
```

What is Lambda Expression in java ?

It is a new feature introduced in java from JDK 1.8 onwards.
It is an anonymous function i.e function without any name.
In java it is used to enable functional programming.
It is used to concise our code as well as we can remove boilerplate code.
Lambda will work only with functional interface.
If the body of the Lambda Expression contains only one statement then curly braces are optional.
We can also remove the variables type while defining the Lambda Expression parameter.
If the lambda expression method contains only one parameter then we can remove () symbol also.

In lambda expression return keyword is optional but if we use return keyword then {} are compulsory.

Independently Lamda Expression is not a statement.

It requires a target variable i.e functional interface reference only.

Lamda target can't be class or abstract class, it will work with functional interface only.

LambdaDemo1.java
----
package com.ravi.lambda;

```java
@FunctionalInterface
interface Vehicle
{
        void run();
}

public class LambdaDemo1
{
        public static void main(String[] args)
        {
                Vehicle car = ()-> System.out.println("Car is running");
                car.run();

                Vehicle bike = ()-> System.out.println("Bike is running");
                bike.run();

                Vehicle bus = ()-> System.out.println("Bus is running");
                bus.run();


        }

}
```

LambdaDemo2.java

```java
package com.ravi.lambda;

@FunctionalInterface
interface Calculate
{
        void doSum(int x, int  y);
}


public class LambdaDemo2 {

        public static void main(String[] args)
        {
                Calculate c1 = (c,d)-> System.out.println("Sum is :"+(c+d));
        c1.doSum(10, 20);
```

```
                }

        }

LambdaDemo3.java
-----------------
package com.ravi.lambda;

@FunctionalInterface
interface Length
{
        int getLength(String str);
}

public class LambdaDemo3 {

        public static void main(String[] args)
        {
                Length l = str -> str.length();

                System.out.println(l.getLength("India"));

        }

}
===============================================================
25-11-2024
-----------
//Program that shows we can write the logic inside Lambda Method Expression
import java.util.*;

@FunctionalInterface
interface Calculator
{
        double getSquareAndCube(int num);
}


public class LambdaDemo4
{

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

```java
            Calculator calc = num ->
            {
                    if(num<=0)
                    {
                            return -1;
                    }
                    else if(num % 2== 0)
                    {
                            return (num*num);
                    }
                    else
                    {
                            return (num*num*num);
                    }
            };

            Scanner sc = new Scanner(System.in);
            System.out.print("Enter a number :");
            int no = sc.nextInt();

            System.out.println(calc.getSquareAndCube(no));


        }

}
```

What is type parameter<T> in java ?

It is a technique through which we can make our application indepenedent of data type. It is
represented by <T>

In java we can pass Wrapper classes as well as User-defined classes to this type
parameter(Only Reference type is reqd).

We cannot pass any primitive type to this type parameter.

TypeParameterDemo.java

```java
package com.ravi.type_parameter;

class Accept<T>   //T is Type Parameter
{
```

```java
        private T data;

        //Parameterized Constructor
        public Accept(T data)   //Student data
        {
                super();
                this.data = data;
        }

        //getter
        public T getData()
        {
                return this.data;
        }
}


public class TypeParameterDemo
{
        public static void main(String[] args)
        {
                Accept<Integer> accInt = new Accept<Integer>(12);
                System.out.println("Integer Object is :"+accInt.getData());

                Accept<Double> accDou = new Accept<Double>(23.89);
                System.out.println("Double Object is :"+accDou.getData());

                Accept<Boolean> accBool = new Accept<Boolean>(true);
                System.out.println("Boolean Object is :"+accBool.getData());

                Accept<Student> accStudent = new Accept<Student>(new Student());
                System.out.println("Student Object is :"+accStudent.getData());
        }

}

class Student
{
        @Override
        public String toString()
        {
                return "Student";
        }
```

```
}
----------------------------------------------------------------
TypeParameterDemo1.java
------------------------
package com.ravi.type_parameter;

class Basket<T>
{
        private T element;  //element variable is Fruit type

        public T getElement()
        {
                return this.element;
        }

        public void setElement(T element) //Fruit element = new Apple();
        {
                this.element = element;
        }
}

class Fruit
{

}
class Apple extends Fruit
{
        @Override
        public String toString()
        {
                return "Apple";
        }
}

class Orange extends Fruit
{
        @Override
        public String toString()
        {
                return "Orange";
        }
}
```

```
public class TypeParameterDemo1
{
 public static void main(String[] args)
 {
        Basket<Fruit> b = new Basket<Fruit>();
        b.setElement(new Apple());

        Apple apple = (Apple) b.getElement();
        System.out.println(apple);

        System.out.println("............");
        b.setElement(new Orange());

        Orange orange = (Orange) b.getElement();
        System.out.println(orange);

 }
}
```

26-11-2024

Record class in java :
----
public abstract class Record extends Object.

record Student(){} //class Student extends Record [Compiler generated code]

It is a new feature introduced from java 17.(In java 14 preview version)

As we know only objects are moving in the network from one place to another place so we need to write BLC class with nessacery requirements to make BLC class as a Data carrier class.

Records are immutable data carrier so, now with the help of record we can send our immutable data from one application to another application.

It is also known as DTO (Data transfer object) OR POJO (Plain Old Java Object) classes.

It is mainly used to concise our code as well as remove the boiler plate code.

In record, automatically constructor will be generated which is known as canonical constructor and the variables which are known as components are by default final.

In order to validate the outer world data, we can write our own constructor which is known as

compact constructor.

Record will automatically generate the implemenation of toString(), equals(Object obj) and hashCode() method.

We can define static and non static method as well as static variable and static block inside the record. We cannot define instance variable and instance block inside the record.

We cann't extend or inherit records because by default every record is implicilty final and It is extending from java.lang.Reocrd class, which is an abstract class.

We can implement an interface by using record.

We don't have setter facility in record because by default components are final.
----
package com.ravi.record_demo;

import java.util.Objects;

public class EmployeeClass {
        private int empId;
        private String empName;

        public EmployeeClass(int empId, String empName)
        {
                super();
                this.empId = empId;
                this.empName = empName;
        }

        public int getEmpId() {
                return empId;
        }

        public void setEmpId(int empId) {
                this.empId = empId;
        }

        public String getEmpName() {
                return empName;
        }

        public void setEmpName(String empName) {

```java
                this.empName = empName;
        }

        @Override
        public String toString() {
                return "EmployeeClass [empId=" + empId + ", empName=" + empName + "]";
        }

        @Override
        public int hashCode() {
                return Objects.hash(empId, empName);
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                EmployeeClass other = (EmployeeClass) obj;
                return empId == other.empId && Objects.equals(empName, other.empName);
        }

}


package com.ravi.record_demo;


 //Canonical Constructor (Components are final)
public record EmployeeRecord(int empId, String empName)
{
        //Compact constructor (Validate the outer World Data)
  public EmployeeRecord
  {
        if(empId <= 0)
        {
                System.err.println("EmpId is invalid");
        }
  }
}
```

```java
package com.ravi.record_demo;

public class Main
{
        public static void main(String[] args)
        {
                EmployeeClass e1 = new EmployeeClass(111, "Scott");
                System.out.println(e1);
                EmployeeClass e2 = new EmployeeClass(111, "Scott");
                System.out.println(e1.equals(e2));
                System.out.println(e1.getEmpName());

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

                EmployeeRecord r1 = new EmployeeRecord(222, "Virat");
                System.out.println(r1);
                EmployeeRecord r2 = new EmployeeRecord(222, "Virat");
                System.out.println(r1.equals(r2));
                System.out.println(r1.empName());


        }

}
```

Working with predefined functional interfaces :

In order to help the java programmer to write concise java code in day to day programming java
software people has provided the following predefined functional interfaces

1) Predicate<T>   boolean test(T x);
2) Consumer<T>   void accept(T x);
3) Function<T,R>   R apply(T x);
4) Supplier<T>   T get();
5) BiPredicate<T,U>   boolean test(T x, U y);
6) BiConsumer<T, U>   void accept(T x, U y);
7) BiFunction<T,U,R>   R apply(T x, U y);
8) UnaryOperator<T>   T apply(T x)
9) BinaryOperator<T>   T apply(T x, T y)

Note :-

All these predefined functional interfaces are provided as a part of java.util.function sub package.

Predicate<T> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It contains an abstract method test() which takes type parameter <T> and returns boolean value.

The main purpose of this interface to test one argument boolean expression.

```
@FunctionalInterface
public interface Predicate<T>
{
  boolean test(T x);
}
```

Note :- Here T is a "type parameter" and it can accept any type of User defined class as well as Wrapper class like Integer, Float, Double and so on.

We can't pass primitive type.

//By using Predicate verify whether the number is even or odd.

```
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo1
{
        public static void main(String[] args)
        {
                Predicate<Integer> p1 = num -> num % 2 ==0;

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter a number :");
                int num = sc.nextInt();
```

```java
                boolean result = p1.test(num);
                System.out.println("Is "+num+" even ??"+result);
                sc.close();



        }

}
```

//By using Predicate verify whether my name starts with 'A' or /not

```java
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo2 {

        public static void main(String[] args)
        {
                Predicate<String> p2 = name -> name.startsWith("A") ;

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter your Name :");
                String name = sc.next();

                System.out.println(name+" starts with A :"+p2.test(name));

                sc.close();

        }

}
```

//Using Predicate verify whether you are pass or fail in exam
based on the marks input

```java
package com.ravi.functional_interface;

import java.util.function.Predicate;
```

```java
record Exam()
{
        public boolean passExam(int marks)
        {
                if(marks > 62)
                {
                        return true;
                }
                else
                {
                        return false;
                }
        }
}

public class PredicateDemo3 {

        public static void main(String[] args)
        {
                Predicate<Exam> p3 = exam -> exam.passExam(40);
                System.out.println(p3.test(new Exam()));
        }

}
```

27-11-2024

Using Predicate verify whether my name is Ravi or not ?

```java
import java.util.function.*;
import java.util.*;

public class PredicateDemo4
{
        public static void main(String[] args)
        {
                Predicate<String> p4 =  str -> str.equalsIgnoreCase("Ravi");

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter your Name :");
                String name = sc.next();

                System.out.println("Are you Ravi :"+p4.test(name));
```

```
                    sc.close();

            }
    }
    ------------------------------------------------------------
    Consumer<T> functional interface :
    ----------------------------------------
    It is a predefined functional interface available in java.util.function sub package.

    It contains an abstract method accept() and returns nothing. It is used to accept the parameter
    value or consume the value.

    @FunctionalInterface
    public interface Consumer<T>
    {
        void accept(T x);
    }

    Program :
    -----------
    package com.ravi.consumer;

    import java.util.function.Consumer;

    record Employee()
    {

    }

    public class ConsumerDemo {

            public static void main(String[] args)
            {
                    Consumer<Integer> c1 =  num -> System.out.println("Integer Object : "+num);
                    c1.accept(12);

                    Consumer<Double>  c2 = num -> System.out.println("Double Object :"+num);
                    c2.accept(23.89);

                Consumer<Employee> c3 = emp -> System.out.println("Employee Object :"+emp);
                    c3.accept(new Employee());
```

```
        }

}
```

Function<T,R> functional interface :

Type Parameters:

T - the type of the input to the function.

R - the type of the result of the function.

It is a predefined functional interface available in java.util.function sub package.

It provides an abstract method apply that accepts one argument(T) and produces a result(R).

Note :- The type of T(input) and the type of R(Result) both will be decided by the user.

```java
@FunctionalInterface
public interface Function<T,R>
{
   public abstract R apply(T x);
}
```

Programs :

Program to find out a cube of a number by using Function functional interface.

FunctionDemo1.java

```java
package com.ravi.function_demo;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionDemo1
{
        public static void main(String[] args)
        {
                Function<Integer, Integer> fn1 = num -> num*num*num;

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter a number :");
                int num = sc.nextInt();
```

```java
            Integer cube = fn1.apply(num);
            System.out.println("Cube of "+num+" is "+cube);
            sc.close();
        }

}
```

FunctionDemo2.java

```java
package com.ravi.function_demo;

import java.util.Scanner;
import java.util.function.Function;

//finding the length of the String
public class FunctionDemo2
{
        public static void main(String[] args)
        {
                Function<String, Integer> fn2 = str -> str.length();
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the name :");
                String name = sc.nextLine();


                Integer length = fn2.apply(name);
                System.out.println("Length of :"+name+ " is :"+length);
                sc.close();


        }

}
```

Supplier<T> prdefined functional interface :

It is a predefined functional interface available in java.util.function sub package.

It provides an abstract method get() which does not take any argument but produces/supply/return a value of type T.

@FunctionalInterface

```java
public interface Supplier<T>
{
    T get();
}

//Programs
package com.ravi.supplier;

import java.util.function.Supplier;

public class SupplierDemo1 {

    public static void main(String[] args)
    {
        Supplier<String> s1 = () -> 40 + 40+" Ravi "+ 80 + 80;
        System.out.println(s1.get());
    }

}
```

```java
package com.ravi.supplier;

import java.util.Scanner;
import java.util.function.Supplier;

record Product(int productId, String productName, double productPrice)
{

}

public class SupplierDemo2 {

    public static void main(String[] args)
    {
        Supplier<Product> s2 = () ->
        {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter Product Id :");
            int id = sc.nextInt();

            System.out.print("Enter Product Name :");
            String name = sc.nextLine();
            name = sc.nextLine();
```

```
            System.out.print("Enter Product Price :");
            double price = sc.nextDouble();

            return new Product(id, name, price);

        };

            Product obj = s2.get();
            System.out.println(obj);


    }

}
```

How to create our own functional interfaces with Type parameter :

```
package com.ravi.custom_fi;

@FunctionalInterface
interface MyInterface<T,U,V,R>
{
        public abstract R apply(T t, U u, V v);
}

public class UserDefinedFunctionalInterface
{
        public static void main(String[] args)
        {
            MyInterface<Integer,Integer, Integer, String> m1 = (a,b,c)-> ""+a+b+c;
            System.out.println(m1.apply(100, 200, 300));
        }
}
```

Note : It is clear that we can define our own functional interface.

BiPredicate<T,U> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a predicate (a boolean-valued function) OF

TWO ARGUMENTS.

The BiPredicate interface has method named test, which takes two parameters and returns a boolean value, basically this BiPredicate is same with the Predicate, instead, it takes 2 arguments for the metod test.

```java
@FunctionalInterface
public interface BiPredicate<T, U>
{
    boolean test(T t, U u);
}
```

Type Parameters:

T - the type of the first argument to the predicate
U - the type of the second argument the predicate
Note : return type is boolean.

```java
import java.util.function.*;
public class Lambda11
{
        public static void main(String[] args)
   {
      BiPredicate<String, Integer> filter = (x, y) ->
                  {
         return x.length() == y;
      };

      boolean result = filter.test("Ravi", 4);
      System.out.println(result);

      result = filter.test("Hyderabad", 10);
      System.out.println(result);
         }
}
```

```java
import java.util.function.BiPredicate;

public class Lambda12
{
  public static void main(String[] args)
  {
```

```java
    // BiPredicate to check if the sum of two integers is even
    BiPredicate<Integer, Integer> isSumEven = (a, b) -> (a + b) % 2 == 0;

        System.out.println(isSumEven.test(2, 3));
        System.out.println(isSumEven.test(5, 7));
    }
}
```

BiConsumer<T, U> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation that accepts two input arguments and returns no result.

It takes a method named accept, which takes two parameters and performs an action without returning any result.

```java
@FunctionalInterface
public interface BiConsumer<T, U>
{
    void accept(T t, U u);
}
```

```java
import java.util.function.BiConsumer;

public class Lambda13
{
  public static void main(String[] args)
  {
    BiConsumer<Integer, String> updateVariables =  (num, str) ->
                {
          num = num * 2;
          str = str.toUpperCase();
          System.out.println("Updated values: " + num + ", " + str);
      };


      int number = 15;
      String text = "nit";

      updateVariables.accept(number, text);
```

```
        // Values after the update (note that the original values are unchanged)
        System.out.println("Original values: " + number + ", " + text);
    }
}
```

BiFunction<T, U, R> Functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a function that accepts two arguments and produces a result R.

The BiFunction interface has a method named apply that takes two arguments and returns a result.

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{
    R apply(T t, U u);
}
```

```
import java.util.function.BiFunction;

public class Lambda14
{
  public static void main(String[] args)
  {
      // BiFunction to concatenate two strings
    BiFunction<String, String, String> concatenateStrings = (str1, str2) -> str1 + str2;

      String result = concatenateStrings.apply("Hello", " Java");
      System.out.println(result);


     // BiFunction to find the length two strings
    BiFunction<String, String, Integer> concatenateLength = (str1, str2) -> str1.length() +
str2.length();

      Integer result1 = concatenateLength.apply("Hello", "Java");
      System.out.println(result1);
```

```
  }
}
```

UnaryOperator<T> :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

It has a single type parameter, T, which represents both the operand type and the result type.

```java
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,R>
{
  public abstract T apply(T x);
}
```

```java
import java.util.function.*;
public class Lambda15
{
        public static void main(String[] args)
        {
                UnaryOperator<Integer> square = x -> x * x;
        System.out.println(square.apply(5));

                UnaryOperator<String> concat = str ->
        str.concat("base");
                 System.out.println(concat.apply("Data"));
        }
}
```

BinaryOperator<T>

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation upon two operands of the same type, producing a result of the same type as the operands.

This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

It has two parameters of same type, T, which represents both the operand types and the result type.

```java
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,U,R>
{
  public abstract T apply(T x, T y);
}
```

```java
import java.util.function.*;
public class Lambda16
{
        public static void main(String[] args)
        {
                BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(3, 5));
        }
}
```

Does an interface extends any class ?

An interface can't extend a class, It can extend only interface.

Every public method of Object class is implicitly re-declare inside every interface as an abstract method to support upcasting.

We can't override any public method of object class as a
default method inide interface.

```java
package com.ravi.interface_member;

interface Alpha
{

}
public class InterfaceMember1 {

        public static void main(String[] args)
        {
                Alpha a1 = null;
                a1.hashCode();
                a1.toString();
                a1.equals(null);
```

```
        }
}
```

Note : An interface contains all the public method of Object class as an abstract method.

```
package com.ravi.interface_member;

@FunctionalInterface
interface Beta
{
        public String toString();

        public int hashCode();

        public boolean equals(Object obj);

        void accept(); //SAM

}

public class InterfaceMember2 {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

        }

}
```

Note : A functional interface can also contain Object class method.

```
package com.ravi.interface_member;

interface Gamma
{
        //We can't override Object class method as a default method inside interface

        /*default int hashCode()
         {
                 return 10;
         }*/

}
```

```java
public class InterfaceDemo3 {

        public static void main(String[] args)
        {
                // TODO Auto-generated method stub

        }

}
```

Note : We can't override public method of Object class as a default method inside an interafce.

Interface from java 9v version

Yes, From java 9 onwards we can also write private static and private non-static methods inside an interface.

The folowing are the advantages of writing private methods :
 a) Code Reusability
 b) Hiding the implementation details by writing the logic
    inside private method (We can achieve 100% abstraction)

These private methods will improve code re-usability inside interfaces.

For example, if two default methods needed to share common and confidential code, a private method would allow them to do so, but without exposing that private method to it□s implementing classes.

Using private methods in interfaces have four rules :

1) private interface method cannot be abstract.
2) private method can be used only inside interface.
3) private static method can be used inside other static and non-static interface methods.
4) private non-static methods cannot be used inside private static methods.
----
package com.ravi.interface_member;

interface Worker
{
        public abstract void m1(); //JDK 1.0

        public default void m2() //JDK 1.8
```

```java
	{
		m4();
		m5();
	}

	public static void m3()  //JDK 1.8
	{
		m5();

	}

	private void m4()   //Java 9 [Private non static method]
	{
		System.out.println("Private non static method");
	}

	private static void m5() //Java 9 [Private static method]
	{
		System.out.println("Private static method");
	}
}

class Implementer implements Worker
{
	@Override
	public void m1()
	{
		System.out.println("M1 method of Implementer class");
	}
}
public class InterfaceNewFeature {

	public static void main(String[] args)
	{
		Implementer i = new Implementer();
		i.m1();  //abstract method
		i.m2();  //default
		Worker.m3(); //public static
	}

}
```

Note : We can achieve 100% abstraction by using private method inside interface.

----

*What is marker interface in java ?

A marker interface is an interface which does not contain any field or method, basically a marker interface is an empty interface or tag interface.

public interface Drawable //Marker interface
{

}

*The main purpose of Marker interface to provide additional information to the JVM regarding the object like object is Serilizable, Clonable OR randomly accessible or not.

In java we have 3 predefined marker interfaces : java.io.Serilizable, java.lang.Cloneable, java.util.RandomAccess.

Note : We can create our own marker interface by using instanceof operator.

 ****What is difference between abstract class and interface ?

The following are the differences between abstract class and interface.

1) An abstract class can contain instance variables but interface variables are by default public , static and final.

2) An abstract class can have state (properties) of an object but interface can't have state of an object.

3) An abstract class can contain constructor but inside an interface we can't define constructor.

4) An abstract class can contain instance and static blocks but inside an interface we can't define any blocks.

5) Abstract class can't refer Lambda expression but using Functional interface we can refer Lambda Expression.

6) By using abstract class multiple inheritance is not possible but by using interface we can achieve multiple inheritance.

==========================OOPs===============================

02-12-2024

Exception Handling :

What is an execption ?

An execption is a runtime error.

An execption is an abnormal situation or un-expected situation in a normal execution flow.

An exception encounter due to dependency, if one part of the program is dependent to another part then there might be a chance of getting Exception.

AN EXCEPTION ALSO ENCOUNTER DUE TO WRONG INPUT GIVEN BY THE USER.

Different Crieteria of Exception :

The following are the different criteria for exception :

1) java.lang.ClassCastException

   If we assign super class object to sub class reference
   variable then at runtime we will java.lang.ClassCastException

   Dog d1 = (Dog) new Animal();

2) java.lang.ArithmeticException
   If we divide a number by zero (int value) then at runtime we will get an exception i.e
java.lang.ArithmeticException

   System.out.println(12/0);

3) java.lang.ArrayIndexOutOfBoundsException
   If we try to access the index of an array which is not
   available or value is not available then we will get
   java.lang.ArrayIndexOutOfBoundsException.

   int []arr = {10,20,30};
   System.out.println(arr[3]);

4) java.lang.NegativeArraySizeException
   An array size must be positive integer, if we pass any
   Negative value then we will get java.lang.NegativeArraySizeException

   int arr[] = new int[-5];

5) java.lang.NullPointerException
   If we call any non static method on reference variable
   which is pointing to null then we will get
   java.lang.NullPointerException

   Case 1:

   String str = null;
   System.out.println(str.length());

   Case 2:

   Scanner sc = new Scanner(System.in);
   System.out.println("Enter your Name :");
   String name = sc.nextLine(); //"null"

   System.out.println(name.length());

 6) java.lang.NumberFormatException :

   If we try to convert any String value into primitive (int) or Wrapper (Integer) type and if the
number is not in a
   proper format then we will get java.lang.NumberFormatException.

   String str = "NIT";
   Integer valueOf = Integer.valueOf(str);
   int i = Integer.parseInt(str);

7) java.util.InputMismatchException
   At the time of reading the data from Scanner class,if we read the data not in a proper format
then we will get an exception java.util.InputMismatchException

   Scanner sc = new Scanner(System.in);
   System.out.println("Enter your Age :");
   int age = sc.nextInt(); //Eleven
   System.out.println("Your Age is :"+age);

Exception Hierarchy :

This Exception hierarchy is available in the diagram (Exception_Hierarchy.png)

Note :- As a developer we are responsibe to handle the Exception. System admin is responsibe to handle the error because we cannot recover from error.

Exception format :

The java software people has provided the format of exception so whenever we print exception object then the format is

Fully Qualified Name : errorMesage

Package Name + Class Name : errorMessage

WAP in java that describes Exception is the super class of all the exceptions (Checked + unchecked) in java.

package com.ravi.exception;

public class ExceptionDemo {

        public static void main(String[] args)
        {
          Exception e1 = new ArithmeticException("Error Message");
          System.out.println(e1);

          Exception e2 = new InterruptedException("Thread is Interrupted");
          System.out.println(e2);

        }
}

03-12-2024

WAP that describes that whenever an exception is encounter in the program then program will be terminated in the middle.

```java
package com.ravi.basic;

import java.util.Scanner;

public class AbnormalTermination
{
        public static void main(String[] args)
        {
                System.out.println("Main method Started!!!");

                Scanner sc = new Scanner(System.in);

                System.out.print("Enter the value of x :");
                int x = sc.nextInt();

                System.out.print("Enter the value of y :");
                int y = sc.nextInt();

                int result = x/y;
                System.out.println("Result is :"+result);

                System.out.println("Main method Completed!!!");
                sc.close();

        }
}
```

In the above program, If we put the value of y as 0 then program will be terminated in the middle, IT IS CALLED ABNORMAL TERMINATION.
Actually JVM has a default exception handler which is responsible to handle the execption and terminate the program in the middle abnormaly.

Key points to remember :

-> With try block we can write either catch block or finally block or both.
-> In between try and catch we can't write any kind of statement.
-> try block will trace our program line by line.
-> If we have any exception inside the try block,With the help of JVM, try block will automatically create the appropriate Exception object and then throw the Exception Object to the nearest catch block.
-> In the try block whenever we get an exception the control will directly jump to the nearest catch block so the remaining code of try block will not be executed.
 -> catch block is responsible to handle the exception.

-> catch block will only execute if there is an exception inside try block.

----

try block :

Whenever our statement is error suspecting statement OR Risky statement then we should write that statement inside the try block.

try block must be followed either by catch block or finally block or both.

*try block is responsible to trace our code line by line, if any execption is encountered then with the help of JVM, TRY BLOCK WILL CREATE APPROPRIATE EXECPTION OBJECT, AND THROW THIS EXCEPTION OBJECT to the nearest catch block.

After the execption in the try block, the remaining code of try block will not be executed because control will directly transfer to the catch block.

In between try and catch block we cannot write any kind of statement.

catch block :

The main purpose of catch block to handle the exception which is thrown by try block.

catch block will only executed if there is an exception in the try block.

```java
package com.ravi.basic;

import java.util.Scanner;

public class TryDemo
{
        public static void main(String[] args)
        {
                System.out.println("Main method started....");
                Scanner sc = new Scanner(System.in);

                try
                {
                        System.out.print("Enter the value of x :");
                        int x = sc.nextInt();

                        System.out.print("Enter the value of y :");
                        int y = sc.nextInt();
```

```
                    int result = x/y;
                    System.out.println("Result is :"+result);
                    System.out.println("End of try block");

            }
            catch(Exception e)
            {
                    System.out.println("Inside Catch Block");
                    System.err.println(e);
            }
            System.out.println("Main method ended....");
            sc.close();
        }
}
```

In the above program if we put the value of y as 0 but still program will be executed normally because we have used try-catch so it is a
normal termination even we have an exception in the program.

```
public class ExceptionDemo
{
        public static void main(String[] args)
        {
                try
                {
                  // System.out.println(10/0);
                            //OR
                  throw new ArithmeticException("Ravi is dividing by zero");

                }
                catch (Exception e)
                {
                        System.out.println("Inside Catch block");
                        System.out.println(e);
                }
        }
}
```

From the above program it is clear that try block implicitly creating the exception object with the help of JVM and throwing the execption object to the nearest catch block.

```
class Ravi
```

```
{

}

public class ExceptionDemo
{
        public static void main(String[] args)
        {
                try
                {
                        throw new Ravi();
                }
                catch (Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

Note : We will get compiletion error because Ravi class does not belong to exception hierarchy so it is not a throwable object.

```
package com.ravi.basic;

import java.util.Scanner;

public class CustomerDemo
{
        public static void main(String[] args)
        {
                System.out.println("Welcome client, Welcome to my application");
                Scanner sc = new Scanner(System.in);

                try
                {
                        System.out.print("Enter the value of a :");
                        int a = sc.nextInt();

                        System.out.print("Enter the value of b :");
                        int b = sc.nextInt();

                        int result = a/b;
                        System.out.println("Result is :"+result);
```

```
                }
                catch(Exception e)
                {
                   System.err.println("Sir, Don't put zero here");
                }

                sc.close();
                System.out.println("Thank you 4 visiting my application");

        }

}
```

Note : The main purpose of exception handling to provide user-friendly message so client can enjoy the services of software/websites.

Exception handlinag = No Abnormal Termination + User-friendly message on wrong input given by the client.

04-12-2024

Throwable class Method to print Exception :

Throwable class has provided the following three methods :

1) public String getMessage() :- It will provide only error message.

2) public void printStackTrace() :- It will provide the complete details regarding exception like exception class name, exception error message, exception class location, exception method name and exception line number.

3) public String toString() :- It will convert the exception into String representation.

```
package com.ravi.basic;

public class PrintStackTrace
{
        public static void main(String[] args)
        {
                System.out.println("Main method started...");
                try
                {
                        String x = "NIT";
                        int y = Integer.parseInt(x);
```

```
                    System.out.println(y);
            }
            catch(Exception e)
            {
                    e.printStackTrace(); //For complete Exception details
                    System.out.println("-------------------------");
                    System.out.println(".........................");
                    System.err.println(e.getMessage()); //only for Exception message
                    System.out.println(".............");
                    System.err.println(e.toString());
            }
            System.out.println("Main method ended...");

    }

}
```

Working with Specific Exception :

While working with exception, in the corresponding catch block we can take Exception (super class) which can handle any type of Exception.

On the other hand we can also take specific type of exception (ArithmetiException, InputMismatchException and so on) which will handle only one type i.e specific type of exception.

```
package com.ravi.basic;

import java.util.InputMismatchException;
import java.util.Scanner;

public class SpecificException
{
        public static void main(String[] args)
        {
    System.out.println("Main started");

                Scanner sc = new Scanner(System.in);

                try
                {
                        System.out.print("Enter your Roll :");
                        int roll = sc.nextInt();
```

```java
                System.out.println("Your Roll is :"+roll);

            }
            catch(InputMismatchException e)
            {
                    e.printStackTrace();
            }
            sc.close();
            System.out.println("Main ended");
        }
}
```

```java
public class ExceptionDemo
{
        public static void main(String[] args)
        {
                try
                {
                        System.out.println("Inside Try!!!");
                        throw new OutOfMemoryError();
                }
                catch (Exception e)
                {
                        System.out.println("Inside Catch!!!");
                        System.out.println(e);
                }
        }
}
```

Note : OutOfMemoryError can't be handled by Exception class.We need to write Error OR Throwable class.

Working with Infinity and Not a number(NaN) :

10/0   -> Infinity (Java.lang.ArithmeticException)
10/0.0 -> Infinity (POSITIVE_INFINITY)

0/0   -> Undefined (Java.lang.ArithmeticException)
0/0.0 -> Undefined (NaN)

While dividing a number with Integral literal in both the cases i.e Infinity (10/0) and Undefined

(0/0) we will get java.lang.ArithmeticException because java software people has not provided any final, static variable support to deal with Infinity and Undefined.

On the other hand while dividing a number with with floating point literal in the both cases i.e Infinity (10/0.0) and Undefined (0/0.0) we have final, static variable support so the program will not be terminated in the middle which are as follows

10/0.0 = POSITIVE_INFINITY
-10/0.0 = NEGATIVE_INFINITY
0/0.0 = NaN

java.lang.Float and java.lang.Double classes are provided the support for these final and static variable, the same OR same type of variables are not available in Integeral Literal classes.

```java
package com.ravi.basic;

public class InfinityFloatingPoint
{
        public static void main(String[] args)
        {
          System.out.println("Main Started");
          System.out.println(10/0.0); //Infinity
          System.out.println(-10/0.0); //-Infinity
          System.out.println(0/0.0);  //NaN
          System.out.println(10/0);
          System.out.println("Main Ended");
        }

}
```

Working with multiple try catch :

According to our application requirement we can provide multiple try-catch in my application to work with multiple execptions.

```java
package com.ravi.basic;
public class MultipleTryCatch
{
        public static void main(String[] args)
        {
          System.out.println("Main method started!!!!");
```

```java
        try
        {
                int arr[] = {10,20,30};
                System.out.println(arr[3]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
                System.err.println("Array index is out of limit!!!");
        }

        try
        {
                String str = null;
                System.out.println(str.length());
        }
        catch(NullPointerException e)
        {
                System.err.println("ref variable is pointing to null");
        }

        System.out.println("Main method ended!!!!");
        }
}
```

Note : Here we are getting all the execptions messages through catch blocks at a time so it is not a better approach from client point of view, We should always provide only one error message to our client.

----

* Single try with multiple catch block :

According to industry standard we should write try with multiple catch blocks so we can provide proper information for each and every exception to the end user.

While working with multiple catch block always the super class catch block must be last catch block.

From java 1.7v this multiple exceptions we can write in a single catch block by using | symbol.

If try block is having more than one exception then always try block will entertain only first exception because control will transfer to the nearest catch block.

package com.ravi.basic;

```java
public class MultyCatch
{
        public static void main(String[] args)
        {
                System.out.println("Main Started...");
                try
                {
                        int c = 10/0;
                        System.out.println("c value is :"+c);

                        int []x = {12,78,56};
                        System.out.println(x[4]);
                }

                catch(ArrayIndexOutOfBoundsException e1)
                {
                        System.err.println("Array is out of limit...");
                }
                catch(ArithmeticException e1)
                {
                        System.err.println("Divide By zero problem...");
                }
                catch(Exception e1)
                {
                        System.out.println("General");
                }

                System.out.println("Main Ended...");
        }
}
```

```java
package com.ravi.basic;

public class MultyCatch1
{
        public static void main(String[] args)
        {
                System.out.println("Main method started!!!");
                try
                {
                  String str1 = null;
                  System.out.println(str1.toUpperCase()); //NULL
```

```java
            String str2 = "Ravi";
            int x = Integer.parseInt(str2);
            System.out.println("Number is :"+x);
        }
        catch(NumberFormatException  | NullPointerException  e)
        {
          if(e instanceof NumberFormatException)
          {
                System.err.println("Number is not in a proper format");
          }
          else if(e instanceof NullPointerException)
          {
                System.err.println("ref variable is pointing to null");
          }
        }

        System.out.println("Main method ended!!");
    }

}
```

----
finally block : [100% guarantee for Exceution]
----
finally is a block which is meant for Resource handling purposes.

According to Software Engineering, the resources are memory creation, buffer creation, opening of a database, working with files, working with network resourses and so on.

Whenever the control will enter inside the try block always the finally block would be executed.

We should write all the closing statements inside the finally block because irrespective of exception finally block will be executed every time.

If we use the combination of try and finally then only the resources will be handled but not the execption, on the other hand if we use try-catch and finally then execption and resourses both will be handled.

package com.ravi.basic;

public class FinallyBlock
{
        public static void main(String[] args)
        {

```
                System.out.println("Main method started");

                try
                {
                        System.out.println(10/0);
                }
                finally
                {
                        System.out.println("Finally Block");
                }

                System.out.println("Main method ended");
        }

}
```

Note : Here we have an exception in the try block but still finally block will be executed.

```
package com.ravi.basic;

public class FinallyWithCatch
{
        public static void main(String[] args)
        {
                try
                {
                   int []x = new int[-2];
                   x[0] = 12;
                   x[1] = 15;
                   System.out.println(x[0]+" : "+x[1]);

                }
                catch(NegativeArraySizeException e)
                {
                        System.err.println("Array Size is in negative value...");

                }
                finally
                {
                   System.out.println("Resources will be handled here!!");
                }
                System.out.println("Main method ended!!!");
        }
```

}

In the above program exception and resourses both are handled because we have a combination of try-catch and finally.

Note :- In the try block if we write System.exit(0) and if this line is executed then finally block will not be executed.

Limitation of finally block :

The following are the limitation of finally block :

1) In order to close the resourses, user is responsible to write finally block manually.

2) Due to finally block the length of the program will be increased.

3) In order to close the resourses inside the finally block, we need to declare the resourses outside of try block.

package com.ravi.basic;

import java.util.InputMismatchException;
import java.util.Scanner;

public class FinallyLimitation
{
        public static void main(String[] args)
        {
          Scanner sc = null;
      try
      {
          sc = new Scanner(System.in);
          System.out.println("Enter your Marks :");
          int marks = sc.nextInt();
          System.out.println("Marks is :"+marks);
      }
      catch(InputMismatchException e)
      {
          System.err.println("Input is invalid");
      }
      finally
      {

```
            System.out.println("Finally block");
            sc.close();
    }


        }
}
```
----

try with resources :

To avoid all the limitation of finally block, Java software people introduced a separate concept i.e
try with resources from java 1.7 onwards.

Case 1:

```
try(resource1 ; resource2)  //Only the resources will be handled
{
}
```

Case 2 :

```
//Resources and Exception both will be  handled
try(resource1 ; resource2)
{
}
catch(Exception e)
{
}
```

Case 3 :
----
try with resourses enhancement from java 9v

```
Resourse r1 = new Resourse();
Resourse r2 = new Resourse();

try(r1; r2)
{
}
catch(Exception e)
{
}
```

There is a predefined interface available in java.lang package called AutoCloseable which contains predefined abstract method i.e close() which throws Exception.

There is another predefined interface available in java.io package called Closeable, this Closeable interface is the sub interface for AutoCloseable interface.

```
public interface java.lang.AutoCloseable
{
   public abstract void close() throws Exception;
}
public interface java.io.Closeable extends java.lang.AutoCloseable
{
   void close() throws IOException;
}
```

Whenever we pass any resourse class object as part of try with resources as a parameter then that class must implements either Closeable or AutoCloseable interface so, try with resourses will automatically call the respective class
close() method even an exception is encountered in the try block.

```
ResourceClass rc = new ResourceClass();
try(rc)
{
}
catch(Exception e)
{

}
```

This ResourceClass must implements either Closeable or AutoCloseable interface so, try block will automatically call the close() method as well as try block will get the guarantee of close() method support in the respective class.

The following program explains how try block is invoking the close() method available in DatabaseResourse class and FileResourse class.

3 files :

DatabaseResourse.java
----
package com.ravi.try_with_resourses;

```java
public class DatabaseResourse implements AutoCloseable
{
        @Override
        public void close() throws Exception
        {
                System.out.println("Database resourse is closed");
        }

}
```

FileResourse.java

```java
package com.ravi.try_with_resourses;

import java.io.Closeable;
import java.io.IOException;

public class FileResourse implements Closeable
{
        @Override
        public void close() throws IOException
        {
                System.out.println("File resourse closed successfully");
        }

}
```

TryWithResoursesDemo.java

```java
package com.ravi.try_with_resourses;

public class TryWithResoursesDemo {

        public static void main(String[] args) throws Exception
        {
                DatabaseResourse dr = new DatabaseResourse();
                FileResourse fr = new FileResourse();

                try(dr; fr)
                {
                        System.out.println(10/0);
                }
```

```java
                catch(ArithmeticException e)
                {
                        System.err.println("Don't put zero here");
                }

        }

}
----------------------------------------------------------------
//Program to close Scanner class automatically using try with resourses

package com.ravi.try_with_resourses;

import java.util.InputMismatchException;
import java.util.Scanner;

public class ScannerAutoClose {

        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);

                try(sc)
                {
                        System.out.println("Enter your Roll Number :");
                        int roll = sc.nextInt();
                        System.out.println("Your Roll number is :"+roll);
                }
                catch(InputMismatchException e)
                {
                        System.err.println("Input is not in a proper format");
                }

                System.out.println("Exception and resourses both are handled");
        }

}
```

Note :- Scanner class internally implementing Closeable interface so it is providing auto closing facility from java 1.7, as a user we need to pass the reference of Scanner class inside try with resources try()

Whenver we write try with resourses then automatically compiler will generate finally block internally to close the resourses automatically.

Nested try block :

If we write a try block inside another try block then it is called Nested try block.

```
try  //Outer try
{
  statement1;
    try  //Inner try
    {
      statement2;
    }
    catch(Exception e) //Inner catch
    {
    }
}
catch(Exception e) //Outer Catch
{
}
```

The execution of inner try block depends upon outer try block that means if we have an exception in the Outer try block then inner try block will not be executed.

```
package com.ravi.basic;

public class NestedTryBlock
{
        public static void main(String[] args)
        {
            try  //outer try
            {
                    String x = "null";
                    System.out.println("It's length is :"+x.length());

                        try  //inner try
                        {
                                String y = "NIT";
                                int z = Integer.parseInt(y);
                                System.out.println("z value is :"+z);
```

```
                }
                catch(NumberFormatException e)
                {
                        System.err.println("Number is not in a proper format");
                }
        }
        catch(NullPointerException e)
        {
                System.err.println("Null pointer Problem");
        }
    }
}
```

06-12-2024

Writing try-catch inside catch block :

We can write try-catch inside catch block but this try-catch block will be exceuted if the catch
block will executed that means if we have an exception in the try block.

```
package com.ravi.basic;

import java.util.InputMismatchException;
import java.util.Scanner;

public class TryWithCatchInsideCatch
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);

                try(sc )
                {
                        System.out.print("Enter your Roll number :");
                        int roll = sc.nextInt();
                        System.out.println("Your Roll is :"+roll);

                }
                catch(InputMismatchException e)
                {
                        System.err.println("Provide Valid input!!");

                        try
```

```java
        {
                System.out.println(10/0);
        }
        catch(ArithmeticException e1)
        {
          System.err.println("Divide by zero problem");
        }

}
finally
{
        try
        {
                throw new ArrayIndexOutOfBoundsException("Array is out of
bounds");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
                System.err.println("Array is out of Bounds");
        }
}
}

}
```

Note : inside finally block we can write try catch block.

----

try-catch with return statement

If we write try-catch block inside a method and that method is returning some value then we should write return statement in both the places i.e inside the try block as well as inside the catch block.

We can also write return statement inside the finally block only, if the finally block is present. After this return statement we cannot write any kind of statement. (Unrechable)

Always finally block return statement having more priority then try-catch return statement.

```java
package com.ravi.advanced;
public class ReturnExample
{
    public static void main(String[] args)
    {
      System.out.println(methodReturningValue());
```

```java
        }


        public static int methodReturningValue()
    {
      try
      {
        System.out.println("Try block");
        return 10/0;
      }
      catch (Exception e)
      {
         System.out.println("catch block");
        return 20;  //return statement is compulsory
      }

     // System.out.println("Unreachable code");

    }
}
```

```java
package com.ravi.advanced;

public class ReturnExample1 {

        public static void main(String[] args)
        {
                System.out.println(m1());
        }

        @SuppressWarnings("finally")
        public static int m1()
        {
                try
                {
                        System.out.println("Inside try");
                        return 100;
                }
                catch(Exception e)
                {
                        System.out.println("Inside Catch");
                        return 200;
                }
```

```
                finally
                {
                        System.out.println("Inside finally");
                        return 300;
                }

                // System.out.println("....");   Unreachable line
        }
}
```

Initialization of a variable in try and catch :

A local variable must be initialized inside try block as well as catch block OR at the time of declaration.

If we initialize inside the try block only then from catch block we cannot access local variable value, Here initialization is compulsory inside catch block.

```
package com.ravi.basic;

public class VariableInitialization
{
        public static void main(String[] args)
        {
          int x;
          try
          {
                  x = 100;
                  System.out.println(x);
          }
          catch(Exception e)
          {
                   x = 200;
                   System.out.println(x);
          }
      System.out.println("Main completed!!!");
        }

}
```

***Difference between Checked Exception and Unchecked Exception :

Checked Exception :

A checked exception is a common exception that must be declared or handled by the application code where it is thrown, Here compiler takes very much care and wanted the clarity regarding the exception by saying that, by using this code you may face some problem at runtime and you did not report me how would you handle this situation at runtime are called Checked exception, so provide either try-catch or declare the method as throws.

Except RuntimeException, all the checked exceptions are directly sub class of java.lang.Exception OR Throwable.

Eg:

FileNotFoundException, IOException, InterruptedException,ClassNotFoundException, SQLException, CloneNotSupportedException, EOFException and so on

Unchecked Exception :-

An unchecked exception is rare and any exception that does not need to be declared or handled by the application code where it is thrown, here compiler does not take any care are called unchecked exception.

Unchecked exceptions are directly entertain by JVM because they are rarely occurred in java.

All the un-checked exceptions are sub class of RuntimeException as well as Error and all the sub classes of Error are considered as Unchecked Exception.

RuntimeException is also Unchecked Exception.

Eg:

ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, NumberFormatException, ClassCastException, ArrayStoreException and so on.

Some Bullet points regarding Checked and Unchecked :

Checked Exception :
----
  1) Common Exception
  2) Compiler takes care (Will not compile the code)
  3) Handling is compulsory (try-catch OR throws)
  4) Directly the sub class of java.lang.Exception OR Throwable

Unchecked Exception :

----

   1) Rare Exception
   2) Comiler will not take any care
   3) Handling is not Compulsory
   4) Sub class of RuntimeException OR Error

When to provide try-catch or declare the method as throws for Checked Exception :-
----
try-catch

We should provide try-catch if we want to handle the exception in the method where checked exception is encountered, as well as if we want to provide user-defined messages to the client.

throws :
----
throws keyword describes that the method might throw an Exception, It also might not. It is used only at the end of a method declaration to indicate what exceptions it supports OR what type of Exception it might throw which will be handled by JVM or caller method.

Note :- It is always better to use try catch so we can provide appropriate user defined messages to our client.

07-12-2024

Exception propagation [Propagation of Exception from Callee to Caller]

Whenever we call a method and if the the callee method contains any kind of exception (checked OR Unchecked) and if callee method doesn't contain any kind of exception handling mechanism (try-catch OR throws) then JVM will propagate the exception object to caller method for handling purpose. This is called Exception Propagation.

If the caller method also does not contain any exception handling mechanism then JVM will terminate the method from the stack frame hence the remaining part of the method(m1 method) will not be executed even if we handle the exception in another caller method like main.

If any of the the caller method does not contain any exception handling mechanism then exception will be handled by JVM, JVM has default exception handler which will provide the exception message and terminates the program abnormally.[07-DEC]

//Program that describes Exception propagation with Unchecked Exception

package com.ravi.exception_propagation_demo;

```java
public class ExceptionPropagationWithUnchecked
{
        public static void main(String[] args)
        {
                System.out.println("Main method started..."); //1
                try
                {
                        m1();
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Hanled in main");
                }
                System.out.println("Main method ended...");
        }

        public static void m1()
        {
                System.out.println("M1 method started..."); //2
                m2();
                System.out.println("M1 method ended...");
        }

        public static void m2()
        {
                System.out.println(10/0);  //3
        }
}
```

//Program that describes Exception propagation with Checked Exception

package com.ravi.exception_propagation_demo;

import java.io.IOException;

public class ExceptionPropagationWithChecked
{
        public static void main(String[] args)
        {
         System.out.println("Main method started...");
         try
         {

```
                m1();
        }
        catch(IOException e)
        {
                System.out.println("Handled in main");
        }
        System.out.println("Main method ended...");
    }


        public static void m1() throws IOException
        {
                System.out.println("M1 method started...");
                m2();
                System.out.println("M1 method ended...");
        }

        public static void m2() throws IOException
        {
            throw new IOException();
        }
}
```
----
*Why compiler takes very much care regarding the checked Exception ?

As we know Checked Exceptions are very common exception so in case of checked exception
"handling is compulsory" because checked Exception depends upon other resources as shown
below.

IOException (we are depending upon System Keyboard OR Files)
FileNotFoundException(We are depending upon the file)
InterruptedException (Thread related problem)
ClassNotFoundException (class related problem)
SQLException (SQL related or database related problem)
CloneNotSupportedException (Object is the resourse)
EOFException(We are depending upon the file)

Types of exception in java :

Exception can be divided into two types :

1) Predefined Exception OR Built-in Exception

2) Userdefined Exception OR Custom Exception

Predefined Exception :-

The Exceptions which are already defined by Java software people for some specific purposes are called predefined Exception or Built-in exception.

Ex :

IOException, ArithmeticException and so on

Userdefined Exception :-

The exceptions which are defined by user according to their own use and requirement are called User-defined Exception.

Ex:-

InvalidAgeException, GreaterMarksException

How to develop User-defined Exceptions :

As a developer we can develop user-defined checked and user-defined unchecked exception.

If we want to develop checked exception then our user-defined class must extends from java.lang.Exception, on the other hand if we want to develop un-checked exception then our user-defined class must extends from java.lang.RuntimeException.

In the user-defined exception class we should write No argument constructor(in case if we don't want to pass any error message) and we should write parameterized constructor with String errorMessage as a parameter (in case if we want to pass any error message) with super keyword.

In order to throw the exception object explicitly we should use throw keyword as well as our user-defined class object must be of Throwable type.

```
class Test extends Throwable
{
        public Test(String errorMessage)
        {
          super(errorMessage);
        }
}

public class CustomDemo
```

```
{
        public static void main(String[] args)
        {
                try
                {
                        throw new Test("Test Problem");
                }
                catch(Throwable e)
                {
                }

        }

}
```

Here Test class is extending from Throwable class so it is throwable type of Object hence we can use throw keyword and it is also Checked Exception so handling is compulsory.

//Program to develop custom checked Exception :
----
package com.ravi.custom_exception;

import java.util.Scanner;

```
@SuppressWarnings("serial")
class InvalidAgeException extends Exception
{
        public InvalidAgeException()
        {
        }

        public InvalidAgeException(String errorMessage)
        {
                super(errorMessage);
        }
}

public class CustomCheckedException
{
        public static void main(String[] args)
        {
```

```java
        try
        {
                        validateAge();
        }
        catch(InvalidAgeException e)
        {
                System.out.println(e);
        }

        }

        public static void validateAge() throws InvalidAgeException
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter your Age :");
                int age = sc.nextInt();

                if(age<18)
                {
                        throw new InvalidAgeException("Age is Invalid");
                }
                else
                {
                        System.out.println("You are eligible for Voting");
                }
        }

}
```

09-12-2024

WAP to develop custom Unchecked Exception :

```java
package com.ravi.custom_exception;

import java.util.Scanner;

@SuppressWarnings("serial")
class GreaterMarksException extends RuntimeException
{
        public GreaterMarksException()
        {
        }
```

```java
        public GreaterMarksException(String errorMessage)
        {
                super(errorMessage);
        }
}


public class CustomUnchecked
{
        public static void main(String[] args)
        {
                validateStudentMarks();
    }


        public static void validateStudentMarks()
        {
                Scanner sc = new Scanner(System.in);
                try(sc)
                {
                        System.out.println("Enter your marks :");
                        int marks = sc.nextInt();

                        if(marks > 100)
                        {
                                throw new GreaterMarksException("Invalid Marks");
                        }
                        else
                        {
                                System.out.println("Your Marks is :"+marks);
                        }
                }
                catch(GreaterMarksException e)
                {
                        e.printStackTrace();
                        System.out.println("..........");
                        System.out.println(e.getMessage());
                }

        }

}
```

** What is the difference between throw and throws :
----
throw [THROWING THE EXCEPTION OBJCET EXPLICITLY.]

We should use throw keyword to throw the exception object explicitly, In case of try block, try block is responsible to create the exception object with JVM as well as throw the exception object to the nearest catch block.
but if a developer wants to throw exception object explicitly then we use throw keyword.

        throw new ArithmeticException();
            throw new LowBalanceException();

after using throw keyword the control will transfer to the nearest catch block so after throw keyword statement, the remaining statements are un-reachable.

throws :-
----
throws keyword describes that the method might throw an Exception, It also might not. It is used only at the end of a
method declaration to indicate what exceptions it supports OR what type of Exception it might throw at runtime.

It is used to skip from the current situation so now the execption will be propagated to the caller method OR JVM for
handling purpose.

It is mainly used to work with Checked Exception.

Some Basic rules we should follow while dealing with Checked Exception :

a) If the try block does not throw any checked exception then in the corresponding catch block we can't handle checked exception.It will generate compilation error i.e "exception never thrown from the corresponding try statement"

Example :-

public class Test
{
        public static void main(String[] args)
        {
                try
                {
                //try block is not throwing checked exception

```
                //i.e. InterruptedException
        }
        catch (InterruptedException e) //error
        {
        }

    }

}
```

Note :- The above rule is not applicable for Unchecked Exception

```
    try
        {

        }
        catch(ArithmeticException e)  //Valid
        {
                e.printStackTrace();
        }
```
----

b) If the try block does not throw any exception then in the corresponding catch block we can write Exception OR Throwable because both are the super classes for all types of Exception whether it is checked or unchecked.

```
package com.ravi.method_related_rule;

import java.io.EOFException;
import java.io.FileNotFoundException;

public class CatchingWithSuperClass
{
    public static void main(String[] args)
    {
            try
            {

            }
            catch(Exception e)  //Exception and Throwable both are allowed
            {
        e.printStackTrace();
            }
```

```
        }

}
-------------------------------------------------------------
```

c) At the time of method overriding if the super class method does not reporting or throwing checked exception then the overridden method of sub class not allowed to throw checked exception otherwise it will generate compilation error but overridden method can throw Unchecked Exception.

```java
package com.ravi.method_related_rule;

class Super
{
        public void show()
        {
                System.out.println("Super class method not throwing checked Exception");
        }
}
class Sub extends Super
{
        @Override
        public void show()  //throws InterruptedException
        {
                System.out.println("Sub class method should not throw checked Exception");
        }
}

public class MethodOverridingWithChecked {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

        }

}
```
```
-------------------------------------------------------------
```

d) If the super class method declare with throws keyword to throw a checked exception, then at the time of method overriding, sub class method may or may not use throws keyword.
          If the Overridden method is also using throws
 keyword to throw checked exception then it must be either same exception class or sub class, it should not be super class as well as we can't add more exceptions in the overridden method.

```java
package com.ravi.method_related_rule;

import java.io.FileNotFoundException;
import java.io.IOException;

class Base
{
        public void show() throws FileNotFoundException
        {
                System.out.println("Super class method ");
        }
}
class Derived extends Base
{
        public void show() //throws IOException
        {
                System.out.println("Sub class method ");
        }
}

public class MethodOverridingWithThrows
{
        public static void main(String[] args)
        {
          System.out.println("Overridden method may or may not throw checked exception but if
it is throwing then must be same or sub class");
        }

}
```
----
 e) Just like return keyword, we can't use throw keyword inside static and non static block to throw an exception because all initializers must be executed normally.

We can use throw keyword in the protection of try-catch so the code will be executed normally.

```java
class Super
{

        {
                System.out.println("NSB");
            try
            {
                        throw new ArithmeticException();
```

```
            }
            catch (ArithmeticException e)
            {
                        System.out.println("Handled");
            }
        }
}

public class ExceptionDemo
{
        public static void main(String[] args)
        {
                new Super();
        }
}
```
----
f) If we call any method and if the method throws java.lang.Exception OR java.lang.Throwable then handling is compulsory at caller Method otherwise it will generate compilation error because Exception and Throwable both are representing checked and Unchecked.

```
public class ExceptionDemo
{
        public static void main(String[] args)
        {
                m1();  //error
        }

        public static void m1() throws Throwable //OR Exception
        {
        }
}
```
==
Online Classes :

Serialization and De-Serialization
enum

Offline Classes :

Multithreading.
Collections Framework(Collection + Generics + Concurrent
                Collection + Stream API)

Multithreading :

Uniprocessing :-

In uniprocessing, only one process can occupy the memory So the
major drawbacks are

1) Memory is westage
2) Resources are westage
3) Cpu is idle

To avoid the above said problem, multitasking is introduced.

In multitasking multiple tasks can concurrently work with CPU so, our task will be completed as
soon as possible.

Multitasking is further divided into two categories.

a) Process based Multitasking [Diagram : 10th Dec]
b) Thread based Multitasking [Diagram : 10th Dec]

Process based Multitasking :

If a CPU is switching from one subtask(Thread) of one process to another subtask of another
process then it is called Process based Multitasking.

Thread based Multitasking :

If a CPU is switching from one subtask(Thread) to another subtask within the same process
then it is called Thread based Multitasking.

What is Thread in java ?

A thread is light weight process and it is the basic unit of CPU which can run concurrently with
another thread within the same context (process).

It is well known for independent execution. The main purpose of multithreading to boost the
execution sequence.

A thread can run with another thread concurrently within the same process so our task will be

completed as soon as possible.

In java whenever we define main method then JVM internally creates a thread called main thread under main group.

Program that describes that main is a Thread :

Whenever we define main method then JVM will create main thread internally under main group, the purpose of this main thread to execute the entire main method code.

In java there is a predefined class called Thread available in java.lang package, this class contains a predefined static factory method currentThread() which will provide currently executing Thread Object.

Thread t = Thread.currentThread(); //static Factory Method

Thread class has provided predefined method getName() to get the name of the Thread.
            public final String getName();

package com.ravi.mt;

public class MainThread {

        public static void main(String[] args)
        {
                //Thread obj = Thread.currentThread();
                //System.out.println("Current thread name is :"+obj.getName());

                        //OR

                String name = Thread.currentThread().getName();
                System.out.println("Current thread name is :"+name);

        }

}

How to create user-defined thread ?

We can create user-defined thread by using the following two packages

  1) By using java.lang package [JDK 1.0]
  2) By using java.util.concurrent sub package [JDK 1.5]

Creating user-defind Thread by using java.lang package :

By using java.lang package we can create user-defined thread by using any one of the following two approaches :

 1) By extending java.lang.Thread class
 2) By implementing java.lang.Runnable interface

 Note :- Thread is a predefined class available in java.lang package where as Runnable is a predefined interface available in java.lang Package.

 Creating user-defined Thread by using extending Thread class :

```java
 package com.ravi.mt;

class UserThread extends Thread
{
        @Override
        public void run()
        {
                System.out.println("My user thread is running in a separate stack");
        }
}

public class MainThread {

        public static void main(String[] args)
        {
                System.out.println("Main thread started");

                UserThread ut = new UserThread();
                ut.start();

                System.out.println(10/0);

                System.out.println("Main thread ended");

        }

}
```

In the above program, we have two threads, main thread which is responsible to execute main method and Thread-0 thread which is responsible to execute run() method. [10-DEC-24]

In entire Multithreading concept start() is the only method which is responsible to create a new thread.
----
public synchronized void start() :
----
start() is a predefined non static method of Thread class which internally performs the following two tasks :

1) It will make a request to the O.S to assign a new thread for concurrent execution.

2) It will implicitly call run() method on the current object.

----
public final boolean isAlive() :-

It is a predefined non static method of Thread class through which we can find out whether a thread has started or not ?

As we know a new thread is created/started after calling start() method so if we use isAlive() method before start() method, it will return false but if the same isAlive() method if we invoke after the start() method, it will return true.

We can't restart a thread in java if we try to restart then It will generate an exception i.e java.lang.IllegalThreadStateException

IsAlive.java
----
package com.ravi.basic;

class Foo extends Thread
{
        @Override
        public void run()
        {
                System.out.println("Child thread is running...");
                System.out.println("It is running with separate stack");
        }
}

```java
public class IsAlive
{
        public static void main(String[] args)
        {
                System.out.println("Main Thread started...");

                Foo f1 = new Foo();
                System.out.println("Is Thread alive : "+f1.isAlive());
                f1.start();
                System.out.println("Thread is alive or not : "+f1.isAlive());

                f1.start();  //java.lang.IllegralThreadStateException

                System.out.println("Main Thread ended...");


        }
}
```

```java
package com.ravi.basic;

class Stuff extends Thread
{
        @Override
        public void run()
        {
                System.out.println("Child Thread is Running!!!!");
        }
}
public class ExceptionDemo
{
        public static void main(String[] args)
        {
                System.out.println("Main Thread Started");

                Stuff s1 = new Stuff();
                Stuff s2 = new Stuff();

                s1.start();
                s2.start();

                System.out.println(10/0);
```

```
                System.out.println("Main Thread Ended");
        }

}
```

Note :- Here main thread is interrupted due to AE but still child thread will be executed because child threads are executing with separate Stack

----

```
package com.ravi.basic;

class Sample extends Thread
{
        @Override
        public void run()
        {
                String name = Thread.currentThread().getName();

                for(int i=1; i<=10; i++)
                {
                        System.out.println(i+" by "+name+" thread");
                }


        }

}

public class ThreadLoop
{
        public static void main(String[] args)
        {
            Sample s1 = new Sample();
            s1.start();

        String name = Thread.currentThread().getName();

                for(int i=1; i<=10; i++)
                {
                        System.out.println(i+" by "+name+" Thread");
                }

                int x = 1;
                do
                {
```

```
                    System.out.println("Enjoy Multithreading by "+name+ " Thread");


                    x++;
          }
          while(x<=10);


     }
}
```

Note : Here processor is frequently switching from main thread to Thread-0 thread so output is un-predicatable

----
How to set and get the name of the Thread :

How to set and get the name of the Thread :

Whenever we create a userdefined Thread in java then by default JVM assigns the name of thread is Thread-0, Thread-1, Thread-2 and so on.

If a user wants to assign some user defined name of the Thread, then Thread class has provided a predefined method called setName(String name) to set the name of the Thread.

On the other hand we want to get the name of the Thread then Thread class has provided a predefined method called getName().

public final void setName(String name) //setter

public final String getName() //getter

```
package com.ravi.basic;
class DoStuff extends Thread
{
       @Override
       public void run()
       {
              String name = Thread.currentThread().getName();
              System.out.println(name +" thread is running Here!!!!");
       }
}
public class ThreadName
{
       public static void main(String[] args)
```

```
        {
                DoStuff t1 = new DoStuff();
                DoStuff t2 = new DoStuff();

                t1.start();
                t2.start();


        System.out.println(Thread.currentThread().getName()+" thread is running.....");
        }
}
```

Note :- If we don't provide our user-defined name for the thread then by default the name would be Thread-0, Thread-1, Thread-2 and so on.

```
package com.ravi.basic;
class Demo extends Thread
{
        @Override
        public void run()
        {
                String name = Thread.currentThread().getName();
                System.out.println("Running Thread name is :"+name);
        }
}
public class ThreadName1
{
        public static void main(String[] args)
        {
          Thread t =  Thread.currentThread();
          t.setName("Parent"); //Changing the name of the main thread

           Demo d1 = new Demo();
           Demo d2 = new Demo();


           d1.setName("Child1");
           d2.setName("Child2");
           d1.start();  d2.start();



           String name = Thread.currentThread().getName();
```

```java
            System.out.println(name + " Thread is running Here..");
        }
}
-------------------------------------------------------------
package com.ravi.basic;

import java.util.InputMismatchException;
import java.util.Scanner;

class BatchAssignment extends Thread
{
        @Override
        public void run()
        {
                String name = Thread.currentThread().getName();

                if(name !=null && name.equalsIgnoreCase("Placement"))
                {
                        this.placementBatch();
                }
                else if(name !=null && name.equalsIgnoreCase("Regular"))
                {
                        this.regularBatch();
                }
                else
                {
                        throw new NullPointerException("Name can't be null");
                }
        }

        public void placementBatch()
        {
                System.out.println("I am a placement batch student.");
        }

        public void regularBatch()
        {
                System.out.println("I am a Regular batch student.");
        }
}


public class ThreadName2
```

```java
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                try(sc)
                {
                        System.out.print("Enter your Batch Title :");
                        String title = sc.next();

                        BatchAssignment b = new BatchAssignment();
                        b.setName(title);

                        b.start();
                }
                catch(InputMismatchException e)
                {
                        System.out.println("Invalid Input");
                }


        }

}
```

Thread.sleep(long millisecond) :

If we want to put a thread into temporarly waiting state then we should use sleep() method.

The waiting time of the Thread depends upon the time specified by the user in millisecond as parameter to sleep() method.

It is a static method of Thread class.

Thread.sleep(1000); //Thread will wait for 1 second

It is throwing a checked Exception i.e InterruptedException because there may be chance that this sleeping thread may be interrupted by a thread so provide either try-catch or declare the method as throws.
----
package com.ravi.basic;

class Sleep extends Thread
{

```java
    @Override
    public void run()
    {

        for(int i=1; i<=10; i++)
        {
                System.out.println("i value is :"+i);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.err.println("Thread interrupted "+e);
                }
        }
    }

}
public class SleepDemo
{
        public static void main(String[] args)
        {
                Sleep s = new Sleep();
                s.start();

        }
}
```

```java
package com.ravi.basic;

class MyTest extends Thread
{

        @Override
        public void run()
        {
                System.out.println("Child Thread id is :"+Thread.currentThread().getId());

                for(int i=1; i<=5; i++)
                {
                        System.out.println("i value is :"+i); // 11  22  33  44
                        try
```

```java
                    {
                            Thread.sleep(1000);
                    }
                    catch(InterruptedException e)
                    {
                            System.err.println("Thread has Interrupted");
                    }
            }
        }
}
public class SleepDemo1
{
        public static void main(String[] args)
        {
                System.out.println("Main Thread id is :"+Thread.currentThread().getId());  //1

                MyTest m1 = new MyTest();
                MyTest m2 = new MyTest();

                m1.start();
                m2.start();
        }
}
```

```java
package com.ravi.basic;

class MyThread1 extends Thread
{
        @Override
        public void run()
        {
                System.out.println("Child Thread is running");
        }
}

public class SleepDemo2 {

        public static void main(String[] args) throws InterruptedException
        {
                MyThread1 m1 = new MyThread1();
                m1.start();

                m1.sleep(2000); //current thread is main
```

System.out.println("Main Thread is Running");

        }

}

Note : Here main thread will go into the sleeping state.

Assignment :

Thread.sleep(long millis, int nanos)

Basic Thread life cycle :

As we know a thread is well known for Independent execution and it contains a life cycle which internally contains 5 states (Phases).

During the life cycle of a thread, It can pass from thses 5 states. At a time a thread can reside to only one state of the given 5 states.

1) NEW State (Born state)

2) RUNNABLE state (Ready to Run state) [Thread Pool]

3) RUNNING state

4) WAITING / BLOCKED state

5) EXIT/Dead state

New State :-

Whenever we create a thread instance(Thread Object) a thread comes to new state OR born state. New state does not mean that the Thread has started yet only the object or instance of Thread has been created.

Runnable state :-

Whenever we call start() method on thread object, A thread moves to Runnable state i.e Ready to run state. Here Thread schedular is responsible to select/pick a particular Thread from

Runnable state and sending that particular thread to Running state for execution.

Running state :-

If a thread is in Running state that means the thread is executing its own run() method in a separate stack Memory.

From Running state a thread can move to waiting state either by an order of thread schedular or user has written some method(wait(), join() or sleep()) to put the thread into temporarly waiting state.

From Running state the Thread may also move to Runnable state directly, if user has written Thread.yield() method explicitly.

Waiting state :-

A thread is in waiting state means it is waiting for it's time period to complete. Once the time period will be completed then it will re-enter inside the Runnable state to complete its remaining task.

Dead or Exit :

Once a thread has successfully completed its run method in the corresponding stack then the thread will move to dead state. Please remember once a thread is dead we can't restart a thread in java.

IQ :- If we write Thread.sleep(1000) then exactly after 1 sec the Thread will re-start?

Ans :- No, We can't say that the Thread will directly move from waiting state to Running state.

The Thread will definetly wait for 1 sec in the waiting mode and then again it will re-enter into Runnable state which is control by Thread Schedular so we can't say that the Thread will re-start just after 1 sec.
----
join() method of Thread class :

The main purpose of join() method to put the current thread into waiting state until the other thread finish its execution.

Here the currently executing thread stops its execution and the thread goes into the waiting state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state.

It also throws checked exception i.e InterruptedException so better to use try catch or declare the method as throws.

It is a non static method so we can call this method with the help of Thread object reference.

```java
package com.ravi.basic;

class Join extends Thread
{
        @Override
        public void run()
        {
                for(int i=1; i<=5; i++)
                {
                        System.out.println("i value is :"+i);
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException e)
                        {
                                e.printStackTrace();
                        }
                }
        }
}

public class JoinDemo
{
        public static void main(String[] args) throws InterruptedException
        {
                System.out.println("Main Thread started");

                Join j1 = new Join();
                Join j2 = new Join();
                Join j3 = new Join();


                j1.start();

                j1.join(); //Here main thread will go to waiting state

                j2.start();
```

```
                j3.start();


                System.out.println("Main Thread ended");
        }

}
------------------------------------------------------------
13-12-2024
-----------
package com.ravi.basic;

class Alpha extends Thread
{
        @Override
        public void run()
        {
                Thread t = Thread.currentThread();
                String name = t.getName();    //Alpha_Thread is current thread

                Beta b1 = new Beta();
                b1.setName("Beta_Thread");
        b1.start();
        try
        {
                        b1.join(); //Alpha thread is waiting 4 Beta Thread to complete

                        System.out.println("Alpha thread re-started");
                }
        catch (InterruptedException e)
        {
                        e.printStackTrace();
                }

                for(int i=1; i<=10; i++)
                {
                        System.out.println(i+" by "+name);
                }

        }
}

public class JoinDemo2
```

```java
{
        public static void main(String[] args)
        {
                Alpha a1 = new Alpha();
                a1.setName("Alpha_Thread");
                a1.start();
        }
}

class Beta extends Thread
{
        @Override
        public void run()
        {
                Thread t = Thread.currentThread();
                String name = t.getName();

                for(int i=1; i<=20; i++)
                {
                        System.out.println(i+" by "+name);
                        try
                        {
                                Thread.sleep(500);
                        }
                        catch(InterruptedException e) {

                        }
                }
                System.out.println("Beta Thread Ended");
        }
}
```

Note : Alpha Thread will wait till the completion of Beta Thread.

```java
package com.ravi.basic;

public class JoinDemo1
{
        public static void main(String[] args) throws InterruptedException
        {
    System.out.println("Main Thread started");

    Thread t = Thread.currentThread();
```

```java
        for(int i=1; i<=10; i++)
        {
                System.out.println(i+" by "+t.getName());
        }

        t.join();  //Deadlock [Main thread is waiting 4 main thread to complete]

        System.out.println("Main Thread Ended");
            }
}
```

Here, It is a deadlock state because main thread is waiting for main thread to complete.

Assignment :

join(long millis)
join(long millis, long nanos)

Anonymous inner class by using Thread class :
----
 1) Anonymous inner class with reference variable :
 ----
package com.ravi.anonymous_thread;

```java
public class AnonymousThreadWithReference
{
        public static void main(String[] args)
        {
                Thread t1 = new Thread()
                {
                        @Override
                        public void run()
                        {
                                String name = Thread.currentThread().getName();
                                System.out.println("Anonymous Thread name is :"+name);
                        }

                };
                t1.start();
```

```
        }

}

2) Anonymous inner class without reference variable :
--------------------------------------------------------
package com.ravi.anonymous_thread;

public class AnonymousWithoutReference {

        public static void main(String[] args)
        {
                new Thread()
                {
                  @Override
                  public void run()
                  {
                          String name = Thread.currentThread().getName();
                          System.out.println("Running thread name is :"+name);
                  }
                }.start();

        }

}
------------------------------------------------------------
Assigning target by Runnable interface :[Loose Coupling]
--------------------------------------------------------
By using Runnable interface approach we can assign different target to our threads.
[13-DEC-24]

package com.ravi.basic;

class Ravi implements Runnable
{
        @Override
        public void run()
        {
                System.out.println("Thread is performing my task ");
        }
}
```

```java
public class RunnableDemo
{
    public static void main(String [] args)
    {
            System.out.println("Main");

        Thread t1 = new Thread(new Ravi());
        t1.start();

    }
}
```

Assign different target for different Threads :

```java
package com.ravi.basic;

class Ravi implements Runnable
{
        @Override
        public void run()
        {
                System.out.println("Thread is performing my task ");
        }
}

public class RunnableDemo
{
    public static void main(String [] args)
    {
            System.out.println("Main");

        Thread t1 = new Thread(new Ravi());
        t1.start();

    }
}
```

Thread class constructor :

We have total 10 constructors in the Thread class, The following are commonly used constructor in the Thread class

1) Thread t1 = new Thread();

2) Thread t2 = new Thread(String name);

3) Thread t3 = new Thread(Runnable target);

4) Thread t4 = new Thread(Runnable target, String name);

5) Thread t5 = new Thread(ThredGroup tg, String name);

6) Thread t6 = new Thread(ThredGroup tg, Runnable target);

7) Thread t7 = new Thread(ThredGroup tg, Runnable target, String name);

14-12-2024

Anonymous inner class by using Runnable interface :

Case 1 :
----
package com.ravi.anonymous_runnable;

public class AnonymousDemo1 {

```java
        public static void main(String[] args)
        {
                Runnable r1 = new Runnable()
                {
                        @Override
                        public void run()
                        {
                                String name = Thread.currentThread().getName();
                                System.out.println("Current Thread Name is :"+name);
                        }
                };

                Thread t1 = new Thread(r1);
                t1.start();

        }

}
```

Case 2:

```
-------
package com.ravi.anonymous_runnable;

public class AnonymousDemo2 {

        public static void main(String[] args)
        {
                Thread t1 = new Thread(new Runnable()
                {
                        @Override
                        public void run()
                        {
                                String name = Thread.currentThread().getName();
                                System.out.println("Current Thread Name is :"+name);


                        }
                });

                t1.start();

        }

}
```

Case 3 :

```
package com.ravi.anonymous_runnable;

public class AnonymousDemo3 {

        public static void main(String[] args)
        {
                Thread t1 = new Thread(()->
System.out.println(Thread.currentThread().getName()));
                t1.start();

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

                new Thread(()-> System.out.println(Thread.currentThread().getName())).start();

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

                new Thread(()->
```

System.out.println(Thread.currentThread().getName()),"Scott").start();

       }

}

Drawback of Multithreading :

Multithreading is very good to complete our task as soon as possible but in some situation, It provides some wrong data or wrong result.

In Data Race or Race condition, all the threads try to access the resource at the same time so the result may be corrupted.

In multithreading if we want to perform read operation and data is not updatable data then multithreading is good but if the data is updatable data (modifiable data) then multithreading may produce some wrong result or wrong data which is known as Data Race OR race condition as shown in the diagram.(14-DEC)

```java
package com.nit.testing;

class Customer implements Runnable
{
   private int availableSeat = 1;
   private int wantedSeat;  //1

        public Customer(int wantedSeat)
        {
         super();
         this.wantedSeat = wantedSeat;
   }

        @Override
        public synchronized void run()
        {
                String name = null;

                if(availableSeat >= wantedSeat)
                {
                        name = Thread.currentThread().getName();
                        System.out.println(wantedSeat+" seat is reserved for "+name);
                        availableSeat = availableSeat - wantedSeat;
                }
```

```java
                else
                {
                        name = Thread.currentThread().getName();
                        System.err.println("Sorry!!"+name+" seat is not available");
                }


        }

}

public class RailwayReservation {

        public static void main(String[] args) throws InterruptedException
        {
          Customer c1 = new Customer(1);

          Thread t1 = new Thread(c1,"Scott");
          Thread t2 = new Thread(c1,"Smith");

          t1.start();


          t2.start();

        }

}
```

```java
package com.ravi.multithreading_drawback;

class Customer
{
        private double availableBalance = 20000;
        private double withdrawAmount;

        public Customer(double withdrawAmount)
        {
                super();
                this.withdrawAmount = withdrawAmount;
        }

        public void withdraw()
```

```java
        {
                String name = null;

                if(withdrawAmount<= availableBalance)
                {
                        name = Thread.currentThread().getName();
                        System.out.println(withdrawAmount+ " Amount, withdraw by :"+name);
                        availableBalance = availableBalance - withdrawAmount;
                }
                else
                {
                        name = Thread.currentThread().getName();
                        System.err.println("Sorry!!"+name+" you have insufficient Balance");
                }
        }

}

public class BankApplication {

        public static void main(String[] args)
        {
                Customer obj = new Customer(20000);

                Runnable r1 = ()->
                {
                  obj.withdraw();
                };

                Thread t1 = new Thread(r1,"Scott");
                Thread t2 = new Thread(r1,"Smith");

                t1.start();

                t2.start();
        }

}
```

Note : Here both the thread will get the balance.

16-12-2024

***Synchronization :

In order to solve the problem of multithreading java software people has introduced synchronization concept.

In order to acheive sycnhronization in java we have a keyword called "synchronized".

It is a technique through which we can control multiple threads but accepting only one thread at all the time.

Synchronization allows only one thread to enter inside the synchronized area for a single object.

Synchronization can be divided into two categories :-

1) Method level synchronization

2) Block level synchronization

1) Method level synchronization :-

In method level synchronization, the entire method gets synchronized so all the thread will wait at method level and only one thread will enter inside the synchronized area as shown in the diagram.(14-DEC-24)

2) Block level synchronization

In block level synchronization the entire method does not get synchronized, only the part of the method gets synchronized so all the thread will enter inside the method but only one thread will enter inside the synchronized block as shown in the diagram (14-DEC-24)

Note :- In between method level synchronization and block level synchronization, block level synchronization is more preferable because all the threads can enter inside the method so only the PART OF THE METHOD GETS synchronized so only one thread will enter inside the synchronized block.

Note : Synchronized area is a restricated area, with
permission only a thread can enter inside synchronized area.

How synchronization logic controls multiple threads ?

Every Object has a lock(monitor) in java environment and this lock can be given to only one Thread at a time.

Actually this lock is available with each individual object provided by Object class.

The thread who acquires the lock from the object will enter inside the synchronized area, it will complete its task without any disturbance because at a time there will be only one thread inside the synchronized area(for single Object). *This is known as Thread-safety in java.

The thread which is inside the synchronized area, after completion of its task while going back will release the lock so the other threads (which are waiting outside for the lock) will get a chance to enter inside the synchronized area by again taking the lock from the object and submitting it to the synchronization mechanism.
This is how synchronization mechanism controls multiple Threads.

Note :- Synchronization logic can be done by senior programmers in the real time industry because due to poor synchronization there may be chance of getting deadlock.

Program on Method Level Synchronization :
----
package com.ravi.thread;

```java
class Table
{
        public synchronized void printTable(int num)
        {
                for(int i=1; i<=10; i++)
                {
                        System.out.println(num+" X "+i+" = "+(num*i));
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException e)
                        {
                                e.printStackTrace();
                        }
                }
                System.out.println("..................");
        }
}
public class MethodLevelSynchronization
{
        public static void main(String[] args)
        {
```

```java
                Table obj = new Table();   //Lock is created Here

                Thread t1 = new Thread()
                {
                        @Override
                        public void run()
                        {
                                obj.printTable(5);
                        }
                };

                Thread t2 = new Thread()
                {
                        @Override
                        public void run()
                        {
                                obj.printTable(10);
                        }
                };

                t1.start();  t2.start();
        }

}
```

//Program on block Level Synchronization :
----
package com.ravi.advanced;

//Block level synchronization

```java
class ThreadName
{
        public void printThreadName()
        {
          String name = Thread.currentThread().getName();
          System.out.println("Thread inside the method is :"+name);

                synchronized(this)  //synchronized Block
                {
                        for(int i=1; i<=9; i++)
                        {
                                System.out.println("i value is :"+i+" by :"+name);
```

```
                    }
                    System.out.println(".............................");
                }
        }
}
public class BlockSynchronization
{
        public static void main(String[] args)
        {
                ThreadName obj1 = new ThreadName(); //lock is created

                Runnable r1 = () -> obj1.printThreadName();

                Thread t1 = new Thread(r1,"Child1");
                Thread t2 = new Thread(r1,"Child2");
                t1.start(); t2.start();
        }
}
```

Limitation/Drawback of Object Level Synchronization :

From the given diagram it is clear that there is no interference between t1 and t2 thread
because they are passing throgh Object1 where as on the other hand there is no interferenec
even in between t3 and t4 threads because they are also passing through Object2 (another
object).

But there may be chance that with t1 Thread (object1), t3 or t4 thread can enter inside the
synchronized area at the same time, simillarly it is also possible that with t2 thread, t3 or t4
thread can enter inside the synchronized area so the conclusion is, synchronization mechanism
does not work with multiple Objects.(Diagram 16-DEC-24)

```
package com.ravi.advanced;
class PrintTable
{
        public synchronized void printTable(int n)
        {
          for(int i=1; i<=10; i++)
          {
                System.out.println(n+" X "+i+" = "+(n*i));
                try
                {
                        Thread.sleep(500);
                }
```

```java
                    catch(Exception e)
                    {
                    }
            }
            System.out.println(".......................");
        }
}

public class ProblemWithObjectLevelSynchronization
{
        public static void main(String[] args)
        {

                PrintTable pt1 = new PrintTable(); //lock1
                PrintTable pt2 = new PrintTable(); //lock2

                Thread t1 = new Thread()  //Anonymous inner class concept
                        {
                        @Override
                        public void run()
                        {
                            pt1.printTable(2);     //lock1
                        }
                        };

                Thread t2 = new Thread()
                        {
                        @Override
                        public void run()
                        {
                            pt1.printTable(3);     //lock1
                        }
                        };

                Thread t3 = new Thread()
                        {
                        @Override
                        public void run()
                        {
                            pt2.printTable(8);     //lock2
                        }
                        };
```

```
                    Thread t4 = new Thread()
                        {
                    @Override
                    public void run()
                    {
                        pt2.printTable(9); //lock2
                    }
                      };
                       t1.start();      t2.start();        t3.start();  t4.start();
        }
}
```

Note : So, It is clear that synchronization logic will not work with multiple Objects.

In order to resolve this synchronization issue, Static
Synchronization is introduced by Oracle Developer.

**Static Synchronization :

If we make our synchronized method as a static method then it is called static synchronization.

Here, To call static synchronized method, object is not required.

The thread will take the lock from class but not object because we can call the static method
with the help of class name.

Unlike Object, we cann't create multiple classes in the same package.

For synchronized block we can write the following code :

synchronized(ClassName.class)
{

}

//Program on static synchronization :
package com.ravi.advanced;
class MyTable
{
        public static synchronized void printTable(int n) //static synchronization
          {
            for(int i=1; i<=10; i++)
```

```java
        {
            try
            {
                Thread.sleep(100);
            }
            catch(InterruptedException e)
            {
                System.err.println("Thread is Interrupted...");
            }
            System.out.println(n+" X "+i+" = "+(n*i));
        }
        System.out.println("-----------------------");
    }
}
public class StaticSynchronization
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                MyTable.printTable(5);
            }
        };

        Thread t2 = new Thread()
        {
            @Override
            public void run()
            {
                MyTable.printTable(10);
            }
        };


        Runnable r3 = ()-> MyTable.printTable(15);
        Thread t3 = new Thread(r3);

        t1.start();
        t2.start();
        t3.start();
```

```
                    }
}
```

In between extends Thread and implements Runnable which one is better and why?

In between extends Thread and implements Runnable, implements Runnable is more better approach due to the following reasons.

1) In extend Thread class approach, We can't extend any other class further
   (Multiple Inheritance not possible by using class) but we can implement multiple interfaces. On the other hand in implements Runnable approach we can extend a single class as well as implement multiple interfaces.

2) In extends Thread class all the Thread class properties are available to sub class so sub class is heavy weight, the same thing is not available while implementing Runnable interface.

3) In extends Thread class approach we have same target for different Threads but by using implements we can provide different target for different Threads.

4) In extends Thread class approach we don't have Lambda expression support but by using Runnable interface we can implement Lambda.

Conclusion :
Implements Runnable Advantage
1) extend a single class
2) Light weight
3) Assigning different targets (Loose coupling)
3) Implemnting Lambda

Thread Priority :

It is possible in java to assign priority to a Thread. Thread class has provided two predefined methods setPriority(int newPriority) and getPriority() to set and get the priority of the thread respectively.

In java we can set the priority of the Thread in numbers from 1- 10 only where 1 is the minimum priority and 10 is the maximum priority.

Whenever we create a thread in java by default its priority would be 5 that is normal priority.

The user-defined thread created as a part of main thread will acquire the same priority of main Thread.

Thread class has also provided 3 final static variables which are as follows :-

public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY = 5;
public static final int MAX_PRIORITY = 10;

Note :- We can't set the priority of the Thread beyond the limit(1-10) so if we set the priority beyond the limit (1 to 10) then it will generate an exception java.lang.IllegalArgumentException.

```java
public class ThreadPriority {

        public static void main(String[] args)
        {
                //Default Priority of the Thread
                Thread t = Thread.currentThread();
                System.out.println("Main thread priority is :"+t.getPriority());

                Thread t1 = new Thread();
                System.out.println("User thread priority is :"+t1.getPriority());

        }

}
```

Note : By default every thread even main thread is having default priority i.e 5.

```java
class Priority extends Thread
{
        @Override
        public void run()
        {
                int priority = Thread.currentThread().getPriority();
                System.out.println("Child Thread priority is :"+priority);
        }
}

public class ThreadPriorityDemo1 {
```

```java
        public static void main(String[] args)
        {
                Thread t = Thread.currentThread();
                t.setPriority(9);

                Priority p1 = new Priority();
                p1.start();

                System.out.println("Main Thread priority is :"+t.getPriority());


        }

}
```

Note : The thread which are created under main Thread will
acquire main thread priority.

```java
class MyPriority extends Thread
{
        @Override
        public void run()
        {
                int count = 0;

                for(int i=1; i<=1000000; i++)
                {
            count++;
                }

                System.out.println("Thread name is :"+Thread.currentThread().getName());
                System.out.println("Thread Priority is :"+Thread.currentThread().getPriority());
        }
}

public class ThreadPriorityDemo2
{
        public static void main(String[] args)
        {
    MyPriority m1 = new MyPriority();
    m1.setPriority(Thread.MIN_PRIORITY);
```

```
        m1.setName("Last");

        MyPriority m2 = new MyPriority();
        m2.setPriority(Thread.MAX_PRIORITY);
        m2.setName("First");

        m1.start();  m2.start();

            }

}
```

Most of time the thread having highest priority will complete its task but we can't say that it will always complete its task first that means Thread schedular dominates over the priority of the Thread.

Thread.yield() :

It is a static method of Thread class.

It will send a notification to thread schedular to stop the currently executing Thread (In Running state) and provide a chance to Threads which are in Runnable state to enter inside the running state having same priority or higher priority than currently executing Thread.

Here The running Thread will directly move from Running state to Runnable state.

The Thread schedular may accept OR ignore this notification message given by currently executing Thread.

Here there is no guarantee that after using yield() method the running Thread will move to Runnable state and from Runnable state the thread can move to Running state.[That is the reason yield() method is not throwing InterruptedExecption]

If the thread which is in runnable state is having low priority than the current executing thread in Running state, then currently executing thread will continue its execution.

*It is mainly used to avoid the over-utilisation a CPU by the current Thread.

class MyThread implements Runnable
{
        @Override
```

```java
        public void run()
        {
                String name = Thread.currentThread().getName();

                for(int i=1; i<=10; i++)
                {
                        System.out.println("i value is :"+i+" by "+name);

                        if(name.equals("Child1"))
                        {
                                Thread.yield();
                        }
                }
        }
}

public class YieldDemo {

        public static void main(String[] args)
        {
                MyThread mt = new MyThread();

                Thread t1 = new Thread(mt, "Child1");
                Thread t2 = new Thread(mt, "Child2");

                t1.start();  t2.start();


        }

}
```

18-12-2024

** Inter Thread Communication(ITC) :

It is a mechanism to communicate or co-ordinate between two synchronized threads within the context to achieve a particular task.

In ITC we put a thread into wait mode by using wait() method and other thread will complete its corresponding task, after completion of the task it will call notify() method so the waiting thread will get a notification to complete its remaining task.

ITC can be implemented by the following method of Object class.

1) public final void wait() throws InterruptedException

2) public native final void notify()

3) public native final void notifyAll()

public final void wait() throws InterruptedException :-

It is a predefined non static method of Object class. We can use this method from synchronized area only otherwise we will get IllegalMonitorStateException.

It will put a thread into temporarly waiting state and it will release the Object lock, It will remain in the wait state till another thread provides a notification message on the same object, After getting the lock (not notification message), It will wake up and it will complete its remaining task.

public native final void notify() :-

It will wake up the single thread that is waiting on the same object.It will not release the lock , once synchronized area is completed then only lock will be released.

Once a waiting thread will get the notification from the another thraed using notify()/notifyAll() method then the waiting thread will move from Waiting state to Runnable state(Ready to run state)

public native final void notifyAll() :-

It will wake up all the threads which are waiting on the same object.It will not release the lock , once synchronized area is completed then only lock will be released.

*Note :- wait(), notify() and notifyAll() methods are defined in Object class but not in Thread class because these methods are related to lock(because we can use these methods from the synchronized area ONLY) and Object has a lock so, all these methods are defined inside Object class.

The following program explains we should use these methods from synchronized area only otherwise we will get java.lang.IllegalMonitorStateException.

package com.ravi.itc;

```
public class ITCDemo1
{
        public  static void main(String[] args) throws InterruptedException
        {
           Object obj = new Object();
           obj.wait();
        }
}
```

The following program explains, if we don't have proper
co-ordination between two threads then Output is un-predicatable.

```
package com.ravi.itc;

class Test extends Thread
{
        private int val = 0;

        @Override
        public void run()
        {
                for(int i=1; i<=10; i++)  //i =
                {
                        val = val + i;      //val = 1    3    6   10    15
                        try
                        {
                                Thread.sleep(100);
                        }
                        catch(InterruptedException e)
                        {
                                e.printStackTrace();
                        }
                }

        }

        public int getVal()
        {
                return this.val;
        }
}

public class ITCDemo2
```

```
{
        public static void main(String[] args) throws InterruptedException
        {
    System.out.println("Main Thread Started...");

    Test t1 = new Test();
    t1.start();

    Thread.sleep(200);

    System.out.println(t1.getVal());
        }

}
```

Note : In the above program, there is no co-ordination between
main thread and child thread so the value of val will change based on loop iteration so output is
un-predictable
----
package com.ravi.itc;

```
class Demo extends Thread
{
        private int val = 0;

        @Override
        public void run()
        {
                synchronized(this)
                {
                        for(int i=1; i<=10; i++)
                        {
                                val = val + i;
                        }
                        System.out.println("Completed My task, Sending u notification");
                        notify();
                }
        }

        public int getVal()
        {
                return this.val;
        }
}
```

```
}

public class ITCDemo3 {

        public static void main(String[] args) throws InterruptedException
        {
                Demo d1 = new Demo();
                d1.start();

                synchronized(d1)
                {
                        System.out.println("main thread is waiting Here :");
                        d1.wait();
                        System.out.println("Main thread got notification :");
                        System.out.println(d1.getVal());
                }


        }

}
```

Note : Here we have co-ordination between main thread and child thread so we will get predicatable output.

```
package com.ravi.itc;

class Customer
{
        private double balance = 10000;

        public synchronized void withdraw(double amount)
        {
                System.out.println("Withdraw is in progress..");

                if(amount> this.balance)
                {
                        System.out.println("Balance is low, waiting for deposit");
                        try
```

```java
                    {
                            wait();
                            System.out.println("Got Notification");
                    }
                    catch(InterruptedException e)
                    {
                            e.printStackTrace();
                    }
            }
            this.balance = this.balance - amount;
            System.out.println("Amount after withdraw is :"+this.balance);
        }

        public synchronized void deposit(double amount)
        {
            System.out.println("Deposit is in progress...");
            this.balance = this.balance + amount;
            System.out.println("Balance after deposit is :"+this.balance);
            System.out.println("Sending notification to withdraw amount");
            notify();
        }

}

public class ITCDemo4 {

        public static void main(String[] args)
        {
            Customer obj = new Customer();

            Thread son = new Thread()
            {
              @Override
              public void run()
              {
                    obj.withdraw(10000);
              }

            };
            son.start();

            Thread father = new Thread()
            {
```

```java
                @Override
                public void run()
                {
                        obj.deposit(10000);
                }
        };

        father.start();

    }

}
```

19-12-2024

```java
package com.ravi.itc;

class TicketSystem
{
    private int availableTickets = 5;   //availableTickets = 3

    public synchronized void bookTicket(int numberOfTickets) //numberOfTickets 4
    {
        while (availableTickets < numberOfTickets) //5 < 4
        {
          System.out.println("Not enough tickets available, Waiting for cancellation...");
           try
           {
               wait();
           }
           catch (InterruptedException e)
           {
               e.printStackTrace();
           }
        }
        availableTickets = availableTickets - numberOfTickets;

        System.out.println("Booked " + numberOfTickets + " ticket(s). Remaining tickets: " +
availableTickets);
        notify();

    }
```

```java
    public synchronized void cancelTicket(int numberOfTickets)//numberOfTickets 3
    {
        availableTickets = availableTickets + numberOfTickets;
        System.out.println("Canceled " + numberOfTickets + " ticket(s). Available tickets: " +
availableTickets);
        notify();
    }
}


public class ITCDemo5
{
    public static void main(String[] args)
    {
        TicketSystem ticketSystem = new TicketSystem(); //lock is available

        Thread bookingThread = new Thread()
        {
            @Override
            public void run()
            {
                int[] ticketsToBook = {2, 4, 4};

                for (int ticket : ticketsToBook) //ticket = 1
                {
                    ticketSystem.bookTicket(ticket);
                    try
                    {
                        Thread.sleep(1000); // give some time b/w booking
                    }
                    catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        };
        bookingThread.start();

        Thread cancellationThread = new Thread()
        {
            @Override
```

```java
        public void run()
        {
            int[] ticketsToCancel = {1, 3, 2};
            for (int ticket : ticketsToCancel) //ticket = 3
            {
                ticketSystem.cancelTicket(ticket);
                try
                {
                    Thread.sleep(1500);  // give some time b/w cancellation
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        }
    };
    cancellationThread.start();


    }
}


------------------------------------------------------------
package com.ravi.itc;

class Resource
{
    private boolean flag = false;

    public synchronized void waitMethod()
        {
                System.out.println("Wait");  //child1  wait  child1 is waiting Child1 notif


        while (!flag)
                {
        try
                {
        System.out.println(Thread.currentThread().getName() + " is waiting...");
        System.out.println(Thread.currentThread().getName()+" is Waiting for Notification");
        wait();
```

```java
            }
                catch (InterruptedException e)
                {
            e.printStackTrace();
        }
    }
    System.out.println(Thread.currentThread().getName() + " thread completed!!");
  }

  public synchronized void setMethod()
      {
                System.out.println("notifyAll");
                this.flag = true;
        System.out.println(Thread.currentThread().getName() + " has make flag value as a true");
        notifyAll(); // Notify all waiting threads that the signal is set
  }
}

public class ITCDemo6
{
   public static void main(String[] args)
                {


      Resource r1 = new Resource(); //lock is created

      Thread t1 = new Thread(() -> r1.waitMethod(), "Child1");
                Thread t2 = new Thread(() -> r1.waitMethod(), "Child2");
                Thread t3 = new Thread(() -> r1.waitMethod(), "Child3");

                t1.start();
      t2.start();
      t3.start();


                Thread setter = new Thread(() -> r1.setMethod(), "Setter_Thread");

                 try
                        {
                Thread.sleep(2000);
            }
                        catch (InterruptedException e)
                        {
```

```
                    e.printStackTrace();
               }

               setter.start();
     }
}
```

ThreadGroup :

It is a predefined class available in java.lang Package.

By using ThreadGroup class we can put 'n' number of threads into a single group to perform some common/different operation.

By using ThreadGroup class constructor, we can assign the name of group under which all the thread will be executed.

ThreadGroup tg = new ThreadGroup(String groupName);

ThreadGroup class has provided the following methods :

public String getName() : To get the name of the Group

pubic int activeCount() : How many threads are alive and running under that particular group.

Thread class has provided constructor to put the thread into particular group.

Thread t1 = new Thread(ThreadGroup tg, Runnable target, String name);

By using ThreadGroup class, multiple threads will be executed under single group.

```
package com.ravi.group;

class Foo implements Runnable
{
        @Override
        public void run()
        {
                String name = Thread.currentThread().getName();
        System.out.println(name+" is Enjoying Full Stack Developer under Batch 38");


        }
}
```

```java
public class ThreadGroupDemo1
{
  public static void main(String[] args) throws InterruptedException
  {

        ThreadGroup tg = new ThreadGroup("Batch 38");

        Thread t1 = new Thread(tg, new Foo(), "Scott");
        Thread t2 = new Thread(tg, new Foo(), "Smith");
        Thread t3 = new Thread(tg, new Foo(), "Alen");
        Thread t4 = new Thread(tg, new Foo(), "John");
        Thread t5 = new Thread(tg, new Foo(), "Martin");

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();

        //Thread.sleep(5000);

        System.out.println("How many threads are active under Batch 38 group
:"+tg.activeCount());

        System.out.println("Name of the the Group is :"+tg.getName());

  }
}
```

```java
package com.ravi.group;

public class ThreadGroupDemo2 {

        public static void main(String[] args)
        {
                Thread t = Thread.currentThread();
                System.out.println(t.toString());
        }

}
```

Output : Thread[#1main, 5, main]

Here first main is the name of the Thread, 5 is the priority and last main represents group name.

Whenever we define a main method then internally, main group is created and under this main group main thread is executed.

```java
package com.ravi.group;

class TatkalTicket implements Runnable
{
        @Override
        public void run()
        {
                String name = Thread.currentThread().getName();
                System.out.println("Tatkal ticket booked by :"+name);

        }

}

class PremiumTatkal implements Runnable
{
        @Override
        public void run()
        {
                String name = Thread.currentThread().getName();
                System.out.println("Premium Tatkal ticket booked by :"+name);

        }
}

public class ThreadGroupDemo3
{
        public static void main(String[] args)
        {
                ThreadGroup tg1 = new ThreadGroup("Tatkal Ticket");
                ThreadGroup tg2 = new ThreadGroup("Premium Tatkal");

                Thread t1 = new Thread(tg1,new TatkalTicket(), "Scott");
                Thread t2 = new Thread(tg2,new PremiumTatkal(), "Smith");
```

```
            t1.start(); t2.start();


      }


}
```

20-12-2024

Daemon Thread [Service Level Thread]

Daemon thread is a low- priority thread which is used to provide background maintenance.

The main purpose of of Daemon thread to provide services to the user thread.

JVM can't terminate the program till any of the non-daemon (user) thread is active, once all the user thread will be completed then JVM will automatically terminate all Daemon threads, which are running in the background to support user threads.

The example of Daemon thread is Garbage Collection thread, which is running in the background for memory management.

In order to make a thread as a Daemon thread , we should use setDaemon(true) which is a non static method Thread class.

```
public class DaemonThreadDemo1
{
  public static void main(String[] args)
      {
          System.out.println("Main Thread Started...");

      Thread daemonThread = new Thread(() ->
                {
        while (true)
                    {
          System.out.println("Daemon Thread is running...");
          try
                          {
            Thread.sleep(1000);
          }
                          catch (InterruptedException e)
                          {
              e.printStackTrace();
```

```
            }
         }
      });

      daemonThread.setDaemon(true);
      daemonThread.start();


      Thread userThread = new Thread(() ->
               {
         for (int i=1; i<=9; i++)
                        {
            System.out.println("User Thread: " + i);
            try
                           {
               Thread.sleep(2000);
            }
                           catch (InterruptedException e)
                           {
               e.printStackTrace();
            }
         }
      });

      userThread.start();

      System.out.println("Main Thread Ended...");
   }
}
```
----
interrupt Method of Thread class :

It is a predefined non static method of Thread class. The main purpose of this method to disturb
the execution of the Thread, if the thread is in waiting or sleeping state.

Whenever a thread is interupted then it throws InterruptedException so the thread (if it is in
sleeping or waiting mode) will get a chance to come out from a particular logic.

Points :-

If we call interrupt method and if the thread is not in sleeping or waiting state then it will behave
normally.

If we call interrupt method and if the thread is in sleeping or waiting state then we can stop the thread gracefully.

*Overall interrupt method is mainly used to interrupt the
thread safely so we can manage the resources easily.

Methods :
----
1) public void interrupt () :- Used to interrupt the Thread but the thread must be in sleeping or waiting mode.

2) public boolean isInterrupted() :- Used to verify whether thread is interrupted or not.

```java
class Interrupt extends Thread
{
  @Override
  public void run()
      {
         Thread t = Thread.currentThread();
         System.out.println(t.isInterrupted());

         for(int i=1; i<=5; i++)
              {
                 System.out.println(i);
       try
              {
                     Thread.sleep(1000);
              }
              catch (Exception e)
              {
                      System.err.println("Thread is Interrupted ");
                      e.printStackTrace();
              }

              }
      }
}
public class  InterruptThread
{
        public static void main(String[] args)
          {
      Interrupt it = new Interrupt();
                System.out.println(it.getState());  //NEW
```

```
                    it.start();
                    it.interrupt();  //main thread is interrupting the child thread
            }
}

Note : main thread is interrupting child thread.
-------------------------------------------------------------
class Interrupt extends Thread
{
    @Override
    public void run()
            {
                try
                {
                 Thread.currentThread().interrupt();

                for(int i=1; i<=10; i++)
                        {
                            System.out.println("i value is :"+i);
                            Thread.sleep(1000);
                        }

                }
                        catch (InterruptedException e)
                        {
                                System.err.println("Thread is Interrupted :"+e);
                        }
                        System.out.println("Child thread completed...");
            }
}
public class  InterruptThread1
{
            public static void main(String[] args)
            {
                    Interrupt it = new Interrupt();
                    it.start();
            }
}

Note : Here Child Thread is interrupting itself.
----------------------------------------------------------
public class InterruptThread2
{
```

```java
    public static void main(String[] args)
        {
      Thread thread = new Thread(new MyRunnable());
      thread.start();

      try
                {
        Thread.sleep(3000); //Main thread is waiting for 3 Sec
      }
                catch (InterruptedException e)
                {
        e.printStackTrace();
      }

      thread.interrupt();
    }
}

class MyRunnable implements Runnable
{
    @Override
    public void run()
        {
      try
                {
        while (!Thread.currentThread().isInterrupted())
                        {
          System.out.println("Thread is running by locking the resource");
          Thread.sleep(500);
        }
      }
                catch (Exception e)
                {
        System.out.println("Thread interrupted gracefully.");
      }
                finally
                {
        System.out.println("Thread resource can be release here.");
      }
    }
}
```

Deadlock :

It is a situation where two or more than two threads are in blocked state forever, here threads are waiting to acquire another thread resource without releasing it's own resource.

This situation happens when multiple threads demands same resource without releasing its own attached resource so as a result we get Deadlock situation and our execution of the program will go to an infinite state as shown in the diagram. (20-DEC-24)

```java
public class DeadlockExample
        {
  public static void main(String[] args)
          {
    String resource1 = "Ameerpet";
    String resource2 = "S R Nagar";

   // t1 tries to lock resource1 then resource2

   Thread t1 = new Thread()
                  {
          @Override
     public void run()
                   {
                        synchronized (resource1)
                             {
                         System.out.println("Thread 1: locked resource 1");
                           try
                                {
                                Thread.sleep(1000);
                                }
                                catch (Exception e)
                                {}

                        synchronized (resource2) //Nested synchronized block
                        {
                                System.out.println("Thread 1: locked resource 2");
                        }
            }
      }
    };


   // t2 tries to lock resource2 then resource1
   Thread t2 = new Thread()
```

```
        {
      @Override
      public void run()
            {
      synchronized (resource2)
                        {
        System.out.println("Thread 2: locked resource 2");
            try
                        {
                        Thread.sleep(1000);
                        }
                        catch (Exception e)
                        {}

        synchronized (resource1) //Nested synchronized block
                {
          System.out.println("Thread 2: locked resource 1");
        }
      }
    }
  };
  t1.start();
  t2.start();
 }
}
```

Note : Both the threads are in infinite waiting state so it is Deadlock situation.

21-12-2024

Remaining methods of Object class :
----
protected native Object clone() throws CloneNotSupportedException
----
Object cloning in Java is the process of creating an exact copy of the original object. In other words, it is a way of creating a new object by copying all the data and attributes from the original object.

The clone method of Object class creates an exact copy of an object.

In order to use clone() method , a class must implements Clonable interface because we can perform cloning operation on Cloneable objects only [JVM must have additional information] otherwise cloning opertion is not possible and JVM will throw an exception at runtime

java.lang.CloneNotSupportedException

We can say an object is a Cloneable object if the corresponding class implements Cloneable interface.

It throws a checked Exception i.e CloneNotSupportedException

Note :- clone() method is not the part of Clonable interface[marker interface], actually it is the method of Object class.

clone() method of Object class follows deep copy concept so hashcode will be different as well as if we modify one object content then another object content will not be modified.

clone() method of Object class has protected access modifier so we need to override clone() method in sub class.

Steps we need to follow to perform clone operation :
----
1) The class must implements Cloneable interface
2) Override clone method [throws OR try catch]
3) In this Overridden method call Object class clone method
4) Perform downcasting at the time creating duplicate object
5) Two different objects are created [deep copy]

CloneMethodOperation.java

```java
package com.ravi.cloneable;

class Customer implements Cloneable
{
        private Integer customerId;
        private String customerName;

        public Customer(Integer customerId, String customerName)
        {
                super();
                this.customerId = customerId;
                this.customerName = customerName;
        }

        @Override
        public String toString()
        {
```

```java
                return "Customer [customerId=" + customerId + ", customerName=" +
customerName + "]";
        }


        public void setCustomerId(Integer customerId)
        {
                this.customerId = customerId;
        }

        public void setCustomerName(String customerName)
        {
                this.customerName = customerName;
        }

        @Override
        protected Object clone() throws CloneNotSupportedException
        {
                return super.clone();
        }
}

public class CloneMethodOperation
{
        public static void main(String[] args) throws CloneNotSupportedException
        {
    Customer c1 = new Customer(111, "Scott");
    Customer c2 = (Customer) c1.clone();  //Down casting

    System.out.println("Before Change :");
    System.out.println(c1);
    System.out.println(c2);

    System.out.println("After Change :");
    c2.setCustomerId(222);
    c2.setCustomerName("Smith");
    System.out.println(c1);
    System.out.println(c2);

    System.out.println(".............");
    System.out.println(c1.hashCode());
    System.out.println(c2.hashCode());
```

```
        }
}
-----------------------------------------------------------

protected void finalize() throws Throwable :
---------------------------------------------
It is a predefined method of Object class.

Garbage Collector automatically call this method just before an object is eligible for garbage
collection to perform clean-up activity.

Here clean-up activity means closing the resources associated with that object like file
connection, database connection, network connection and so on we can say resource
de-allocation.

Note :- JVM calls finalize method only one per object.

        This method is deprecated from java 9V.




class Student
{
        private Integer studentId;
        private String studentName;
        public Student(Integer studentId, String studentName)
        {
                super();
                this.studentId = studentId;
                this.studentName = studentName;
        }
        @Override
        public String toString()
        {
                return "Student [studentId=" + studentId + ", studentName=" + studentName + "]";
        }

        @Override
        public void finalize()
        {
          System.out.println("Finalize method is invoked");
        }
}
```

```java
public class FinalizeDemo
{
        public static void main(String[] args) throws InterruptedException
        {
                Student s1 = new Student(111, "Raj");
                System.out.println(s1);

                s1 = null;

                System.gc();  //calling garbage collector explicitly

                Thread.sleep(4000);

                System.out.println(s1);


        }

}
```

ote : If Student object is eligible for GC then JVM will defnetly call finalize() method of Student class.
----

*What is the difference between final, finally and finalize

final :- It is a keyword which is used to provide some kind of   restriction like class is final, Method is
        final,variable is final.

finally :- if we open any resource as a part of try block then   that particular resource must be closed inside
                finally block otherwise program will be terminated ab-normally and the corresponding resource will not be closed (because the remaining lines of try block will not be executed)

finalize() :- It is a method which JVM is calling automatically just before object destruction so if any resource (database, file and network) is associated with
                that particular object then it will be closed
                or de-allocated by JVM by calling finalize().

Collection Frameworks in Java (40 - 45% IQ):

 Collections framework is nothing but handling individual Objects(Collection Interface) and

Group of objects(Map interface).

We know only object can move from one network to another network.

A collections framework is a class library to handle group of Objects.

It is implemented by using java.util package.

It provides an architecture to store and manipulate group of objects.

All the operations that we can perform on data such as searching, sorting, insertion and deletion can be done by using collections framework because It is the data structure of Java.

The simple meaning of collections is single unit of Objects.

It provides the following sub interfaces :

1) List (Accept duplicate elements)
2) Set (Not accepting duplicate elements)
3) Queue (Storing and Fetching the elements based on some order i.e FIFO)

Note : Collection is a predefined interface available in java.util package where as Collections is a predefined class which is available from JDK 1.2V which contains only static methods (Constructor is private)

Methods of Collection interface :
----
a) public boolean add(E element) :- It is used to add an item/element in the in the collection.

b) public boolean addAll(Collection c) :- It is used to insert the specified collection elements in the existing collection(For merging the Collection)

c) public boolean retainAll(Collection c) :- It is used to retain all the elements from existing element. (Common Element)

d) public boolean removeAll(Collection c) :- It is used to delete all the elements from the existing collection.

e) public boolean remove(Object element) :- It is used to delete an element from the collection based on the object.

f) public int size() :- It is used to find out the size of the Collection [Total number of elements available]

g) public void clear() :- It is used to clear all the elements at once from the Collection.

All the above methods of Collection interface will be applicable to all the sub interfaces like List, Set and Queue.

List interface :

It is the sub interface of Collection interface introduced from JDK 1.2V.

It is internally an Array so it stores the object in a sequence order by using index.

Here we can store the object by using index because List interface has provided add(int index, E element) method which will add the object based on the index position.

List interface can accept duplicate elements.

We can perform sorting operation directly by using sort(Comparator<T> cmp) method or by using Collections.sort(List<E> list) interface as a parameter.

We can iterate the elements of List interface by using Iterator and ListIterator interface.

List interface Hierarchy :

This hierarchy diagram is available [26th DEC 24]

Behaviour of List interface Specific classes :

----

* It stores the elements on the basis of index because internally it is using array concept.
* It can accept duplicate, homogeneous and hetrogeneous elements.
* It stores everything in the form of Object.
* When we accept the collection classes without generic concept then compiler generates a warning message because It is unsafe object.
* By using generic (<>) we can eliminate compilation warning and still we can take homogeneous as well as hetrogeneous.(<Object>)
* In list interface few classes are dynamically Growable like Vector and ArrayList. [26-DEC]

==

27-12-2024

Methods of List interface :

----

1) public boolean isEmpty() :- Verify whether List is empty or not

2) public void clear() :- Will clear all the elements, Basically List will become empty.

3) public int size() :- To get the size of the Collections(Total number of elements are available in the collection)

4) public void add(int index, Object o) :- Insert the element based on the index position.

5) public boolean addAll(int index, Collection c) :- Insert the Collection based on the index position

6) public Object get(int index) :- To retrieve the element based on the index position

7) public Object set(int index, Object o) :- To override or replace the existing element based on the index position

8) public Object remove(int index) :- remove the element based on the index position

9) public boolean remove(Object element) :- remove the element based on the object element, It is the Collection interface method extended by List interface

10) public int indexOf() :- index position of the element

11) public int lastIndex() :- last index position of the element

12) public Iterator iterator() :- To fetch or iterate or retrieve the elements from Collection in forward direction only.

13) public ListIterator listIterator() :- To fetch or iterate or retrieve the elements from Collection in forward and backward direction.

*** How many ways we can fetch Collection Object :

There are 9 ways to fetch the Collection Object which are as
follows :

1) By using toString() method of respective class [JDK 1.0]
2) By using Ordinary for loop [JDK 1.0]
3) By using for-each loop [JDK 1.5]
4) By using java.util.Enumeration interface [JDK 1.0]
5) By using java.util.Iterator interface [JDK 1.2]

6) By using java.util.ListIterator interface [JDK 1.2]
7) By using forEach(Consumer<T> cons) Method [JDK 1.8]
8) By using Method Reference(::) [JDK 1.8]
9) By using SplIterator iterface [JDK 1.8]

Note : Among all these, Enumeration, Iterator, ListIterator, SplIterator are the cursors so they can move from one direction to another direction.

Enumeration :

It is a predefined interface available in java.util package from JDK 1.0 onwards(Legacy interface).

We can use Enumeration interface to fetch or retrieve the Objects one by one from the Collection because it is a cursor.

We can create Enumeration object by using elements() method of the legacy Collection class. Internally it uses anonymous inner class object.

public Enumeration elements();

Enumeration interface contains two methods :
----
1) public boolean hasMoreElements() :- It will return true if the Collection is having more elements.

2) public Object nextElement() :- It will return collection object so return type is Object and move the cursor to the next line.

Note :- It will only work with legacy Collections classes.

Iterator interface :

It is a predefined interface available in java.util package available from 1.2 version.

It is used to fetch/retrieve the elements from the Collection in forward direction only because it is also a cursor.

It is also using private inner class i.e Itr class.

public Iterator iterator();

Example :

 Iterator itr = fruits.iterator();

Now, Iterator interface has provided two methods

public boolean hasNext() :-

It will verify, the element is available in the next position or not, if available it will return true otherwise it will return false.

public Object next() :- It will return the collection object and move the cursor to the element object.

ListIterator interface :

It is a predefined interface available in java.util package and it is the sub interface of Iterator available from JDK 1.2v.

It is used to retrieve the Collection object in both the direction i.e in forward direction as well as in backward direction. Here the inner class name is LstItr class.

public ListIterator listIterator();

Example :

ListIterator lit =   fruits.listIterator();

1) public boolean hasNext() :-
It will verify the element is available in the next position or not, if available it will return true otherwise it will return false.

2) public Object next() :- It will return the next position collection object.

3) public boolean hasPrevious() :-
It will verify the element is available in the previous position or not, if available it will return true otherwise it will return false.

4) public Object previous () :- It will return the previous position collection object.

Note :- Apart from these 4 methods we have add(), set() and remove() method in ListIterartor

interface.

Spliterator interface :

It is a predefined interface available in java.util package from java 1.8 version.

It is a cursor through which we can fetch the elements from the
Collection [Collection, array, Stream]

It is the combination of hasNext() and next() method.

It is using forEachRemaining(Consumer <T>) method for fetching the elements.

By using forEach() method :

From java 1.8 onwards every collection class provides a method forEach() method, this method
takes Consumer functional interface as a parameter.
This method is avilable in java.lang.Iterable interface.

How forEach(Consumer<T> cons) method works internally ?

Case 1 :

```java
//Anonymous inner class
package com.ravi.collection;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternal1 {

        public static void main(String[] args)
        {
                Vector<String> fruits = new Vector<>();
                fruits.add("Orange");
                fruits.add("Apple");
                fruits.add("Mango");
                fruits.add("Banana");
                fruits.add("Gauva");

                Consumer<String> cons = new Consumer<String>()
                {
                        @Override
```

```java
                    public void accept(String t)
                    {
                            System.out.println(t.toUpperCase());
                    }
            };

            fruits.forEach(cons);


    }

}
```

Case 2 :

```java
//By using Lambda
package com.ravi.collection;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternal2 {

        public static void main(String[] args)
        {
                Vector<String> fruits = new Vector<>();
                fruits.add("Orange");
                fruits.add("Apple");
                fruits.add("Mango");
                fruits.add("Banana");
                fruits.add("Gauva");

        Consumer<String> cons = str-> System.out.println(str.toUpperCase());

        fruits.forEach(cons);


        }

}
```

Case 3 :

Assigning Lambda to the Consumer Functional interface :

```java
package com.ravi.collection;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternal3 {

        public static void main(String[] args)
        {
                Vector<String> fruits = new Vector<>();
                fruits.add("Orange");
                fruits.add("Apple");
                fruits.add("Mango");
                fruits.add("Banana");
                fruits.add("Gauva");

    fruits.forEach(str-> System.out.println(str.toUpperCase()));

        }

}
```

Method Reference :

It is a new feature introduced in java from JDK 1.8V

It uses :: operator to refer an existing Method.

It is an improvement over Lambda Expression because while working with Lambda we need to
write the method body (logic) but in method reference, we can refer an existing method
to execute by using :: operator.

```java
package com.ravi.collection;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Spliterator;
import java.util.Vector;

public class RetrievingCollectionObject
```

```java
{
    public static void main(String[] args)
    {
        Vector<String> fruits = new Vector<>();
        fruits.add("Orange");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Banana");
        fruits.add("Gauva");

        System.out.println("1) By using toString() Method");
        System.out.println(fruits.toString());

        System.out.println(".............................");
        System.out.println("2) By using for loop ");

        for(int i=0; i<fruits.size(); i++)
        {
            System.out.println(fruits.get(i));
        }

        System.out.println(".............................");
        System.out.println("3) By using for-each loop ");

        for(String fruit : fruits)
        {
            System.out.println(fruit.toUpperCase());
        }

        System.out.println(".............................");
        System.out.println("4) By using Enumeration interface : ");

        Enumeration<String> ele = fruits.elements();

        while(ele.hasMoreElements())
        {
            System.out.println(ele.nextElement());
        }

        System.out.println(".............................");
        System.out.println("5) By using Iterator interface : ");
```

```java
Iterator<String> itr = fruits.iterator();

while(itr.hasNext())
{
     System.out.println(itr.next());
}

System.out.println("..............................");
     System.out.println("6) By using ListIterator interface : ");


     ListIterator<String> lstItr = fruits.listIterator();

     System.out.println("IN FORWARD DIRECTION :");

     while(lstItr.hasNext())
     {
           System.out.println(lstItr.next());
     }

     System.out.println("IN BACKWARD DIRECTION :");

     while(lstItr.hasPrevious())
     {
           System.out.println(lstItr.previous());
     }

     System.out.println("..............................");
     System.out.println("7) By using SplIterator interface : ");

     Spliterator<String> spliterator = fruits.spliterator();
     spliterator.forEachRemaining(fruit -> System.out.println(fruit));

     System.out.println("..............................");
     System.out.println("8) By using forEach() Method : ");
     fruits.forEach(fruit -> System.out.println(fruit));


     System.out.println("..............................");
     System.out.println("9) By using Method Reference : ");
     fruits.forEach(System.out::println);

    }
```

}

Working with List interface implemented classes :

As we know, in List interface we have 4 implemented classes which are as follows :

  1) Vector<E>
  2) Stack<E>
  3) ArrayList<E>
  4) LinkedList<E>

Vector<E> :
----
public class Vector<E> extends AbstractList<E> implements List<E>, Serializable, Clonable, RandomAccess

Vector is a predefined class available in java.util package under List interface.

Vector is always from java means it is available from jdk 1.0 version.

It can accept duplicate, null, homogeneous as well as hetrogeneous elements.

Vector and Hashtable, these two classes are available from jdk 1.0, remaining Collection classes were added from 1.2 version. That is the reason Vector and Hashtable are called legacy(old) classes.

The main difference between Vector and ArrayList is, ArrayList methods are not synchronized so multiple threads can access the method of ArrayList where as on the other hand most the methods are synchronized in Vector so performance wise Vector is slow.

*We should go with ArrayList when Threadsafety is not required on the other hand we should go with Vector when we need ThreadSafety for reterival operation.

It also stores the elements on index basis.It is dynamically growable with initial capacity 10. The next capacity will be 20 i.e double of the first capacity.

new capacity = current capacity * 2;

It implements List, Serializable, Clonable, RandomAccess interfaces.

Constructors in Vector :

We have 4 types of Constructor in Vector

1) Vector v1 = new Vector();
     It will create the vector object with default capacity is 10

2) Vector v2 = new Vector(int initialCapacity);
     Will create the vector object with user specified capacity.

3) Vector v3 = new Vector(int initialCapacity, int capacityIncrement);
     Eg :-   Vector v = new Vector(1000,5);

     Initially It will create the Vector Object with initial capacity 1000 and then when the capacity will be full then increment by 5 so the next capacity would be 1005, 1010 and so on.

 4) Vector v4 = new Vector(Collection c);
     We can achieve loose coupling

30-12-2024

Why generic : To accept type safe object, without generic, type casting is required as well as objects are not type safe so we will get java.lang.ClassCastException.

```
package com.ravi.vector;

import java.util.Vector;

public class GenericDemo
{
        public static void main(String[] args)
        {
                Vector v1 = new Vector<>();
                v1.add(12);
                v1.add(14);
                v1.add(17);
                v1.add(90);

                for(int i=0; i<v1.size(); i++)
                {
                  Integer val = (Integer) v1.get(i);
                  System.out.println(val);
                }
```

```
            System.out.println("Adding String object in the collection");
            v1.add("Ravi");
            v1.add("Hyd");

            for(int i=0; i<v1.size(); i++)
            {
               Integer val = (Integer) v1.get(i);
               System.out.println(val);
            }
      }

}
```

//Program on Vector class :

//Program to show, How we can remove Vector object by using
index as well as Object as a parameter.

package com.ravi.vector;

import java.util.Collections;
import java.util.Vector;

public class VectorDemo {

      public static void main(String[] args)
      {
               Vector<String> listOfCity = new Vector<>();
               listOfCity.add("Hyderabad");
               listOfCity.add("Pune");
               listOfCity.add("Indore");
               listOfCity.add("Bhubneswar");
               listOfCity.add("Kolkata");

      Collections.sort(listOfCity);

      System.out.println(listOfCity);

      //Remove the element based on the index position
      listOfCity.remove(2);
      System.out.println(listOfCity);

      //Remove based on the Object
```

```java
        listOfCity.remove("Kolkata");
        System.out.println(listOfCity);


            }

}
------------------------------------------------------------
//Vector Program on capacity

package com.ravi.vector;

import java.util.*;

public class VectorDemo1 {
        public static void main(String[] args)
        {
                Vector<Integer> v = new Vector<>(100,25);
                System.out.println("Initial capacity is :" + v.capacity());

                for (int i = 0; i < 100; i++)
                {
                        v.add(i);
                }

                System.out.println("After adding 100 elements  capacity is :" + v.capacity());

                v.add(101);
                System.out.println("After adding 101th elements  capacity is :" + v.capacity()); //
200

                for(int i=0; i<v.size(); i++)
                {
                        if(i%5==0)
                        {
                                System.out.println();
                        }
                        System.out.print(v.get(i)+"\t");
                }


        }
}
--------------------------------------------------------------
```

```java
package com.ravi.vector;

//Array To Collection and static method of Collections class

import java.util.*;
public class VectorDemo2
{
        public static void main(String args[])
        {
                Vector<Integer> v = new Vector<>();

                int x[]={22,20,10,40,15,58};

    //Adding array values to Vector
                for(int i=0; i<x.length; i++)
                {
                        v.add(x[i]);
                }
                Collections.sort(v);
                System.out.println("Maximum element is :"+Collections.max(v));
                System.out.println("Minimum element is :"+Collections.min(v));
                System.out.println("Vector Elements :");

                v.forEach(y -> System.out.println(y));

                System.out.println(".....................");
                Collections.reverse(v);
                v.forEach(y -> System.out.println(y));
        }
}
-------------------------------------------------------------
//Working with Custom object

package com.ravi.vector;

import java.util.Vector;

record MobileProduct(Integer productId, String productName)
{
}

public class VectorDemo3
{
```

```java
    public static void main(String[] args)
    {
                Vector<MobileProduct> listOfProducts = new Vector<>();
                listOfProducts.add(new MobileProduct(444, "Apple"));
                listOfProducts.add(new MobileProduct(111, "Redmi"));
                listOfProducts.add(new MobileProduct(222, "Vivo"));
                listOfProducts.add(new MobileProduct(333, "Oppo"));

                listOfProducts.forEach(prod -> System.out.println(prod));

    }
}
```

Program that shows performance wise Vector is not good in comparison to ArrayList

System is a predefined class available in java.lang package and it contains a predefined static method currentTimeMillis() , the return type of this method is long, actually it returns the current time of the system in ms.

```java
  public static native long currentTimeMillis()
```

//Program to describe that ArrayList is better than Vector in performance

```java
package com.ravi.vector;

import java.util.ArrayList;
import java.util.Vector;

public class VectorDemo4
{
        public static void main(String[] args)
        {
                ArrayList<Integer> al = new ArrayList<>();

                long startTime = System.currentTimeMillis();

                for(int i=0; i<1000000; i++)
                {
                        al.add(i);
                }

                long endTime = System.currentTimeMillis();
```

```java
                    long timeTaken = endTime - startTime;

                    System.out.println("Time taken by ArrayList class :"+timeTaken+" ms");


        Vector<Integer> v1 = new Vector<>();

                    startTime = System.currentTimeMillis();

                    for(int i=0; i<1000000; i++)
                    {
                            v1.add(i);
                    }

                    endTime = System.currentTimeMillis();

                    timeTaken = endTime - startTime;

                    System.out.println("Time taken by Vector class :"+timeTaken+" ms");

        }
}
```

----

public Object[] toArray() :

It will convert the Collection object into Array.

```java
package com.ravi.vector;

import java.util.Collections;
import java.util.Vector;

public class VectorDemo5
{

        public static void main(String[] args)
        {
                    Vector<String> listOfCity = new Vector<>();
                    listOfCity.add("Surat");
                    listOfCity.add("Pune");
                    listOfCity.add("Ahmadabad");
                    listOfCity.add("Vanaras");
```

```java
            Collections.sort(listOfCity);

            listOfCity.forEach(System.out::println);

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

            Vector<Integer> listOfNumbers = new Vector<>();
            listOfNumbers.add(500);
            listOfNumbers.add(900);
            listOfNumbers.add(400);
            listOfNumbers.add(300);
            listOfNumbers.add(800);
            listOfNumbers.add(200);
            listOfNumbers.add(100);

            System.out.println("Original Data...");
            System.out.println(listOfNumbers);


            System.out.println("Ascending Order...");
            Collections.sort(listOfNumbers);
            System.out.println(listOfNumbers);

            System.out.println("Descending Order...");
            Collections.sort(listOfNumbers, Collections.reverseOrder());
            System.out.println(listOfNumbers);

            //Converting Our Vector(Collection Object) into Array

             Object[] cities = listOfCity.toArray();

             for(Object city : cities)
             {
                    System.out.println(city);
             }

        }

}
```
----
package com.ravi.vector;

```java
import java.util.Scanner;
import java.util.Vector;

public class VectorDemo6
{
    public static void main(String[] args)
    {
        Vector<String> toDoList = new Vector<>();

        Scanner scanner = new Scanner(System.in);

        int choice;
        do
        {
            System.out.println("To Do List Menu:");
            System.out.println("1. Add Task");
            System.out.println("2. View Tasks");
            System.out.println("3. Mark Task as Completed");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");

            choice = scanner.nextInt();
            scanner.nextLine();

            switch (choice)
            {
                case 1:
                    // Add Task
                    System.out.print("Enter task description: ");
                    String task = scanner.nextLine();
                    toDoList.add(task);
                    System.out.println("Task added successfully!\n");
                    break;
                case 2:
                    // View Tasks
                    System.out.println("To Do List:");
                    for (int i = 0; i < toDoList.size(); i++)
                    {
                        System.out.println((i + 1) + ". " + toDoList.get(i));
                    }
                    System.out.println();
                    break;
                case 3:
```

```java
            // Mark Task as Completed
            System.out.print("Enter task number to mark as completed: ");
            int taskNumber = scanner.nextInt();  //1
            if (taskNumber >= 1 && taskNumber <= toDoList.size())
            {
                String completedTask = toDoList.remove(taskNumber - 1);
                System.out.println("Task marked as completed: " + completedTask + "\n");
            }
            else {
                System.out.println("Invalid task number!\n");
            }
            break;
        case 4:
            System.out.println("Exiting ToDo List application. Goodbye!");
            break;
        default:
            System.out.println("Invalid choice. Please enter a valid option.\n");
        }


    }
    while (choice != 4);



    scanner.close();
  }
}
```

31-12-2024

Enumeration interface method :

From java 9v, Enumeration interface has provided a predefined default method called
asIterator(), the return type of this method is Iterator interface.

    public Iterator asIterator();

VectorDemo7.java
----
package com.ravi.vector;

import java.util.Enumeration;
import java.util.Iterator;

```java
import java.util.Vector;

record Product(int productId, String productName)
{

}

public class VectorDemo7
{
        public static void main(String[] args)
        {
                Vector<Product> listOfProduct = new Vector<>();
                listOfProduct.add(new Product(111, "Laptop"));
                listOfProduct.add(new Product(222, "Mobile"));
                listOfProduct.add(new Product(333, "Camera"));
                listOfProduct.add(new Product(444, "Bag"));
                listOfProduct.add(new Product(555, "Watch"));

                Enumeration<Product> elements = listOfProduct.elements();

                Iterator<Product> itr = elements.asIterator();
                itr.forEachRemaining(System.out::println);

        }

}
```

Note : From Java 8V, Iterator interface is also providing
    forEachRemaining(Consumer<T> cons).

Stack<E>
----
public class Stack<E> extends Vector<E>

It is a predefined class available in java.util package. It is the sub class of Vector class
introduced from JDK 1.0 so, It is also a legacy class.

It is a linear data structure that is used to store the Objects in LIFO (Last In first out) order.

Inserting an element into a Stack is known as push operation where as extracting an element
from the top of the stack is known as pop operation.

It throws an exception called java.util.EmptyStackException, if Stack is empty and we want to

fetch the element.

It has only one constructor as shown below

Stack s = new Stack();

Will create empty Stack Object.

Methods :
----
public E push(Object o) :- To insert an element in the bottom of the Stack.

public E pop() :- To remove and return the element from the top of the Stack.

public E peek() :- Will fetch the element from top of the Stack without removing.

public boolean empty() :- Verifies whether the stack is empty or not (return type is boolean)

public int search(Object o) :- It will search a particular element in the Stack and it returns OffSet
position (int value). If the element is not present in the Stack it will return -1

//Programs on Stack

//Program to insert and fetch the elements from stack

```java
package com.ravi.stack;
import java.util.*;
public class Stack1
{
    public static void main(String args[])
    {
        Stack<Integer> s = new Stack<>();
        try
        {
            s.push(12);
            s.push(15);
                            s.push(22);
                            s.push(33);
                            s.push(49);
                            System.out.println("After insertion elements are :"+s);

            System.out.println("Fetching the elements using pop method");
            System.out.println(s.pop());
            System.out.println(s.pop());
```

```java
                System.out.println(s.pop());
                System.out.println(s.pop());
                System.out.println(s.pop());



                        System.out.println("After deletion elements are :"+s);//[]

                        System.out.println("Is the Stack empty ? :"+s.empty());
        }
                catch(EmptyStackException e)
                {
                   e.printStackTrace();
                }

    }
}
```

//add(Object obj) is the method of Collection implements by Vector class.

```java
package com.ravi.stack;
import java.util.*;
public class Stack2
{
    public static void main(String args[])
    {
        Stack<Integer> st1 = new Stack<>();
        st1.add(10);
        st1.add(20);
        st1.forEach(x -> System.out.println(x));

        Stack<String> st2 = new Stack<>();
        st2.add("Java");
        st2.add("is");
        st2.add("programming");
        st2.add("language");
        st2.forEach(x -> System.out.println(x));

        Stack<Character> st3 = new Stack<>();
        st3.add('A');
        st3.add('B');
        st3.forEach(x -> System.out.println(x));
```

```java
        Stack<Double> st4 = new Stack<>();
        st4.add(10.5);
        st4.add(20.5);
        st4.forEach(x -> System.out.println(x));
    }
}
```

```java
package com.ravi.stack;
import java.util.Stack;

public class Stack3
{
        public static void main(String[] args)
                {
                        Stack<String> stk= new Stack<>();
                        stk.push("Apple");
                        stk.push("Grapes");
                        stk.push("Mango");
                        stk.push("Orange");
                        System.out.println("Stack: " + stk);

                        String fruit = stk.peek();
                        System.out.println("Element at top: " + fruit);
                        System.out.println("Stack elements are : " + stk);
                }
}
```

```java
//Searching an element in the Stack
package com.ravi.stack;
import java.util.Stack;  // 1   -1  false  2
public class Stack4
{
        public static void main(String[] args)
                {
                        Stack<String> stk= new Stack<>();
                        stk.push("Apple");
                        stk.push("Grapes");
                        stk.push("Mango");
                        System.out.println("Offset Position is : " + stk.search("Mango")); //1

                        System.out.println("Offser Position is : " + stk.search("Banana")); //-1
                    System.out.println("Is stack empty ? "+stk.empty());         //false
```

```
                              System.out.println("Index Position is : " + stk.indexOf("Mango")); //2
                    }
          }
          --------------------------------------------------------------
          ArrayList<E>
          ------------
          public class ArrayList<E>  extends AbstractList<E> implements List<E>, Serializable, Clonable,
          RandomAccess
```

It is a predefined class available in java.util package under List interface from java 1.2v.

It accepts duplicate,null, homogeneous and hetrogeneous elements.

It is dynamically growable array.

It stores the elements on index basis so it is simillar to dynamic array.

Initial capacity of ArrayList is 10. The new capacity of Arraylist can be calculated by using the formula
new capacity = (current capacity * 3)/2 + 1 [Almost 50% increment]

*All the methods declared inside an ArrayList is not synchronized so multiple thread can access the method of ArrayList.

*It is highly suitable for fetching or retriving operation when duplicates are allowed and Thread-safety is not required.

Here Iterator is Fail Fast Iteartor.

It implements List,Serializable, Clonable, RandomAccess interfcaes

Constructor of ArrayList :

In ArrayList we have 3 types of Constructor:
Constructor of ArrayList :

1) ArrayList al1 = new ArrayList();
   Will create ArrayList object with default capacity 10.

2) ArrayList al2 = new ArrayList(int initialCapacity);
   Will create an ArrayList object with user specified Capacity

3) ArrayList al3 = new ArrayList(Collection c)

We can copy any Collection interface implemented class data to the current object   reference (Coping one Collection data to another)

01-01-2025

```java
package com.ravi.arraylist;

import java.util.ArrayList;

public class ArrayListDemo
{
    public static void main(String[] args)
    {

        ArrayList<Integer> numbers = new ArrayList<>();

        numbers.add(100);
        numbers.add(200);
        numbers.add(300);
        numbers.add(400);

        int sum = 0;
        for (int number : numbers)
        {
            sum += number;
        }


        System.out.println("Sum of numbers: " + sum);
    }
}
```

```java
package com.ravi.arraylist;

import java.util.ArrayList;

record Customer(Integer custId, String custName, Double custSal)
{
}

public class ArrayListDemo1
{
        public static void main(String[] args)
```

```
        {
                var listofCustomers = new ArrayList<Customer>();
                listofCustomers.add(new Customer(111, "Scott", 123456D));
                listofCustomers.add(new Customer(222, "Smith", 123456D));
                listofCustomers.add(new Customer(333, "Martin", 123456D));
                listofCustomers.add(new Customer(444, "John", 123456D));

                listofCustomers.forEach(System.out::println);


        }
}
```

```
package com.ravi.arraylist;

//Program to merge and retain of two collection addAll()   retainlAll()
import java.util.*;
public class ArrayListDemo2
        {
                public static void main(String args[])
                {
                  ArrayList<String> al1=new ArrayList<>();
                  al1.add("Ravi");
                  al1.add("Rahul");
                  al1.add("Rohit");

                  ArrayList<String> al2=new ArrayList<>();
                  al2.add("Pallavi");
                  al2.add("Sweta");
                  al2.add("Puja");

                  al1.addAll(al2);

        al1.forEach(str -> System.out.println(str.toUpperCase()) );

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

                  ArrayList<String> al3=new ArrayList<>();
                  al3.add("Ravi");
                  al3.add("Rahul");
                  al3.add("Rohit");

                  ArrayList<String> al4=new ArrayList<>();
                  al4.add("Pallavi");
```

```
                al4.add("Rahul");
                al4.add("Raj");

                al3.retainAll(al4);

        al3.forEach(x -> System.out.println(x));
    }
}
```

How to create fixed length and Immutable object :
----
a) Creating a fixed length array by using asList():

Arrays class has provided a predefined static method called asList(T ...x), by using this asList()
method we create a fixed length array.

In this fixed lengh array we can't add/remove any new element but we can replace the existing
element using new element.

If we try to add a new element then we will get an Excption
java.lang.UnsupportedOperationException.

List<E> list = Arrays.asList(T ...x);
----
package com.ravi.arraylist;

import java.util.Arrays;
import java.util.List;

public class FixedLengthArray {

        public static void main(String[] args)
        {
                List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8);
        //numbers.add(9); //Invalid [java.lang.UnsupportedOperationException]
                numbers.set(0, 100);
                System.out.println(numbers);
        }

}
----
b) Creating an immutable List by using List.of(E ...x)

List interface has provided various static methods called of(E ...x) available from java 9 which creates an immutable List.

We can't perform any add or replace operation otherwise we will get java.lang.UnsupportedOperationException.

List<E> list = List.of(E ...x)

```java
package com.ravi.arraylist;

import java.util.List;

public class ImmutableListDemo
{
        public static void main(String[] args)
        {
                List<Integer> immutableList = List.of(1,2,3,4,5,6,7,8,9,10,11,12);
                //immutableList.set(0, 10);
                //immutableList.add(13);

                System.out.println(immutableList);
        }

}
```

```java
//Program to fetch the elements in forward and backward
//direction using ListIterator interface

package com.ravi.arraylist;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

public class ArrayListDemo3
{
public static void main(String args[])
  {
        List<String> listOfName = Arrays.asList("Rohit","Akshar","Pallavi","Sweta"); //Length is fixed
```

```java
        Collections.sort(listOfName);

        //Fetching the data in both the direction
        ListIterator<String> lst = listOfName.listIterator();

        System.out.println("In Forward Direction..");
        while(lst.hasNext())
        {
                System.out.println(lst.next());
        }
        System.out.println("In Backward Direction..");
        while(lst.hasPrevious())
        {
                System.out.println(lst.previous());
        }

 }
}
```

```java
//Serialization and De-serialization on ArrayList Object
package com.ravi.arraylist;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class ArrayListDemo4
{
  public static void main(String[] args) throws IOException
  {
        ArrayList<String> listOfIceCream = new ArrayList<>();
        listOfIceCream.add("Vanila");
        listOfIceCream.add("Strwbarry");
        listOfIceCream.add("Butter Scotch");

        //Serialization Operation
    var fos = new FileOutputStream("D:\\new\\IceCream.txt");
        var oos = new ObjectOutputStream(fos);
```

```java
        try(fos ; oos)
        {
                oos.writeObject(listOfIceCream);
                System.out.println("Data Stored in the file");
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }

        //De-Serialization
        var fin = new FileInputStream("D:\\new\\IceCream.txt");
        var ois = new ObjectInputStream(fin);

        try(fin ; ois)
        {
                ArrayList<String> list = (ArrayList<String>) ois.readObject();
                System.out.println(list);
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }

 }
}
```

Performing Serialization and De-serialization on custom object:

```java
package com.ravi.arraylist;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.ArrayList;

record Employee(Integer employeeId, String employeeName) implements Serializable
{
}
```

```java
public class ArrayListSerialization
{
        public static void main(String[] args) throws IOException
        {
                ArrayList<Employee> listOfEmployees = new ArrayList<>();
                listOfEmployees.add(new Employee(111, "A"));
                listOfEmployees.add(new Employee(222, "B"));
                listOfEmployees.add(new Employee(333, "C"));
                listOfEmployees.add(new Employee(444, "D"));
                listOfEmployees.add(new Employee(555, "E"));

                String filePath = "D:\\new\\Employee.txt";
                //Serialization
                var fos = new FileOutputStream(filePath);
                var oos = new ObjectOutputStream(fos);

                try(fos; oos)
                {
                        oos.writeObject(listOfEmployees);
                        System.out.println("Object data stored successfully");
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }

                //De-Serialization

                var fin = new FileInputStream(filePath);
                var ois = new ObjectInputStream(fin);

                try(fin; ois)
                {

                        ArrayList<Employee> empList = (ArrayList<Employee>) ois.readObject();
                        empList.forEach(System.out::println);

                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
```

```
        }

}
```

Note : As we know ArrayList class already implements from java.io.Serializable interafce (marker interface) but we want to perform serialization operation on Employee object than Employee class should have implement from java.io.Serializable otherwise we will get Runtime Exception java.io.NotSerializableException.

02-01-2025

How to convert one collection object to another collection
object :

```java
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.Vector;

public class LooseCoupling {

        public static void main(String[] args)
        {
                ArrayList<String> cityName = new ArrayList<>();
                cityName.add("Hyd");
                cityName.add("Bglr");
                cityName.add("Kolkata");

                //Convert this ArrayList into Vector
                Vector<String> listOfCity = new Vector<>(cityName);
                System.out.println(listOfCity);


        }

}
```

Note : Vector class has provided a constructor which accepts Collection as a parameter where we can assign any Collection interface implemented class object as shown in the above program.

Collections.sort(List<E> list , Comparator<T> t) :
----
sort() is a predefined static method of Collections class which accept two parameters List and Comparator.

Collections class has provided a predefined static method reverseOrder() which is used to reverse the Collection, the return type is Comaparator interface.

```java
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.Collections;

public class ArrayListDemo5
{
    public static void main(String[] args)
    {
        ArrayList<String> cities = new ArrayList<>();

        cities.add("Hyderabad");
        cities.add("Delhi");
        cities.add("Banglore");
        cities.add("Chennai");

        System.out.println("Before sorting: " + cities);

        Collections.sort(cities);
        System.out.println("After sorting (Ascending): " + cities);

        Collections.sort(cities, Collections.reverseOrder());
        System.out.println("After sorting (Descending): " + cities);

    }
}
```

```java
package com.ravi.arraylist;

//Program on ArrayList that contains null values as well as we can pass
//the element based on the index position
import java.util.ArrayList;
import java.util.LinkedList;
public class ArrayListDemo6
{
```

```java
        public static void main(String[] args)
        {

                ArrayList<Object> al = new ArrayList<>(); //Generic type
                al.add(12);
                al.add("Ravi");
                al.add(12);
                al.add(3,"Hyderabad");
                al.add(1,"Naresh");
                al.add(null);
                al.add(11);
                System.out.println(al);  //12 Naresh Ravi  12  Hyderabad null  11
        }
}
```

```java
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.List;

record Professor(String name, String specialization)
{
}

class Department
{
        private String departmentName;
        private List<Professor> professors;

        public Department(String departmentName)
        {
                super();
                this.departmentName = departmentName;
                this.professors = new ArrayList<Professor>();  //Composition
        }

        public void addProferssor(Professor professor)
        {
                professors.add(professor);
        }

        public String getDepartmentName()
        {
                return departmentName;
```

```java
        }

        public List<Professor> getProfessors()
        {
            return professors;
        }
}

public class ArrayListDemo7
{
    public static void main(String[] args)
    {
        Department department = new Department("Computer Science");

        department.addProferssor(new Professor("James", "Java"));
        department.addProferssor(new Professor("Martin", "Python"));
        department.addProferssor(new Professor("Scott", ".Net"));
        department.addProferssor(new Professor("Smith", "Adv. Java"));

        System.out.println("Department Name is :"+department.getDepartmentName());
        System.out.println("Professors in :"+department.getDepartmentName());

        List<Professor> professors = department.getProfessors();
        professors.forEach(System.out::println);

    }
}
```

```java
package com.ravi.arraylist;

import java.util.ArrayList;

public class ArrayListDemo8
{
    public static void main(String[] args)
    {
        ArrayList<String> original = new ArrayList<>();
        original.add("BCA");
        original.add("MCA");
        original.add("BBA");

        @SuppressWarnings("unchecked")
                ArrayList<String> clonedCopy =(ArrayList<String>) original.clone();
```

```
            System.out.println(clonedCopy);


            ArrayList<String> copy = new ArrayList<>(original);
            System.out.println(copy);



    }
}
```

How to copy the data from the Original List :

We can copy the content from original list by using the following two ways :

 1) By using clone() method
 2) By using constructor (Loose Coupilng)

```java
package com.ravi.arraylist;

import java.util.ArrayList;

public class ArrayListDemo8
{
    public static void main(String[] args)
    {
        ArrayList<String> original = new ArrayList<>();
        original.add("BCA");
        original.add("MCA");
        original.add("BBA");

        @SuppressWarnings("unchecked")
                    ArrayList<String> clonedCopy =(ArrayList<String>) original.clone();
        System.out.println(clonedCopy);


        ArrayList<String> copy = new ArrayList<>(original);
        System.out.println(copy);


    }
}
```
----

public List subList(int fromIndex, int toIndex) :

It is used to fetch/retrieve the part of the List based on the given index. The return type of this method is List, Here fromIndex is inclusive and toIndex is exclusive.

public boolean contains(Object element) :

It is used to find the given element object in the corresponsing List, if available it will return true otherwise false.

public boolean removeIf(Predicate<T> filter)

It is used to remove the elements based on boolean condition passed in the Predicate.

package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo9 {

```java
	public static void main(String[] args)
	{
		ArrayList<Integer> list = new ArrayList<>();
		list.add(1);
		list.add(2);
		list.add(3);
		list.add(4);
		list.add(5);
		list.add(6);
		list.add(7);
		list.add(8);
		list.add(9);
		list.add(10);

		//public List subList(int fromIndex, int toIndex)
		List<Integer> subList = list.subList(2, 5);
		System.out.println(subList);

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

		//public boolean contains(Object obj)
```

```java
        boolean contains = list.contains(9);
        System.out.println(contains);

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

        //public int indexOf(Object obj)
        System.out.println(list.indexOf(1));


        //public boolean removeIf(Predicate<T> filter)
        list.removeIf(num -> num%2==0);
        System.out.println(list);



        }
}
------------------------------------------------------------
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.List;

public class RemoveIfDemo {

        public static void main(String[] args)
        {

                List<String> listOfNames = new ArrayList<String>();
                listOfNames.add("Raj");
                listOfNames.add("Rohit");
                listOfNames.add("Rohan");
                listOfNames.add("Ankit");
                listOfNames.add("Scott");

                System.out.println("Original List");
                System.out.println(listOfNames);

                //Remove all the names which starts from 'R'
                System.out.println("Remove all the name which starts from R");
                listOfNames.removeIf(str -> str.startsWith("R"));
                System.out.println(listOfNames);

        }
```

```
}
```
----
public void trimToSize() :

Used to reduce the capacity.

public void ensureCapacity(int minCapacaity):

Increase the capacity of the ArrayList to avoid frequent resizing.

The minCapacaity parameter will specify that ArrayList will definetly hold the number of
elements specified in the parameter of ensureCapacity() method.

After using ensureCapacity() method, still it is dynamically growable.

```java
package com.ravi.arraylist;

import java.util.ArrayList;

public class ArrayListDemo10 {

    public static void main(String[] args)
    {
            ArrayList<String> list = new ArrayList<>(100);
    list.add("Java");
    list.add("World");

    //public void trimToSize()
    list.trimToSize();
    System.out.println("Trimmed List Size: " + list.size());

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


    ArrayList<Integer> listOfNumber = new ArrayList<>();

    // public void ensureCapacity(int minCapacity)
    //Increase the capacity of the ArrayList to avoid frequent resizing.
    listOfNumber.add(999);

    listOfNumber.ensureCapacity(100);
```

```
        for (int i = 0; i < 50; i++)
        {
            listOfNumber.add(i);
        }

        System.out.println("List size: " + listOfNumber.size());



        }

}
```

Limitation of ArrayList :

Time Complexity of ArrayList :

The time complexity of ArrayList to insert and delete an element from the middle would be O(n)
[Big O of n] because 'n' number of elements will be re-located so, it is not a good choice to
perform insertion and deletion operation in the middle of the List.

On the other hand time complexity of ArrayList to retrieve an element from the List would be
O(1) because by using get(int index) method we can retrieve the element randomly from the list.
ArrayList class implements RandomAccess marker interface which provides the facility to fetch
the elements Randomly.
[02-JAN]

03-01-2025

In order to insert and delete the element in middle of the list frequently, we introduced LinkedList
class.

LinkedList :
----
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>,
Cloneable, Serializable

It is a predefined class available in java.util package under List interface from JDK 1.2v.

It is ordered by index position like ArrayList except the elements (nodes) are doubly linked to
one another. This linkage provide us new method for adding and removing the elements from
the middle of LinkedList.
```

*The important thing is, LikedList may iterate more slowely than ArrayList but LinkedList is a good choice when we want to insert or delete the elements frequently in the list.

From jdk 1.6 onwards LinkedList class has been enhanced to support basic queue operation by implementing Deque<E> interface.

LinkedList methods are not synchronized.

It inserts the elements by using Doubly linked List so insertion and deleteion is very easy.

ArrayList is using Dynamic array data structure but LinkedList class is using LinkedList (Doubly LinkedList) data structure.

At the time of searching an element, It will start searching from first(Head) node or last node OR the closer one.

**Here Iterators are Fail Fast Iterator.

Constructor:
-------------
It has 2 constructors

1) LinkedList list1 = new LinkedList();
    It will create a LinkedList object with 0 capacity.

2) LinkedList list2 = new LinkedList(Collection c);
    Interconversion between the collection

Methods of LinkedList class:
-------------------------------
1) void addFirst(Object o)
2) void addLast(Object o)

3) Object getFirst()
4) Object getLast()

5) Object removeFirst()
6) Object removeLast()

Note :- It stores the elements in non-contiguous memory location.

The time complexcity for insertion and deletion is O(1) The time complexcity for seraching O(n) becuaee it serach the elemnts using node reference.

-----------------------------------------------------------------

LinkedList stores the element on the basis of index :

------------------------------------------------------

```java
package com.ravi.linked_list;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class LinkedListDemo
{
 public static void main(String args[])
 {
     LinkedList<Object> list=new LinkedList<>();
         list.add("Ravi");
         list.add("Vijay");
         list.add("Ravi");
         list.add(null);
         list.add(42);


         System.out.println("3rd Position Element is :"+list.get(3));

         //Iterator interface

          Iterator<Object> itr = list.iterator();
          itr.forEachRemaining(System.out::println); //JDK 1.8


 }
}
```

```java
package com.ravi.linked_list;

import java.util.*;
public class LinkedListDemo1
{
    public static void main(String args[])
    {
        LinkedList<String> list= new LinkedList<>(); //generic
        list.add("Item 2");//2
        list.add("Item 3");//3
```

```java
        list.add("Item 4");//4
        list.add("Item 5");//5
        list.add("Item 6");//6
        list.add("Item 7");//7

        list.add("Item 9"); //10

        list.add(0,"Item 0");//0
        list.add(1,"Item 1"); //1

        list.add(8,"Item 8");//8
                list.add(9,"Item 10");//9
         System.out.println(list);

                          list.remove("Item 5");

                          System.out.println(list);

                          list.removeLast();
                           System.out.println(list);

                            list.removeFirst();
                           System.out.println(list);

                         list.set(0,"Ajay"); //set() will replace the existing value
                         list.set(1,"Vijay");
                         list.set(2,"Anand");
                         list.set(3,"Aman");
                         list.set(4,"Suresh");
                         list.set(5,"Ganesh");
                         list.set(6,"Ramesh");
                         list.forEach(x -> System.out.println(x));


    }
}
```

Note : We are performing frequent insertion and deletion operation, due to its doubly linked list structure, the performance of list will be fast.
----
package com.ravi.linked_list;

//Methods of LinkedList class

```java
import java.util.LinkedList;
public class LinkedListDemo2
{
    public static void main(String[] argv)
    {
        LinkedList<String> list = new LinkedList<>();

        list.addFirst("Ravi");  //  Rahul
        list.add("Rahul");
        list.addLast("Anand");

        System.out.println(list.getFirst());
        System.out.println(list.getLast());

        list.removeFirst();
        list.removeLast();

        System.out.println(list); //[Rahul]
    }
}
```

04-01-2025

```java
package com.ravi.linked_list;
//ListIterator methods (add(), set(), remove())
import java.util.*;
public class LinkedListDemo3
{
        public static void main(String[] args)
        {
                LinkedList<String> city = new LinkedList<> ();
        city.add("Kolkata");
                city.add("Bangalore");
                city.add("Hyderabad");
                city.add("Pune");
                System.out.println(city);

                ListIterator<String> lt = city.listIterator();

        while(lt.hasNext())
                {
                        String cityName =  lt.next();
```

```
                        if(cityName.equals("Kolkata"))
                        {
            lt.remove();
                        }
                        else if(cityName.equals("Hyderabad"))
                        {
            lt.add("Ameerpet");
                        }
                        else if(cityName.equals("Pune"))
                        {
            lt.set("Mumbai");
                        }
                    }
                city.forEach(System.out::println);
        }
}
```

Here there is no ConcurrentModificationException because ListIterator is modifying the structure by it's own method hence there is no problem because it is internal structure modification.

```
package com.ravi.linked_list;

//Insertion, deletion, displaying and exit

import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;

public class LinkedListDemo4
{
 public static void main(String[] args)
        {
    List<Integer> linkedList = new LinkedList<>();
    Scanner scanner = new Scanner(System.in);

      while (true)
                {
      System.out.println("Linked List: " + linkedList); //[]
      System.out.println("1. Insert Element");
      System.out.println("2. Delete Element");
                System.out.println("3. Display Element");
      System.out.println("4. Exit");
      System.out.print("Enter your choice: ");
```

```java
int choice = scanner.nextInt();
switch (choice)
            {
    case 1:
        System.out.print("Enter the element to insert: ");
        int elementToAdd = scanner.nextInt();
        linkedList.add(elementToAdd);
        break;
    case 2:
        if (linkedList.isEmpty())
                        {
            System.out.println("Linked list is empty. Nothing to delete.");
        }
                    else
                    {
            System.out.print("Enter the element to delete: ");
            int elemenetToDelete = scanner.nextInt();
        boolean remove =    linkedList.remove(Integer.valueOf(elemenetToDelete));

            if(remove)
            {
                System.out.println("Element "+elemenetToDelete+ " is deleted Successfully" );
            }
            else
            {
                System.out.println("Element "+elemenetToDelete+" not available is the
LinkedList");
            }

        }
        break;
                        case 3:
                            System.out.println("Elements in the linked list.");
        linkedList.forEach(System.out::println);
                            break;
    case 4:
        System.out.println("Exiting the program.");
        scanner.close();
        System.exit(0);
    default:
        System.out.println("Invalid choice. Please try again.");
}
```

```
        }
   }
}
```

Loose Coupling Program :

```java
package com.ravi.linked_list;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import java.util.Vector;

public class LinkedListDemo5 {

        public static void main(String[] args)
        {


                List<String> listOfName = Arrays.asList("Ravi","Rahul","Ankit", "Rahul");

                LinkedList<String> list = new LinkedList<>(listOfName);
                list.forEach(System.out::println);




        }

}
```

```java
package com.ravi.linked_list;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

record Product(Integer productId, String productName)
{

}
```

```java
public class LinkedListDemo6 {

        public static void main(String[] args)
        {
                List<Product> listOfProduct = new LinkedList<Product>();
                listOfProduct.add(new Product(1, "ApplePhone"));
                listOfProduct.add(new Product(2, "MiPhone"));
                listOfProduct.add(new Product(3, "VivoPhone"));

                System.out.println("Is list empty :"+listOfProduct.isEmpty());

                Iterator<Product> iterator = listOfProduct.iterator();
                iterator.forEachRemaining(prod ->
System.out.println(prod.productName().toUpperCase()));



                String productName = listOfProduct.get(1).productName();
                System.out.println("1st position product name is :"+productName);




        }

}
```

```java
import java.util.Deque;
import java.util.LinkedList;

public class LinkedListDemo6
{
    public static void main(String[] args)
                {
        // Create a LinkedList and treat it as a Deque
        Deque<String> deque = new LinkedList<>();


        deque.addFirst("Ravi");  // Ravi  Pallavi
        deque.addFirst("Raj");


        deque.addLast("Pallavi");
```

```
        deque.addLast("Sweta");


        System.out.println("Deque: " + deque);


        String first = deque.removeFirst();
        String last = deque.removeLast();


        System.out.println("Removed first element: " + first);
        System.out.println("Removed last element: " + last);
        System.out.println("Updated Deque: " + deque);
    }
}
```

Set interface :

Set interface is the sub interface of Collection available from JDK 1.2V

Set interface never accept duplicate elements, Here internally equals(Object obj) method is working from the respective class.

Set interface does not maintain any order (because internally It does not use Array concept, Actually It uses hashing algorithm)

On Set interface we can't use ListIterator interface.

Set interface supports all the methods of Collection interface, few more methods were added from java 9v.

Set interface Hierarchy :
(04-JAN-25)

What is hashing algorithm ?

Hashing algorithm is a technique through which we can search, insert and delete an element in more efficient way in comparison to our classical indexing approach.

Hashing algorithm, internally uses Hashtable data structute, Hashtable data structure internally uses Bucket data structure.

Here elements are inserted by using hashing algorithm so the time complaxity to insert, delete

and search an element would be O(1).

HashSet (UNSORTED, UNORDERED , NO DUPLICATES)

----

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Clonabale, Serializable

It is a predefined class available in java.util package under Set interface and introduced from JDK 1.2V.

It is an unsorted and unordered set.

It accepts hetrogeneous and homogeneous both kind of data.

*It uses the hashcode of the object being inserted into the Collection. Using this hashcode it finds the bucket location.

It doesn't contain any duplicate elements as well as It does not maintain any order while iterating the elements from the collection.

It can accept one null value.

HashSet methods are not synchronized.

HashSet is used for fast searching operation.

It has constant performance in all the operations like insert, delete and search.

It contains 4 types of constructors :

1) HashSet hs1 = new HashSet();
    It will create the HashSet Object with default capacity is 16. The default load fator or Fill Ratio is 0.75   (75% of HashSet is filled up then new HashSet Object will be created having double capacity)

2) HashSet hs2 = new HashSet(int initialCapacity);
    will create the HashSet object with user specified capacity.

3) HashSet hs3 = new HashSet(int initialCapacity, float loadFactor);
    we can specify our own initialCapacity and loadFactor(by default load factor is 0.75)

4) HashSet hs = new HashSet(Collection c);
    Interconversion of Collection.

06-01-2025

//Programs on HashSet

//Unsorted, Unordered and no duplicates

```java
import java.util.*;
public class HashSetDemo
{
 public static void main(String args[])
 {
            HashSet<Integer> hs = new HashSet<>();
                hs.add(67);
                hs.add(89);
                hs.add(33);
                hs.add(null);
                hs.add(45);
                hs.add(12);
                hs.add(35);


                hs.forEach(num-> System.out.println(num));
        }
}
```

Note : At the time of adding element, we call the hashCode() method on the element object, the by default hashcode value of null is 0.

```java
import java.util.*;
public class HashSetDemo1
{
    public static void main(String[] argv)
    {
    HashSet<String> hs=new HashSet<>();
        hs.add("Ravi");
        hs.add("Vijay");
        hs.add("Ravi");
        hs.add("Ajay");
        hs.add("Palavi");
        hs.add("Sweta");
        hs.add(null);
        hs.add(null);
        hs.forEach(str -> System.out.println(str));
```

```java
    }
}
----------------------------------------------------------------
package com.ravi.collection;

import java.util.Arrays;
import java.util.HashSet;

public class HashSetDemo2 {

    public static void main(String[] args)
    {
            Boolean bool[] = new Boolean[5];

            HashSet<String> hs = new HashSet<>();

            bool[0] = hs.add(new String("nit"));
            bool[1] = hs.add("nit");
            bool[2] = hs.add("Haryana");
            bool[3] = hs.add("Jaipur");
            bool[4] = hs.add("Hyderabad");

            System.out.println(Arrays.toString(bool));

            hs.forEach(System.out::println);

            System.out.println("Verify nit object is available or not ");

            if(hs.contains("nit"))
            {
                    System.out.println("nit object is available");
            }
            else
            {
                    System.out.println("nit object is not available");
            }

            //Remove an object if it starts from character 'H'

            hs.removeIf(str -> str.charAt(0) =='H');
            System.out.println(hs);
```

```
        }

}
-----------------------------------------------------------
//add, delete, display and exit
import java.util.HashSet;
import java.util.Scanner;

public class HashSetDemo3
{
    public static void main(String[] args)
            {
        HashSet<String> hashSet = new HashSet<>();
        Scanner scanner = new Scanner(System.in);

        while (true)
                {
            System.out.println("Options:");
            System.out.println("1. Add element");
            System.out.println("2. Delete element");
            System.out.println("3. Display HashSet");
            System.out.println("4. Exit");

            System.out.print("Enter your choice (1/2/3/4): ");
            int choice = scanner.nextInt();

            switch (choice)
                        {
                case 1:
                    System.out.print("Enter the element to add: ");
                    String elementToAdd = scanner.next();
                    if (hashSet.add(elementToAdd))
                                        {
                        System.out.println("Element added successfully.");
                    }
                                        else
                                        {
                        System.out.println("Element already exists in the HashSet.");
                    }
                    break;
                case 2:
                    System.out.print("Enter the element to delete: ");
```

```
                    String elementToDelete = scanner.next();
                    if (hashSet.remove(elementToDelete))
                                            {
                        System.out.println("Element deleted successfully.");
                    }
                                        else
                                        {
                        System.out.println("Element not found in the HashSet.");
                    }
                    break;
                    case 3:
                    System.out.println("Elements in the HashSet:");
                    hashSet.forEach(System.out::println);
                    break;
                    case 4:
                    System.out.println("Exiting the program.");
                    scanner.close();
                    System.exit(0);
                    default:
                    System.out.println("Invalid choice. Please try again.");
            }

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

LinkedHashSet<E> [It maintains order]

----

public class LinkedHashSet extends HashSet implements Set, Clonable, Serializable

It is a predefined class in java.util package under Set interface and introduced from java 1.4v.

It is the sub class of HashSet class.

It is an orderd version of HashSet that maintains a doubly linked list across all the elements.

It internally uses Hashtable and LinkedList data structures.

We should use LinkedHashSet class when we want to maintain an order.

When we iterate the elements through HashSet the order will be unpredictable, while when we iterate the elements through LinkedHashSet then the order will be same as they were inserted

in the collection.

It accepts hetrogeneous and null value is allowed.

It has same constructor as HashSet class.

```java
import java.util.*;
public class LinkedHashSetDemo
{
 public static void main(String args[])
        {
                LinkedHashSet<String> lhs=new LinkedHashSet<>();
                lhs.add("Ravi");
                lhs.add("Vijay");
                lhs.add("Ravi");
                lhs.add("Ajay");
                lhs.add("Pawan");
                lhs.add("Shiva");
                lhs.add(null);
                lhs.add("Ganesh");
                lhs.forEach(str -> System.out.println(str));
        }
}
```

```java
import java.util.*;

public class LinkedHashSetDemo1
{
    public static void main(String[] args)
        {
      LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();

       linkedHashSet.add(10);
       linkedHashSet.add(5);
       linkedHashSet.add(15);
       linkedHashSet.add(20);
       linkedHashSet.add(5);


       System.out.println("LinkedHashSet elements: " + linkedHashSet);

       System.out.println("LinkedHashSet size: " + linkedHashSet.size());
```

```java
        int elementToCheck = 15;
        if (linkedHashSet.contains(elementToCheck))
                    {
            System.out.println(elementToCheck + " is present in the LinkedHashSet.");
        }
                    else
                    {
            System.out.println(elementToCheck + " is not present in the LinkedHashSet.");
        }

        int elementToRemove = 10;
        linkedHashSet.remove(elementToRemove);
        System.out.println("After removing " + elementToRemove + ", LinkedHashSet elements: " +
linkedHashSet);

            linkedHashSet.clear();
        System.out.println("After clearing, LinkedHashSet elements: " + linkedHashSet); //[]
    }
}
```

SortedSet interface :

 As we know Collections.sort(List list) method accept list as a parameter so, we can't perform
sorting operation by using sort() method on HashSet and LinkedHashSet.

 In order to provide automatic sorting facility, Set interface has provided one more interface i.e
SortedSet interface available from JDK 1.2.

 SortedSet interface provided default natural sorting order, default natural sorting order means, if
it is number then ascending order but if it is String then alphabetical OR dictionary order.

 In order to sort the element either in default natural sorting order or user-defined sorting order
we are using Comparable or Comparator interfaces.

07-01-2025

*** Difference between Comparable<T> and Comparator<T> :

Difference is available in the Paint Diagram :[06-JAN-25]

Program on Comparable interface :

Employee.java(R)

```java
 package com.ravi.comparable;

public record Employee(Integer employeeId, String employeeName) implements
Comparable<Employee>
{
        @Override
        public int compareTo(Employee e2)
        {
                return this.employeeName().compareTo(e2.employeeName());
        }

}
```

EmployeeComparable.java

```java
package com.ravi.comparable;

import java.util.ArrayList;
import java.util.Collections;

public class EmployeeComparable
{
        public static void main(String[] args)
        {
                ArrayList<Employee> listOfEmployee = new ArrayList<>();
                listOfEmployee.add(new Employee(333, "Aryan"));
                listOfEmployee.add(new Employee(444, "Raj"));
                listOfEmployee.add(new Employee(222, "Zuber"));
                listOfEmployee.add(new Employee(111, "Satish"));

                Collections.sort(listOfEmployee);
                System.out.println("Sorting based on the Name :");
                listOfEmployee.forEach(System.out::println);
        }

}
```

Limitation of Comparable interface :

The following are the limitation of Comparable interface :

1) We need to modify the Source code to implements the BLC
   class from Comparable interface so we can write sorting logic inside compareTo() method.

2) We can write only one sorting logic that means we can sort   either based on the employeeId or employeeName.

3) Comparable is a Functional interface because It contains
   only one abstract method but due to the current object requirement (this keyword) we can't use Comparable with Lambda Implementation.

To avoid the above said limitations, We introduced Comparator functional interface :

//Program on Comparator Functional interface :

Product.java(R)
----
package com.ravi.comparator;

public record Product(Integer productId, String productName)
{

}
ProductComparator.java
----
package com.ravi.comparator;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class ProductComparator
{
        public static void main(String[] args)
        {
                ArrayList<Product> listOfProduct = new ArrayList<>();
                listOfProduct.add(new Product(444, "Mobile"));
                listOfProduct.add(new Product(333, "Smart Phone"));
                listOfProduct.add(new Product(222, "Laptop"));
                listOfProduct.add(new Product(111, "Camera"));

                System.out.println("Original Data :");
                listOfProduct.forEach(System.out::println);

                //Anonymous class for Comparator
                Comparator<Product> compId = new Comparator<Product>()

```
            {
                    @Override
                    public int compare(Product p1, Product p2)
                    {
                            return p1.productId().compareTo(p2.productId());
                    }
            };

            Collections.sort(listOfProduct, compId);
            System.out.println("Sorting the Product Object using ID :");
            listOfProduct.forEach(System.out::println);

            System.out.println("Sorting the Data by using name :");

            Collections.sort(listOfProduct,(p1,p2)->
p1.productName().compareTo(p2.productName()));
            listOfProduct.forEach(System.out::println);



        }

}
```
==
```
//How to sort Integer Object in descending Order by using
  Comparator interface.

package com.ravi.comparator;

import java.util.ArrayList;
import java.util.Collections;

public class DescendingInteger {

        public static void main(String[] args)
        {
                ArrayList<Integer> listOfNumber = new ArrayList<>();
                listOfNumber.add(56);
                listOfNumber.add(34);
                listOfNumber.add(12);
                listOfNumber.add(9);
                listOfNumber.add(99);

                Collections.sort(listOfNumber,(i1,i2)-> i2.compareTo(i1));
```

```
                System.out.println(listOfNumber);


        }

}
---------------------------------------------------------
List intreface sort() method :
-------------------------------
List interface has provided sort(Comparator<t> cmp) method introduced from JDK 1.8 which
accepts Comapartor as a parameter.

package com.ravi.comparator;

import java.util.ArrayList;

public class NewStyleOfSorting {

        public static void main(String[] args)
        {
                ArrayList<Integer> listOfNumber = new ArrayList<>();
                listOfNumber.add(56);
                listOfNumber.add(34);
                listOfNumber.add(12);
                listOfNumber.add(9);
                listOfNumber.add(99);

                listOfNumber.sort((i1,i2)-> i1.compareTo(i2));

                System.out.println(listOfNumber);

                ArrayList<String> listOfCity = new ArrayList<>();
                listOfCity.add("Ajmer");
                listOfCity.add("Mubai");
                listOfCity.add("Bhubneswar");
                listOfCity.add("Chennai");
                listOfCity.add("Hyderabad");

                listOfCity.sort((s1,s2)-> s2.compareTo(s1));
                System.out.println(listOfCity);


        }
}
```

08-01-2025

TreeSet<E>

TreeSet<E>
----

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Clonable, Serializable

It is a predefined class available in java.util package under Set interface available from JDK 1.2v.

TreeSet class uses Red Black tree data structure.

TreeSet, TreeMap and PriorityQueue are the three sorted collection in the entire Collection Framework so these classes never accepting non comparable objects.

It will sort the elements in natural sorting order i.e ascending order in case of number , and alphabetical order or Dictionary order in the case of String. In order to sort the elements according to user choice, It uses Comparable/Comparator interface.

It does not accept duplicate and null value (java.lang.NullPointerException)

It does not accept non comparable type of data if we try to insert it will throw a runtime exception i.e java.lang.ClassCastException

TreeSet implements NavigableSet.

NavigableSet extends SortedSet.

It contains 4 types of constructors :
----
1) TreeSet t1 = new TreeSet();
     create an empty TreeSet object, elements will be inserted in using Comparable.

2) TreeSet t2 = new TreeSet(Comparator c);
    Customized sorting order.

3) TreeSet t3 = new TreeSet(Collection c);
    loose coupling.

4) TreeSet t4 = new TreeSet(SortedSet s);

We can merge two TreeSet object to copy the data.

```java
//program that describes by default TreeSet provides default natural sorting order
import java.util.*;
public class TreeSetDemo
{
        public static void main(String[] args)
        {
                SortedSet<Integer> t1 = new TreeSet<>();
                t1.add(4);
                t1.add(7);
                t1.add(2);
                t1.add(1);
                t1.add(9);
                System.out.println(t1);

                NavigableSet<String> t2 = new TreeSet<>();
                t2.add("Orange");
                t2.add("Mango");
                t2.add("Banana");
                t2.add("Grapes");
                t2.add("Apple");
                System.out.println(t2);
        }
}
```

```java
package com.ravi.treeset;

import java.util.Iterator;
import java.util.Spliterator;
import java.util.TreeSet;

record Customer(Integer custId, String customerName) implements Comparable<Customer>
{

        @Override
        public int compareTo(Customer c2)
        {
                return this.custId().compareTo(c2.custId());
        }

}
```

```java
public class TreeSetDemo2 {

    public static void main(String[] args)
    {
        TreeSet<Customer> ts1 = new TreeSet<>();
        ts1.add(new Customer(222, "Zuber"));
        ts1.add(new Customer(111, "Vaibhav"));
        ts1.add(new Customer(333, "Aryan"));
        ts1.add(new Customer(444, "Raj"));
        System.out.println("Sorted Based on the ID :");
        ts1.forEach(System.out::println);

        System.out.println("Retrieve Based on the Spliterator :");
        Spliterator<Customer> spl = ts1.spliterator();
        spl.forEachRemaining(System.out::println);

        System.out.println("Retrieve by using descendingIterator in reverse order :");
        Iterator<Customer> itr = ts1.descendingIterator();
        itr.forEachRemaining(System.out::println);

    }

}
```

Note :- descendingIterator() is a predefined method of TreeSet class which will traverse in the descending order and return type of this method is Iterator interface available from JDK 1.6

public Iterator descendingIterator()

//Sort the data by using Comparator interface :

package com.ravi.treeset;

import java.util.TreeSet;

record Product(Integer productId, String productName)
{

}

public class TreeSetDemo3 {

```java
        public static void main(String[] args)
        {
                TreeSet<Product> ts1 = new
TreeSet<>((p1,p2)->p1.productName().compareTo(p2.productName()));
                ts1.add(new Product(333,"Laptop"));
                ts1.add(new Product(222,"Camera"));
                ts1.add(new Product(111,"Mobile"));

                System.out.println(ts1);

        }

}
```

```java
 import java.util.*;
public class TreeSetDemo2
{
        public static void main(String[] args)
        {
                Set<String> t = new TreeSet<>((s1,s2)-> s2.compareTo(s1));
                t.add("6");
                t.add("5");
                t.add("4");
                t.add("2");
                t.add("9");
                Iterator<String> iterator = t.iterator();
                iterator.forEachRemaining(x -> System.out.println(x));


        }
}
```

```java
import java.util.*;

public class TreeSetDemo3
        {
        public static void main(String[] args)
        {
                Set<Character> t = new TreeSet<>();
                t.add('A');
                t.add('C');
                t.add('B');
```

```
                t.add('E');
                t.add('D');
                Iterator<Character> iterator = t.iterator();
                iterator.forEachRemaining(x -> System.out.println(x));
        }
}
```

```
package com.ravi.treeset;

import java.util.ArrayList;
import java.util.TreeSet;

public class TreeSetDemo4 {

        public static void main(String[] args)
        {
                ArrayList<String> al = new ArrayList<>();
                al.add("Mango");
                al.add("Orange");
                al.add("Apple");

                //ArrayList to TrreSet by using Collection(I)
                TreeSet<String> ts = new TreeSet<>(al);
                System.out.println(ts);


                //TreeSet to TreeSet by using SortedSet
                TreeSet<String> ts2 = new TreeSet<>(ts);
                System.out.println(ts2);


        }

}
```

```
//TreeSet with custom object to sort the data in different criteria
 package com.ravi.treeset;

import java.util.TreeSet;

record Student(Integer studentId, String studentName)
{
```

```
}

public class TreeSetDemo5
{
        public static void main(String[] args)
        {
          TreeSet<Student> ts1 = new TreeSet<>((st1,
st2)->st1.studentId().compareTo(st2.studentId()) );
          ts1.add(new Student(444, "Aryan"));
          ts1.add(new Student(111, "Zuber"));
          ts1.add(new Student(222, "Raj"));
          ts1.add(new Student(333, "Rahul"));
          System.out.println("Sorting based on the Id :");
          ts1.forEach(std -> System.out.println(std));

          TreeSet<Student> ts2 = new TreeSet<>((st1,
st2)->st1.studentName().compareTo(st2.studentName()));
          ts2.add(new Student(444, "Aryan"));
          ts2.add(new Student(111, "Zuber"));
          ts2.add(new Student(222, "Raj"));
          ts2.add(new Student(333, "Rahul"));
          System.out.println("Sorting based on the Name :");
          ts2.forEach(std -> System.out.println(std));
        }

}
```

Methods of SortedSet interface :
----
public E first() :- Will fetch first element

public E last() :- Will fetch last element

public SortedSet headSet(int range) :- Will fetch the values which are less than specified range.

public SortedSet tailSet(int range) :- Will fetch the values which are equal and greater than the specified range.

public SortedSet subSet(int startRange, int endRange) :- Will fetch the range of values where startRange is inclusive and endRange is exclusive.

Note :- headSet(), tailSet() and subSet(), return type is SortedSet.

```java
import java.util.*;
public class SortedSetMethodDemo
{
    public static void main(String[] args)
    {
        TreeSet<Integer> times = new TreeSet<>();
        times.add(1205);
        times.add(1505);
        times.add(1545);
                    times.add(1600);
        times.add(1830);
        times.add(2010);
        times.add(2100);

        SortedSet<Integer> sub = new TreeSet<>();

                    sub =  times.subSet(1545,2100);
        System.out.println("Using subSet() :-"+sub);//[1545, 1600,1830,2010]
        System.out.println(sub.first());
        System.out.println(sub.last());

            sub = times.headSet(1545);
                System.out.println("Using headSet() :-"+sub); //[1205, 1505]

             sub =  times.tailSet(1545);
                 System.out.println("Using tailSet() :-"+sub); //[1545 to 2100]

    }
}
```

NavigableSet :

It is a predefined interface available in java.util package from JDK 1.6v

It is used to navigate among the elements, Unlike SortedSet which provides range of value.
Here we can navigate among the values as shown below.

```java
import java.util.*;

public class NavigableSetDemo
{
   public static void main(String[] args)
```

```
    {
        NavigableSet<Integer> ns = new TreeSet<>();
        ns.add(1);
        ns.add(2);
        ns.add(3);
        ns.add(4);
        ns.add(5);
        ns.add(6);

                System.out.println("lower(3): " + ns.lower(3));//Just below than the specified
element or null

        System.out.println("floor(3): " + ns.floor(3)); //Equal  less or null

        System.out.println("higher(3): " + ns.higher(3));//Just greater than specified element or null

        System.out.println("ceiling(3): " + ns.ceiling(3));//Equal or greater or null



    }
}
```

Map interface :

Map Hierarchy diagram available [08-JAN-25]

09-01-2025

Map interface :

As we know Collection interface is used to hold single Or individual object but Map interface will hold group of objects in the form key and value pair. {key = value}

Map<K,V> interface is not the part the Collection, It is a separate interface.

Before Map interface We have Dictionary<K,V>(abstract class) class and it is extended by Hashtable<K,V> class in JDK 1.0V

Map interface works with key and value pair introduced from 1.2V.

Here key and value both are objects.

Here key must be unique and value may be duplicate.

Each key and value pair is creating one Entry.(Entry is nothing but the combination of key and value pair)

```
interface Map<K,V>
{
    interface Entry<K,V>
     {
        //key and value
     }
}
```

How to represent this entry interface (Map.Entry in .java) [Map$Entry in .class]

In Map interface whenever we have a duplicate key then the old key value will be replaced by new key(duplicate key) value.

Map interface has defined forEach(BiConsumer cons) method to work with group of Objects.It does not extends Iterable interface.

Iterator and ListIterator we can't work directly using Map.

Methods of Map interface :
----
1) Object put(Object key, Object value) :- To insert one entry in the Map collection. It will return the value of old Object key, if the key is already available(Duplicate key), If key is not available (new key) then it will return null.

2) Object putIfAbsent(Object key, Object value) :- It will insert an entry, if and only if, key is not available , if the key is already available then it will not insert the Entry to the Map Collection

3) Object get(Object key) :- It will return corresponding value of key, if the key is not present then it will return null.

4) Object getOrDefault(Object key, Object defaultValue) :- To avoid null value this method has been introduced from JDK 1.8V, here we can pass some defaultValue to avoid the null value.

5) boolean containsKey(Object key) :- To Search a particular key

6) boolean containsValue(Object value) :- To Search a particular value

7) int size() :- To count the number of Entries.

8) remove(Object key) :- One complete entry will be removed.

9) void clear() :- Used to clear the Map

10) boolean isEmpty() :- To verify Map is empty or not?

11) void putAll(Map m) :- Merging of two Map collection

Methods of Map interface to convert the Map into Collection :

We have Map interafce methods through which we can convert Map interface into Collection
interface which is known as collection views method.

1) public Set<Object> keySet() : It will retrieve all the keys.

2) public Collection values() : It will retrieve all the values.

3) public Set<Map.Entry> entrySet() : It will retrieve key and value both in a single object.
         a) getKey()
           b) getValue()

10-01-2025

**** How HashMap works internally ?
----
a) While working with HashSet or HashMap every object must be compared because duplicate
objects are not allowed.

b) Whenever we add any new key to verify whether key is unique or duplicate, HashMap
internally uses hashCode(), == operator and equals method.

c) While adding the key object in the HashMap, first of all it will invoke the hashCode() method
to retrieve the corresponding key hashcode value.
    Example :- hm.put(key,value);
           then internally key.hashCode();

d) If the newly added key and existing key hashCode value both are same (Hash collision), then

only == operator is used for comparing those keys by using reference or memory address, if both keys references are same then existing key value will be replaced with new key value.

        If the reference of both keys are different then only equals(Object obj) method is invoked to compare those keys by using state(data). [content comparison]

        If the equals(Object obj) method returns true (content wise both keys are same), this new key is duplicate then existing key value will be replaced by new key value.

        If equals(Object obj) method returns false, this new key is unique, new entry (key-value) will be inserted in the same Bucket by using Singly LinkedList

        Note :- equals(Object obj) method is invoked only when two keys are having same hashcode as well as their references are different.

e) Actually by calling hashcode method we are not comparing the objects, we are just storing the objects in a group so the currently adding key object will be compared with its SAME HASHCODE GROUP objects, but not with all the keys which are available in the Map.

f) The main purpose of storing objects into the corresponding group to decrease the number of comparison so the efficiency of the program will increase.

g) To insert an entry in the HashMap, HashMap internally uses Hashtable data structure.

h) Now, for storing same hashcode object into a single group, hash table data structure internally uses one more data structure called Bucket.

i) The Hashtable data structure internally uses Node class array object.

j) The bucket data structure internally uses LinkedList data structure, It is a single linked list again implemented by Node class only.

*k) A bucket is group of entries of same hash code keys.

l) Performance wise LinkedList is not good to serach, so from java 8 onwards LinkedList is changed to Binary tree to decrease the number of comparison within the same bucket hashcode if the number of entries are greater than 8.

** equals() and hashCode() method contract :

Both the methods are working together to find out the duplicate objects in the Map.

*If equals() method invoked on two objects and it returns true then hashcode of both the objects must be same.

Note : IF TWO OBJECTS ARE HAVING SAME HASH CODE THEN IT MAY BE SAME OR DIFFERENT BUT IF EQUALS(OBJECT OBJ) METHOD RETURN TRUE THEN BOTH OBJECTS MUST RETURN SAME HASHCODE.

```java
package com.ravi.map;

import java.util.HashMap;

public class HashMapInternal {

    public static void main(String[] args)
    {

                HashMap<String,Integer> hm1 = new HashMap<>();
                hm1.put("A", 1);
                hm1.put("A", 2);
                hm1.put(new String("A"), 3);
                System.out.println("Size is :"+hm1.size());
                System.out.println(hm1);

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

                HashMap<Integer,Integer> hm2 = new HashMap<>();
                hm2.put(128, 1);
                hm2.put(128, 2);
                System.out.println("Size is :"+hm2.size());
                System.out.println(hm2);
                System.out.println("...................");



                HashMap<Object,Object> hm3 = new HashMap<>();
                hm3.put("A", 1);
                hm3.put("A", 2);
                hm3.put(new String("A"), 3);
                hm3.put(65, 4);
                System.out.println("Size is :"+hm3.size());
                System.out.println(hm3);
```

```
        }

}
```

What will happen if we don't follow the contract ?
Case 1 :

If we override only equals(Object obj)

If we override only equals(Object obj) method for content comparison
then same object (duplicate object) will have different hashcode (due to new keyword) hence
same object (content wise) will move into two different buckets [Duplication].

Case 2 :

If we override only hashCode() method

If we overrdie only hashCode() method then two objects which are having same hashcode (due
to overriding) will go to same bucket but == operator and equals(Object obj) method of Object
class, both will return false hence duplicate object will be inserted into same bucket.

So, the conclusion is, compulsory we need to override both the methods for removing duplicate
elements.

```java
package com.ravi.map;

import java.util.HashMap;
import java.util.Objects;

class Customer
{
        private int customerId;
        private String customerName;

        public Customer(int customerId, String customerName)
        {
                super();
                this.customerId = customerId;
                this.customerName = customerName;
        }

        @Override
```

```java
        public int hashCode() {
                return Objects.hash(customerId, customerName);
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                Customer other = (Customer) obj;
                return customerId == other.customerId && Objects.equals(customerName,
other.customerName);
        }


}
public class HashMapInternalDemo1
{

        public static void main(String[] args)
        {
          Customer c1 = new Customer(111, "Scott");
          Customer c2 = new Customer(111, "Scott");

          System.out.println(c1.hashCode()+" : "+c2.hashCode());

          HashMap<Customer,String> map = new HashMap<>();
          map.put(c1, "A");  //c1   B
          map.put(c2, "B");

          System.out.println(map.size());
        }

}
```

Customer class we are using as a HashMap key so it must override
hashCode() and equals(Object obj) as well as as advanced level, It must be immutable class.

All the Wrapper classes and String class are immutable as well as

hashCode() and equals(Object obj) methods are overridden in these classes so perfectly suitable to becoming HashMap key.

```java
package com.ravi.map;

import java.util.HashMap;

record Customer(Integer id, String name)
{

}

public class HashMapInternalDemo
{

        public static void main(String[] args)
        {
          Customer c1 = new Customer(111, "Scott");
          Customer c2 = new Customer(111, "Scott");

          //System.out.println(c1.hashCode()+" : "+c2.hashCode());

          HashMap<Customer,String> map = new HashMap<>();
          map.put(c1, "A");
          map.put(c2, "B");

          System.out.println(map.size());
        }

}
```

so final conclusion is, In our user-defined class which we want to use as a HashMap key must be immutable and hashCode() and equals(Object obj) method must be overridden.

Instead of BLC class we can also use simply record because record is implicitly final and hashCode() and equals(Object obj) methods are overridden.

```java
package com.ravi.map;

import java.util.Scanner;

public class CustomStringHashCode
{
```

```java
    public static int customHashCode(String str)
    {
        if (str == null)
        {
            return 0; // default hashCode value for null is 0
        }

        int hashCode = 0;

        for (int i = 0; i < str.length(); i++)  //NIT
        {
            char charValue = str.charAt(i);
            hashCode = 31 * hashCode + charValue;
        }

        return hashCode;
    }

    public static void main(String[] args)
    {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter your String :");
            String str = sc.next();

            System.out.println(str+" hashcode from String class :"+str.hashCode());

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

            System.out.println(str+" hashcode for my class
:"+CustomStringHashCode.customHashCode(str));
            sc.close();
    }
}
```

Note : Both methods are producing same hashcode value.
     Hashcode may also be -ve value so use shift operator to
     make it positive.

HashMap<K,V> [UNORDERED, UNSORTED, DUPLICATE KEYS ARE NOT ALLOWED]

HashMap<K,V> :- [Unsorted, Unordered, No Duplicate keys]
----
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Serializable,

Clonable

It is a predefined class available in java.util package under Map interface available from JDK 1.2v.

It gives us unsorted and Unordered map. when we need a map and we don't care about the order while iterating the elements through it then we should use HashMap.

It inserts the element based on the hashCode of the Object key using hashing technique [hasing alogorithhm]

It does not accept duplicate keys but value may be duplicate.

It accepts only one null key(because duplicate keys are not allowed) but multiple null values are allowed.

HashMap is not synchronized.

Time complexcity of search, insert and delete will be O(1)

We should use HashMap to perform fast searching opeartion.

For eliminating duplicate keys in hashMap object we should compulsory follow the contract between hashcode and equals(Object obj)

It contains 4 types of constructor

1) HashMap hm1 = new HashMap();
    It will create the HashMap Object with default capacity is 16. The default load fator or Fill Ratio is 0.75   (75% of HashMap is filled up then new HashMap Object will be created having double capacity)

2) HashMap hm2 = new HashMap(int initialCapacity);
    will create the HashMap object with user specified capacity

3) HashMap hm3 = new HashMap(int initialCapacity, float loadFactor);
    we can specify our own initialCapacity and loadFactor(by default load factor is 0.75)

4)HashMap hm4 = new HashMap(Map m);
    Interconversion of Map Collection

//Program on HashMap :

```
----------------------
package com.ravi.map;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;

public class HashMapDemo1
{
        public static void main(String[] args)
        {
    HashMap<Integer, String> map = new HashMap<>();

    map.put(1, "Vanilla");
    map.put(2, "Butterscotch");
    map.put(3, "Chocolate");
    map.put(4, "Cotton Candy");

    System.out.println("HashMap: " + map); //{key = value}

    String value = map.get(2);
    System.out.println("Value for key 2: " + value);


    value = map.getOrDefault(3, "Key is not available");
    System.out.println("Value for key 3: " + value);


    boolean hasKey = map.containsKey(3);
    System.out.println("HashMap contains key 3: " + hasKey);

    boolean hasValue = map.containsValue("Vanilla");
    System.out.println("HashMap contains value 'Vanilla': " + hasValue);


    map.remove(1);
    System.out.println("HashMap after removing key 1: " + map);


    System.out.println("Iterating through HashMap:");
    for (HashMap.Entry<Integer, String> entry : map.entrySet())
    {
```

```java
                System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
            }


        System.out.println("Iterating through Iterator");

        Iterator<Entry<Integer, String>> itr = map.entrySet().iterator();
        itr.forEachRemaining(System.out::println);

        System.out.println("Iterating through forEach(BiConsumer<T,U>)");
        map.forEach((k,v)-> System.out.println("Key is :"+k+" Value is :"+v));


        int size = map.size();
        System.out.println("Size of HashMap: " + size);

        map.clear();
        System.out.println("HashMap after clearing: " + map); //{}


    }
}
----------------------------------------------------------------
package com.ravi.map;

import java.util.HashMap;

public class HashMapDemo2
{
    public static void main(String[] args)
    {
        HashMap<Integer, String> studentRecords = new HashMap<>();

        studentRecords.put(101, "Scott");
        studentRecords.put(102, "Smith");
        studentRecords.put(103, "Martin");
        studentRecords.put(104, "Aryan");

        System.out.println("Student Records: " + studentRecords);


        int searchId = 109;
        String studentName = studentRecords.get(searchId);
```

```java
        if (studentName != null)
        {
            System.out.println("Student with ID " + searchId + " is " + studentName);
        }
        else
        {
            System.out.println("Student with ID " + searchId + " not found.");
        }



        System.out.println(studentRecords.put(103, "Rahul"));
        System.out.println("Updated Records: " + studentRecords);


        studentRecords.remove(104);
        System.out.println("Records after removal: " + studentRecords);


        int idToCheck = 101;
        System.out.println("Does ID " + idToCheck + " exist? " +
studentRecords.containsKey(idToCheck));


        String nameToCheck = "Aryan";
        System.out.println("Does name '" + nameToCheck + "' exist? " +
studentRecords.containsValue(nameToCheck));


        System.out.println("Iterating through records:");
        for (HashMap.Entry<Integer, String> entry : studentRecords.entrySet())
        {
            System.out.println("ID: " + entry.getKey() + ", Name: " + entry.getValue());
        }

        studentRecords.clear();
        System.out.println("All records cleared: " + studentRecords);

    }
}
```

```java
package com.ravi.map;
```

```java
import java.util.Collection;
import java.util.HashMap;
import java.util.Set;

public class HashMapDemo3
{
  public static void main(String[] args)
  {
            HashMap<Integer,String> newmap1 = new HashMap<>();

            HashMap<Integer,String> newmap2 = new HashMap<>();

            newmap1.put(1, "OCPJP");
            newmap1.put(2, "is");
            newmap1.put(3, "best");

            System.out.println("Values in newmap1: "+ newmap1);

            newmap2.put(4, "Exam");

            newmap2.putAll(newmap1);

            System.out.println("Iterating through forEach()");
            newmap2.forEach((k,v)->System.out.println(k+" : "+v));

            System.out.println("All the Unique keys");
            Set<Integer> setOfKeys = newmap2.keySet();
            System.out.println(setOfKeys);

            System.out.println("All the values");
            Collection<String> values = newmap2.values();
            System.out.println(values);

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

            System.out.println("Loose Coupling for Merging one Map to another");

            HashMap<Integer, String> hm1 = new HashMap<>();

            hm1.put(1, "Ravi");
            hm1.put(2, "Rahul");
            hm1.put(3, "Rajen");
```

```java
        HashMap<Integer, String> hm2        = new HashMap<>(hm1);

        System.out.println("Mapping of HashMap hm1 are : "        + hm1);

        System.out.println("Mapping of HashMap hm2 are : " + hm2);


 }
}
```

```java
package com.ravi.map;

import java.util.HashMap;

record Employee(Integer empId, String empName)
{

}

public class HashMapDemo4
{
        public static void main(String[] args)
        {
                Employee e1 = new Employee(101,"Aryan");
                Employee e2 = new Employee(102,"Pooja");
                Employee e3 = new Employee(101,"Aryan");
                Employee e4 = e2;

                HashMap<Employee,String> hm = new HashMap<>();
                hm.put(e1,"Ameerpet");
                hm.put(e2,"S.R Nagar");
                hm.put(e3,"Begumpet");
                hm.put(e4,"Panjagutta");

                hm.forEach((k,v)-> System.out.println(k+" : "+v));
        }
}
```

20-01-2025

```java
package com.ravi.map;

import java.util.HashMap;

public class HashMapDemo5
{
    public static void main(String[] args)
    {
        // Create a HashMap to store book titles and their availability (true = available, false =
borrowed)

        HashMap<String, Boolean> library = new HashMap<>();

        library.put("Core Java", true);
        library.put("Advanced Java", true);
        library.put("HTML", false);
        library.put("JavaScript", true);

        // Display the initial library status
        System.out.println("Initial Library Status: " + library);


        // Borrow a book
        String bookToBorrow = "Advanced Java";

        if (library.containsKey(bookToBorrow) && library.get(bookToBorrow))
        {
            library.put(bookToBorrow, false);
            System.out.println(bookToBorrow + " has been borrowed.");
        }
        else
        {
            System.out.println(bookToBorrow + " is not available for borrowing.");
        }


        String bookToReturn = "HTML";

        if (library.containsKey(bookToReturn) && !library.get(bookToReturn))
        {
            library.put(bookToReturn, true); // Mark the book as available
            System.out.println(bookToReturn + "Book has been returned.");
```

```java
        }
        else
        {
            System.out.println(bookToReturn + "Book is not borrowed.");
        }


        // Check the availability of a book
        String bookToCheck = "JavaScript";

        if (library.containsKey(bookToCheck))
        {
          String availability = library.get(bookToCheck) ? "available" : "borrowed";
            System.out.println(bookToCheck + " Book is " + availability + ".");
        }
        else
        {
            System.out.println(bookToCheck + " is not in the library.");
        }


        //Display the final library status

        System.out.println("Final Library Status:");
        for (HashMap.Entry<String, Boolean> entry : library.entrySet())
        {
            String status = entry.getValue() ? "Available" : "Borrowed";
            System.out.println("Book: " + entry.getKey() + ", Status: " + status);
        }


    }
}
```

LinkedHashMap<K,V>

LinkedHashMap<K,V>

----

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

It is a predefined class available in java.util package under Map interface available from 1.4.

It is the sub class of HashMap class.

It maintains insertion order. It contains a doubly linked with the elements or nodes so It will iterate more slowly in comparison to HashMap.

It uses Hashtable and LinkedList data structure.

If We want to fetch the elements in the same order as they were inserted in the Map then we should go with LinkedHashMap.

It accepts one null key and multiple null values.

It is not synchronized.

It has also 4 constructors same as HashMap

1) LinkedHashMap hm1 = new LinkedHashMap();
   will create a LinkedHashMap with default capacity 16 and load factor 0.75

2) LinkedHashMap hm1 = new LinkedHashMap(iny initialCapacity);

3) LinkedHashMap hm1 = new LinkedHashMap(int initialCapacity, float loadFactor);

4) LinkedHashMap hm1 = new LinkedHashMap(Map m);

```java
import java.util.*;
public class LinkedHashMapDemo
{
        public static void main(String[] args)
        {
                LinkedHashMap<Integer,String> l = new LinkedHashMap<>();
                l.put(1,"abc");
                l.put(3,"xyz");
                l.put(2,"pqr");
                l.put(4,"def");
                l.put(null,"ghi");
                System.out.println(l);
        }
}
```
----
import java.util.*;

```java
public class LinkedHashMapDemo1
{
    public static void main(String[] a)
    {
        Map<String,String> map = new LinkedHashMap<>();
        map.put("Ravi", "1234");
                map.put("Rahul", "1234");
                map.put("Aswin", null);
                map.put("Samir", null);

                map.forEach((k,v)->System.out.println(k+" : "+v));
    }
}
```

Note : To maintain order we should use LinkedHashMap.

Hashtable<K,V>
----
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Clonable, Serializable

It is predefined class available in java.util package under Map interface from JDK 1.0.

Like Vector, Hashtable is also form the birth of java so called legacy class.

It is the sub class of Dictionary class which is an abstract class.

*The major difference between HashMap and Hashtable is, HashMap methods are not synchronized where as Hastable methods are synchronized.

HashMap can accept one null key and multiple null values where as Hashtable does not contain anything as a null(key and value both). if we try to add null then JVM will throw an exception i.e NullPointerException.

The initial default capacity of Hashtable class is 11 where as loadFactor is 0.75.

It has also same constructor as we have in HashMap.(4 constructors)

1) Hashtable hs1 = new Hashtable();
    It will create the Hashtable Object with default capacity as 11 as well as load factor is 0.75

2) Hashtable hs2 = new Hashtable(int initialCapacity);

will create the Hashtable object with specified capacity

3) Hashtable hs3 = new Hashtable(int initialCapacity, float loadFactor);
   we can specify our own initialCapacity and loadFactor

4) Hashtable hs = new Hashtable(Map c);
   Interconversion of Map Collection

```java
import java.util.*;
public class HashtableDemo
     {
      public static void main(String args[])
            {
              Hashtable<Integer,String> map=new Hashtable<>();
              map.put(1, "Java");
              map.put(2, "is");
              map.put(3, "best");
              map.put(4,"language");

              //map.put(5,null);

              System.out.println(map);

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

               for(Map.Entry m : map.entrySet())
                  {
                          System.out.println(m.getKey()+" = "+m.getValue());
                  }
          }
}
```

```java
import java.util.*;
public class HashtableDemo1
{
   public static void main(String args[])
         {
   Hashtable<Integer,String> map=new Hashtable<>();
    map.put(1,"Priyanka");
    map.put(2,"Ruby");
    map.put(3,"Vibha");
    map.put(4,"Kanchan");
```

```
        map.putIfAbsent(5,"Bina");
        map.putIfAbsent(24,"Pooja");
        map.putIfAbsent(26,"Ankita");

    map.putIfAbsent(1,"Sneha");
    System.out.println("Updated Map: "+map);
 }
}
```

WeakHashMap<K,V> :
----
public class WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>

It is a predefined class in java.util package under Map interface.It was introduced from JDK 1.2v onwards.

While working with HashMap, keys of HashMap are of strong reference type. This means the entry of map will not be deleted by the garbage collector even though the key is set to be null as well as Object is also not eligible for Garbage Collector.

On the other hand while working with WeakHashMap, keys of WeakHashMap are of weak reference type. This means the entry and corresponding object of a map is deleted by the garbage collector if the key value is set to be null because it is of weak type.

So, HashMap dominates over Garbage Collector where as Garbage Collector dominates over WeakHashMap.

It does not implements Cloneable and Serailizable because It is mainly used for inventory system where we need to manage the data and we can insert and delete object data frequently in the inventory.

It contains 4 types of Constructor :
----
1) WeakHashMap wm1 = new WeakHashMap();

    Creates an empty WeakHashMap object with default capacity is 16 and load fator 0.75

2) WeakHashMap wm2 = new WeakHashMap(int initialCapacity);

3) WeakHashMap wm3 = new WeakHashMap(int initialCapacity, float loadFactor);

Eg:- WeakHashMap wm = new WeakHashMap(10,0.9);

capacity - The capacity of this map is 10. Meaning, it can store 10 entries.

loadFactor - The load factor of this map is 0.9. This means whenever our hashtable is filled up by 90%, the entries are moved to a new hashtable of double the size of the original hashtable.

4) WeakHashMap wm4 = new WeakHashMap(Map m);

package com.ravi.map;

import java.util.WeakHashMap;

```java
record Product(Integer productId, String productName)
{

        @Override
        public void finalize()
        {
           System.out.println("Product Object is eligible for GC" );
        }
}

public class WeakHashMapDemo {

        public static void main(String[] args) throws InterruptedException
        {
                Product p1 = new Product(111, "Camera");

                WeakHashMap<Product,String> product = new WeakHashMap<>();
                product.put(p1, "Hyderabad");

                System.out.println(product);

                p1 = null;

                System.gc();

                Thread.sleep(3000);

                System.out.println(product); //{}
```

}

}

Note : Here Product object and WeakHashMap entry both will be
        deleted because keys are weak type.
        It is suitable for database inventory where we want frequently delete the Product Object.
----
2 Days : Map
1 Day : Queue
2 Days : Generics
1 Day   : Concurrent Collection (COWAL, COWAS, CHM)
8 Days : Stream API
         (Optional + Method Reference+ New Date and Time API)

Online class : Inner class + Videos (LL)
---   21-01-2025

How to generate OR find out System hashcode value :

System class has provided a predefined native and static method called
identityHashCode(Object obj), It is used to
generate System hashcode.
It accepts Object as a parameter and return type of this method is int.

If we don't override hashCode() method in the corresponding class then System generated
Hashcode and Object class hashcode would be same.

public native static int identityHashCode(Object obj)

Program :
----
package com.nit.collection;

class Employee
{

}

public class SystemHashCode {

```
        public static void main(String[] args)
        {
                String str = "india";

                System.out.println("Hashcode of str from String class :"+str.hashCode());
                System.out.println("System generated Hashcode
:"+System.identityHashCode(str));

                Employee e1 = new Employee();
                System.out.println("Object class hashcode :"+e1.hashCode());
                System.out.println("System generated Hashcode
:"+System.identityHashCode(e1));


        }

}
```

IdentityHashMap<K,V> [Comparing keys based on the Memory reference]
----
public class IdentityHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,
Clonable, Serializable.

It was introduced from JDK 1.4 onwards.

The IdentityHashMap uses == operator to compare keys.

As we know HashMap uses equals() and hashCode() method for comparing the keys based on
the hashcode of the object it will serach the bucket location and insert the entry their only.

So We should use IdentityHashMap where we need to check the reference or memory address
instead of logical equality.

HashMap uses hashCode of the "Object key" to find out the bucket loaction in Hashtable, on the
other hand IdentityHashMap does not use hashCode() method actually It uses
System.identityHashCode(Object o)

IdentityHashMap is more faster than HashMap in case of key Comparison.

The default capacity is 32.

It has three constrcutors, It does not contain loadFactor specific constructor.

1) IdentityHashMap ihm1 = new IdentityHashMap();
   Will create IdentityHashMap with default capacity is 32.

2) IdentityHashMap ihm2 = new IdentityHashMap(int initialCapacity);
   Will create IdentityHashMap with user specified capacity

3) IdentityHashMap ihm3 = new IdentityHashMap(Map map)

```java
package com.nit.collection;

import java.util.HashMap;
import java.util.IdentityHashMap;

public class IdentityHashMapDemo {

        public static void main(String[] args)
        {
                IdentityHashMap<String,Integer> ihm = new IdentityHashMap<>();
                ihm.put("Raj", 123);
                ihm.put(new String("Raj"), 456);
                System.out.println(ihm.size());  //2
                System.out.println(ihm); //{Raj = 123, Raj = 456}


                HashMap<String,Integer> hm = new HashMap<>();
                hm.put("Raj", 123);
                hm.put(new String("Raj"), 456);
                System.out.println(hm.size());  //1
                System.out.println(hm); //{Raj = 456}

        }

}
```

SortedMap<K,V>

It is a predefined interface available in java.util package under Map interface available from JDK 1.2V.

We should use SortedMap interface when we want to insert the key element based on some sorting order i.e the default natural sorting order.

Internally It uses Comparable and Comaprator interfaces.

```
------------------------------------------------------------
TreeMap<K,V>
------------
public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V> ,
Clonable, Serializable
```

It is a predefined class avaialble in java.util package under Map interface available for 1.2V.

It is a sorted map that means it will sort the elements by natural sorting order based on the key or by using Comparator interface as a constructor parameter.

It does not allow non comparable keys.(ClassCastException)

It does not accept null key but null value allowed.(NPE)

It uses Red black tree data structure.

TreeMap implements NavigableMap and NavigableMap extends SortedMap. SortedMap extends Map interface.

TreeMap contains 4 types of Constructors :

1) TreeMap tm1 = new TreeMap(); //creates an empty TreeMap

2) TreeMap tm2 = new TreeMap(Comparator cmp); //user defined soting logic

3) TreeMap tm3 = new TreeMap(Map m); //loose Coupling

4) TreeMap tm4 = new TreeMap(SortedMap m);

```
------------------------------------------------------------
package com.ravi.treemap_demo;

import java.util.TreeMap;

public class TreeMapDemo {

        public static void main(String[] args)
        {
                TreeMap<Integer,String> tm1 = new TreeMap<>();
                tm1.put(3, "Scott"); //compareTo
                tm1.put(9, "Smith");
                tm1.put(1, "Martin");
```

```java
            tm1.put(2, "John");
            tm1.put(4, "Alen");

            tm1.forEach((k,v)-> System.out.println(k+" : "+v));

    }

}
```

Note : put() method, internally uses compareTo() method of Integer class to sort the key object in ascending order

----

```java
package com.ravi.treemap_demo;

import java.util.TreeMap;

public class TreeMapDemo {

    public static void main(String[] args)
    {
            TreeMap<Integer,String> tm1 = new TreeMap<>((i1,i2)-> i2.compareTo(i1));
            tm1.put(3, "Scott"); //compare
            tm1.put(9, "Smith");
            tm1.put(1, "Martin");
            tm1.put(2, "John");
            tm1.put(4, "Alen");

            tm1.forEach((k,v)-> System.out.println(k+" : "+v));

    }

}
```

```java
import java.util.*;
public class TreeMapDemo1
{
    public static void main(String args[])
    {
        TreeMap map = new TreeMap();
        map.put("one","1");
        map.put("two",null);
        map.put("three","3");
                        map.put("four",4);
```

```java
        displayMap(map);

        map.forEach((k, v) -> System.out.println("Key = " + k + ", Value = " + v));

    }
    static void displayMap(TreeMap map)
    {
        Collection c = map.entrySet();   //Set<Map.Entry>

        Iterator i = c.iterator();
        i.forEachRemaining(x -> System.out.println(x));
    }
}
```

```java
package com.ravi.treemap_demo;

import java.util.TreeMap;

record Product(Integer productId, String productName) implements Comparable<Product>
{

        @Override
        public int compareTo(Product p2)
        {
                return this.productId().compareTo(p2.productId());
        }

}


public class TreeMapDemo2 {

        public static void main(String[] args)
        {
          TreeMap<Product,String> map = new TreeMap<>();
          map.put(new Product(333, "Camera"), "Hyderabad");
          map.put(new Product(111, "Mobile"), "Indore");
          map.put(new Product(222, "Laptop"), "Kolkata");
          map.put(new Product(444, "Headphone"), "Mumbai");

          map.forEach((k,v)-> System.out.println("Key is :"+k+" value is :"+v));
```

```
        }

}
------------------------------------------------------------
package com.ravi.treemap_demo;

import java.util.TreeMap;

record Product(Integer productId, String productName)
{

}


public class TreeMapDemo2 {

        public static void main(String[] args)
        {
          TreeMap<Product,String> map = new TreeMap<>((p1, p2)->
p1.productId().compareTo(p2.productId()));
            map.put(new Product(333, "Camera"), "Hyderabad");
            map.put(new Product(111, "Mobile"), "Indore");
            map.put(new Product(222, "Laptop"), "Kolkata");
            map.put(new Product(444, "Headphone"), "Mumbai");

            map.forEach((k,v)-> System.out.println("Key is :"+k+" value is :"+v));

        }

}
------------------------------------------------------------
22-01-2025
-----------
Program on TreeMap Constructor :
----------------------------------
package com.ravi.map;

import java.util.HashMap;
import java.util.SortedMap;
import java.util.TreeMap;

public class TreeMapDemo6 {
```

```java
        public static void main(String[] args)
        {
                SortedMap<String, Integer> map1 = new TreeMap<>();
                map1.put("ravi@gmail.com", 1234);
                map1.put("raj@gmail.com", 4566);
                map1.put("scott@gmail.com", 7788);
                map1.put("aryan@gmail.com", 1010);

                //TreeMap(SortedMap<K,V>)
                TreeMap<String, Integer> map2 = new TreeMap<>(map1);
                System.out.println(map2);

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

                //HashMap to TreeMap
                HashMap<Integer, String> hm1 = new HashMap<>();
                hm1.put(89, "Ravi");
                hm1.put(71, "Scott");
                hm1.put(17, "Smith");
                hm1.put(13, "Martin");

                TreeMap<Integer,String> tm1 = new TreeMap<>(hm1);
                System.out.println(tm1);


        }

}
----------------------------------------------------------------
//Program on TreeMap constructor to provide user-defined sorting
  logic
  package com.ravi.map;

import java.util.TreeMap;

record Student(Integer studentId, String studentName)
{

}

public class TreeMapDemo7 {

        public static void main(String[] args)
```

```
        {
                TreeMap<Student,String> tm1 = new TreeMap<Student,String>((s1,s2)->
s1.studentId().compareTo(s2.studentId()));
                tm1.put(new Student(222, "Aryan"), "Ameerpet");
                tm1.put(new Student(111, "Zuber"), "S.R Nagar");
                System.out.println("Sorted based on the ID :");
                tm1.forEach((k,v)-> System.out.println("Key is :"+k+" value is :"+v));

                TreeMap<Student,String> tm2 = new TreeMap<Student,String>((s1,s2)->
s1.studentName().compareTo(s2.studentName()));
                tm2.put(new Student(222, "Aryan"), "Ameerpet");
                tm2.put(new Student(111, "Zuber"), "S.R Nagar");
                System.out.println("Sorted based on the Name :");
                tm2.forEach((k,v)-> System.out.println("Key is :"+k+" value is :"+v));


        }

}
```

Method of SortedMap interface :

Methods of SortedMap interface :
----
1) firstKey() //first key

2) lastKey() //last key

3) headMap(int keyRange) //less than the specified range

4) tailMap(int keyRange) //equal or greater than the specified range

5) subMap(int startKeyRange, int endKeyRange) //the range of key where startKey will be inclusive and endKey will be exclusive.

return type of headMap(), tailMap() and subMap() return type would be SortedMap(I)

```
import java.util.*;
public class SortedMapMethodDemo
        {
 public static void main(String args[])
        {
```

```java
        SortedMap<Integer,String> map=new TreeMap<>();
         map.put(100,"Amit");
         map.put(101,"Ravi");

         map.put(102,"Vijay");
         map.put(103,"Rahul");

         System.out.println("First Key: "+map.firstKey());  //100
          System.out.println("Last Key "+map.lastKey());   //103
          System.out.println("headMap: "+map.headMap(102));  //100 101
          System.out.println("tailMap: "+map.tailMap(102));  //102 103
          System.out.println("subMap: "+map.subMap(100, 102)); //100 101

      }
 }
```

Assignment for NavigableMap Methods :
----
1) ceilingEntry(K key)
2) ceilingKey(K key)
3) floorEntry(K key)
4) floorKey(K key)
5) higherEntry(K key)
6) higherKey(K key)
7) lowerEntry(K key)
8) lowerKey(K key)

Properties :
----
public class Properties extends Hashtable<K,V>

It is a legacy class and It represents a persistent set of properties.

It is subclass of Hashtable available in java.util package.

It is used to maintain the persistent data in the key-value form. It takes both key and value as a String format.

It is used to load properties file in our java application directly at runtime without compilation/deploymnet.

Constructors :

Commonly we are using this constructor :

    Properties p1 = new Properties();
    Creates an empty property list.

Methods :
----
1) public void load(InputStream stream): Reads a property list (key   and value pair) from the input byte stream.

2) public void load(Reader reader):Reads a property list (key and   value pair) from the Character Oriented stream.

3) Object setProperty(String key, String value) : It Calls the Hashtable method put internally.

4) public String getProperty(String key) :Searches for the property   with the specified key in this property list.

5) public void store(OutputStream out, String comments) : It
    Writes this property list (key and element pairs) in this Properties table to the output stream.

6) public void store(Writer writer, String comments) : It
    Writes this property list (key and element pairs) in this Properties table to the character stream.

//Program on Properties class

In this program, we need to follow the following two steps :

Step 1:

We need to create a properties file with any file name but
extension must be .properties as shown below :

db.properties : {key = value}
----
driverName = oracle.jdbc.driver.OracleDriver
userName = scott
password = tiger

PropertiesDemo1.java

```
---------------------
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

public class PropertiesDemo1 {

        public static void main(String[] args) throws IOException
        {
                FileReader reader = new FileReader("D:\\new\\db.properties");

                Properties p = new Properties();
                p.load(reader);

                String driver = p.getProperty("driverName");
                String userName = p.getProperty("userName");
                String password = p.getProperty("password");

                System.out.println(driver);
                System.out.println(userName);
                System.out.println(password);

        }

}
```

If we make changes in the properties file then directly (without compilation) we can take the the value in our java file so after any modification in the properties file we need not to re-compile/re-deploy our java program.

```
package com.ravi.map;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;

public class PropertiesDemo2
{
        public static void main(String[] args) throws IOException
        {
                String filePath = "D:\\new\\data.properties";
```

```java
            var properties = new Properties();

            var writer = new FileWriter(filePath);
            try(writer)
            {
                    properties.setProperty("book", "Java");
                    properties.setProperty("author", "James");
                    properties.setProperty("price", "1200");

                    properties.store(writer, "Book Properties set");

            }
            catch(Exception e)
            {
                    e.printStackTrace();
            }

            //Reading the data from Properties file

            var reader = new FileReader(filePath);

            try(reader)
            {
                    properties.load(reader);
                    System.out.println("Book Name is "+properties.getProperty("book"));
                    System.out.println("Author Name is "+properties.getProperty("author"));
                    System.out.println("Price Name is "+properties.getProperty("price"));
            }
            catch(Exception e)
            {
                    e.printStackTrace();
            }

        }
}
```

==
24-01-2025

Queue interface :-
----
1) It is sub interface of Collection(I) available from JDK 1.5V hence it support all the methods of Collection interface.

2) It works in FIFO(First In first out)

3) It is an ordered collection.

4) In a queue, insertion is possible from last is called REAR where as deletion is possible from the starting is called FRONT of the queue.

5)In order to support Basis Queue operation, LinkedList class   implements Deque and Deque interface exetnds Queue
interface.

PriorityQueue<E> :
----
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

It is a predefined class in java.util package, available from Jdk 1.5 onwards.

It stores the elements using balanced binary heap tree, meaning the smallest element is at the head of the queue.

The elements of the priority queue are ordered according to their natural ordering (binary heap tree), or by using Comparator provided at queue construction time, depending on which constructor is used.

A priority queue does not permit null elements.

It provides natural sorting order so we can't take non-comparable objects(hetrogeneous types of Object)

The initial capacity of PriorityQueue is 11.

Constructor :
----
1) PriorityQueue pq1 = new PriorityQueue();
   Will create PriorityQueue object with default capacity is 11, Elements will be inserted based on binary heap tree.

2) PriorityQueue pq2 = new PriorityQueue(int initialCapacity);
Will create PriorityQueue with user specified capacity

3) PriorityQueue pq3 = new PriorityQueue(int initialCapacity, Comparator cmp);

Will create PriorityQueue with user specified capacity and own userdefined order.

4) PriorityQueue pq3 = new PriorityQueue(Comparator cmp);
    Will create PriorityQueue with user defined sorting order

5) PriorityQueue pq4 = new PriorityQueue(Collection c);
    Loose coupling

Methods :-
----
public boolean offer(E e) /public boolean add(E e) :- Used to add an element in the Queue

public E poll() :- It is used to fetch the elements from head of the queue, after fetching it will delete the element.

public E peek() :- It is also used to fetch the elements from head of the queue, Unlike poll it will only fetch but not delete the element.

public boolean remove(Object element) :- It is used to remove an element. The return type is boolean.
---import java.util.PriorityQueue;

```java
public class PriorityQueueDemo
{
    public static void main(String[] argv)
    {
        PriorityQueue<Object> pq = new PriorityQueue<>();
        pq.add("Orange");
                    pq.add("Apple");
                    pq.add("Mango");
                    pq.add("Guava");
                    pq.add("Grapes");

                    //pq.add(null); // Inavlid
                    //pq.add(23);    //Invalid

                    System.out.println(pq);

    }
}
```

```java
import java.util.PriorityQueue;
public class PriorityQueueDemo1
{
    public static void main(String[] argv)
    {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
                    pq.add(11);
        pq.add(2);
        pq.add(4);
                    pq.add(6);
        System.out.println(pq);  //2  6  4  11
          }
}
```

```java
import java.util.PriorityQueue;
public class PriorityQueueDemo2
{
    public static void main(String[] argv)
    {
        PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("2");
        pq.add("4");
                    pq.add("6");
        System.out.print(pq.peek() + " ");  //2  2  3  4  4
        pq.offer("1");
                    pq.offer("9");
        pq.add("3");   //       4  6   9

        pq.remove("1");
        System.out.print(pq.poll() + " ");
        if (pq.remove("2"))
                {
           System.out.print(pq.poll() + " ");
                    }
        System.out.println(pq.poll() + " " + pq.peek()+"  "+pq.poll());
         }
}
```

```java
package com.ravi.comparable;

import java.util.Comparator;
import java.util.PriorityQueue;

record Task(String name, int priority)
```

```java
{
}


public class PriorityQueueDemo3 {
    public static void main(String[] args)
    {
        PriorityQueue<Task> taskQueue = new PriorityQueue<>(new Comparator<Task>()
                {
            @Override
            public int compare(Task t1, Task t2)
            {
                return Integer.compare(t1.priority(), t2.priority());
            }
        });

        taskQueue.add(new Task("Submit report", 4));
        taskQueue.add(new Task("Find Bug", 2));
        taskQueue.add(new Task("Write Program", 1));
        taskQueue.add(new Task("Execute Program", 3));

        while (!taskQueue.isEmpty())
        {
            System.out.println("Executing: " + taskQueue.poll());
        }
    }
}
```

Note : For Custom object we are using Comparator interface.

Serialization : If we want to store the Object data in a file then
            it is called Serialization.

De-Serialization :If we want to read the Object data from the file
            thehn it is called De-Serialization.

Volatile :

While working in a multithreaded environment multiple threads can perform read and write operation with common variable (chances of Data inconsistency so use synchronized OR AtomicInteger) concurrently.

In order to store the value temporarly, Every thread is having local cache memory (PC Register) but if we declare a variable with volatile modifier then variable's value is not stored in a thread's local cache; it is always read from the main memory.

So the conclusion is, a volatile variable value is always read from and written directly to the main memory, which ensures that changes made by one thread are visible to all other threads immediately.

```java
package com.nit.testing;

class SharedData
{
    private volatile boolean flag = false;

    public void startThread()
    {
        Thread writer = new Thread(() ->
        {
            try
            {
                Thread.sleep(1000);  //Writer thread will go for 1 sec waiting state
                flag = true;
                System.out.println("Writer thread make the flag value as true");
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        });

        Thread reader = new Thread(() ->
        {
            while (!flag)  //From cache memory still the value of flag is false
            {

            }
            System.out.println("Reader thread got the updated value");
        });

        writer.start();
        reader.start();
    }
```

```
}

public class VolatileExample
{
    public static void main(String[] args)
    {
        new SharedData().startThread();
    }
}
```

27-01-2025

Generics :

What is the need of Generics ?

As we know our compiler is known for Strict type checking because java is a statically typed
checked language.

The basic problem with collection is, It can hold any kind of Object.

```
ArrayList al = new ArrayList();
al.add("Ravi");
al.add("Aswin");
al.add("Rahul");
al.add("Raj");
al.add("Samir");

for(int i =0; i<al.size(); i++)
{
    String s = (String) al.get(i);
    System.out.println(s);
}
```

By looking the above code it is clear that Collection stores everything in the form of Object so
here even after adding String type only we need to provide casting as shown below.

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
        ArrayList al = new ArrayList();
```

```
                al.add(12);
                al.add(15);
                al.add(18);
                al.add(22);
                al.add(24);

                for (int i=0; i<al.size(); i++)
                {
            Integer x = (Integer) al.get(i);
                        System.out.println(x);
                }

    }

}
```

Note : Even we are accepting only Integer type of Object but still type casting is required.

```
import java.util.*;
class Test1
{
        public static void main(String[] args)
        {
                ArrayList al = new ArrayList(); //raw type
                al.add("Ravi");
                al.add("Ajay");
                al.add("Vijay");

                for(int i=0; i<al.size(); i++)
                {
                String name =  (String) al.get(i); //type casting is required
                System.out.println(name.toUpperCase());
                }

        }
}
```

```
import java.util.*;
class Test2
{
        public static void main(String[] args)
        {
                ArrayList t = new ArrayList(); //raw type
```

```
            t.add("alpha");
            t.add("beta");
            for (int i = 0; i < t.size(); i++)
            {
              String str =(String) t.get(i);
               System.out.println(str);
            }

             t.add(1234);
             t.add(1256);
              for (int i = 0; i < t.size(); ++i)
             {
                     String obj= (String)t.get(i); //we can't perform type casting here
                     System.out.println(obj);
             }
         }
}
```

Even after type casting there is no guarantee that the things which are coming from ArrayList Object is String only because we can add anything in the Collection as a result java.lang.ClassCastException
----
To avoid all the above said problem Generics came into picture from JDK 1.5 onwards

 -> It deals with type safe Object so there is a guarantee of both the end i.e putting inside and getting outside.

 Example:-
 ArrayList<String > al = new ArrayList<>();

 Now here we have a guarantee that only String can be inserted as well as only String will come out from the Collection so we can perform String related operation.

Advantages of Generics :
----
 1) Type Safe Object (No Compilation warning)
 2) No need of type casting
 3) Strict compile time checking. (*Type Erasure)
 ----
 import java.util.*;
public class Test3
```
```

```java
{
public static void main(String[] args)
{
                ArrayList<String> al = new ArrayList<>();  //Generic type
                al.add("Ravi");
                al.add("Ajay");
                al.add("Vijay");

        for(int i=0; i<al.size(); i++)
                {
                String name = al.get(i); //no type casting is required
                System.out.println(name.toUpperCase());
                }
  }
}
```

//Program that describes the return type of any method can be type safe, We can apply generics on method return type]

```java
package com.ravi.generics;

import java.util.ArrayList;
import java.util.List;

class Dog
{
        public List<Dog> getDogList()
        {
                ArrayList<Dog> d = new ArrayList<>();
        d.add(new Dog());
                d.add(new Dog());
                d.add(new Dog());
                return d;
        }
}

public class Test4
{
        public static void main(String[] args)
        {
          Dog d1 = new Dog();
          Dog dog = d1.getDogList().get(1);
          System.out.println(dog);
```

```
        }

}
```

Note :- In the above program the compiler will stop us from returning anything which is not compaitable List<Dog> and there is a guarantee that only "type safe list of Dog object" will be returned so we need not to provide type casting as shown below
Dog d2 = (Dog) d1.getDogList().get(0); //before generic.

//Mixing generic with non-generic

//Mixing generic with non-generic
import java.util.*;

```java
class Car
{
}
public class Test5
{
        public static void main(String [] args)
        {
        ArrayList<Car> a = new ArrayList<>();
        a.add(new Car());
        a.add(new Car());
    a.add(new Car());

        ArrayList b = a;  //assigning Generic to raw type
    System.out.println(b);
        }
}
```

//Mixing generic to non-generic
import java.util.*;
```java
public class Test6
{
        public static void main(String[] args)
        {
                List<Integer> myList = new ArrayList<>();
                myList.add(4);
                myList.add(6);
                myList.add(5);
```

```java
            UnknownClass u = new UnknownClass();
            int total = u.addValues(myList);
            System.out.println("The sum of Integer Object is :"+total);
        }
}
class UnknownClass
{
        public int addValues(List list)  //generic to raw type
        {
        Iterator it = list.iterator();
        int total = 0;
        while (it.hasNext())
        {
                int i = ((Integer)it.next());
                total += i;                    //total =  15
        }
        return total;
        }
}
```

Note :-
In the above program the compiler will not generate any warning message because even though we are assigning type safe Integer Object to unsafe or raw type List Object but this List Object is not inserting anything new in the collection so there is no risk to the caller.

```java
//Mixing generic to non-generic
import java.util.*;
public class Test7
{
        public static void main(String[] args)
        {
                List<Integer> myList = new ArrayList<>();

                myList.add(4);
                myList.add(6);
                UnknownClass u = new UnknownClass();
                int total = u.addValues(myList);
                System.out.println(total);
        }
}
class UnknownClass
{
    public int addValues(List list)
```

```
        {
                list.add(5);        //adding object to raw type
                Iterator it = list.iterator();
                int total = 0;
                while (it.hasNext())
                {
                int i = ((Integer)it.next());
                total += i;
                }
                return total;
        }
}
```

Here Compiler will generate warning message because the unsafe object is inserting the value 5 to safe object.
----

*Type Erasure

In the above program the compiler will generate warning message because the unsafe List Object is inserting the Integer object 5 so the type safe Integer object is getting value 5 from unsafe type so there is a problem to the caller method.

By writing ArrayList<Integer> actually JVM does not have any idea that our ArrayList was suppose to hold only Integers.

All the type safe generics information does not exist at runtime. All our generic code is Strictly for compiler.

There is a process done by java compiler called "Type erasure" in which the java compiler converts generic version to non-generic type.

List<Integer> myList = new ArrayList<Integer>();

At the compilation time it is fine but at runtime for JVM the code becomes

List myList = new ArrayList();

Note :- GENERIC IS STRICTLY COMPILE TIME PROTECTION.

import java.util.*;
public class TypeErasureDemo

```java
{
        public static void main(String[] args)
        {

        }

        public void accept(List<String> listOfString)
        {
        }

        public void accept(List<Integer> listOfInteger)
        {
        }
}
```

Note : In the above program we will get compilation error because
       generic information does not exist at runtime so method overloading is not possible.
-- //Polymorphism with array

//Polymorphism with array

```java
import java.util.*;
abstract class Animal
{
        public abstract void checkup();
}

class Dog extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Dog checkup");
        }
}

class Cat extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Cat checkup");
        }
```

```java
}

class Bird extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Bird checkup");
        }
}

public class  Test8
{
        public static void checkAnimals(Animal ...animals)
        {
                for(Animal animal : animals)
                {
                        animal.checkup();
                }
        }

        public static void main(String[] args)
        {
                Dog []dogs={new Dog(), new Dog()};

                Cat []cats={new Cat(), new Cat(), new Cat()};

                Bird []birds = {new Bird(), new Bird()};


                checkAnimals(dogs);
                checkAnimals(cats);
                checkAnimals(birds);
        }
}
```

Note :-From the above program it is clear that polymorphism(Upcasting) concept works with array.

```java
import java.util.*;
abstract class Animal
{
        public abstract void checkup();
```

```java
}

class Dog extends Animal
{
    @Override
        public void checkup()
        {
                System.out.println("Dog checkup");
        }
}

class Cat extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Cat checkup");
        }
}
class Bird extends Animal
{
        @Override
        public void checkup()
        {
                System.out.println("Bird checkup");
        }
}
public class Test9
{

        public void checkAnimals(List<Animal> animals)
        {
                for(Animal animal : animals)
                {
         animal.checkup();
                }
        }
        public static void main(String[] args)
        {
                List<Dog> dogs = new ArrayList<>();
                dogs.add(new Dog());
                dogs.add(new Dog());
```

```
            List<Cat> cats = new ArrayList<>();
            cats.add(new Cat());
            cats.add(new Cat());

            List<Bird> birds = new ArrayList<>();
            birds.add(new Bird());
            birds.add(new Bird());

            Test9 t = new Test9();
            t.checkAnimals(dogs);
            t.checkAnimals(cats);
            t.checkAnimals(birds);


        }
}
```

Note :- The above program will generate compilation error.

So from the above program it is clear that polymorphism does not work in the same way for generics as it does with arrays.

Example :

Parent [] arr = new Child[5]; //valid
Object [] arr = new String[5]; //valid

But in generics the same type is not valid

List<Object> list = new ArrayList<Integer>(); //Invalid
List<Parent> mylist = new ArrayList<Child>(); //Invalid

```
import java.util.*;
public class Test10
{
public static void main(String [] args)
        {
            /*ArrayList<Object> al = new ArrayList<String>(); [Compile time]
            ArrayList al = new ArrayList();  [Runtime, Type Erasure]
            al.add("Ravi");*/


                Object []obj = new String[3]; //valid with Array
```

```
            obj[0] = "Ravi";
            obj[1] = "hyd";
            obj[2] =  90; //java.lang.ArrayStoreException
            System.out.println(Arrays.toString(obj));
        }
}
```

Note :- Program will generate java.lang.ArrayStoreException because we are trying to insert 90 (integer value) into String array.

In Array we have an Exception called ArrayStoreException (Which protect us to assign some illegal value in the array) but the same Exception or such type of exception, is not available with Generics (due to Type Erasure) that is the reason in generics, compiler does not allow upcasting concept.
(It is a strict compile time protection)

WildCard character (?)

<?> : Unknown type (All posibilities)

<Dog> : Only we can assign Dog.

<Animal> : Only we can assign Animal.

<? extends Animal> : Upper Bound [Animal and all the classes
                         which are subclasses of Animal are allowed]
                                Here in future, there is a chance of inserting Wrong
                                collection because Animal can have more sub classes (chances
of Wrong collection)

<? super Dog> : Lower Bound [Only Dog and Super classes of
                  Dog i.e Animal and Object are allowed]

----
import java.util.*;
class Parent
{
}
class Child extends Parent
{
}
```

```java
public class Test11
{
public static void main(String [] args)
        {
                //ArrayList<Parent> lp = new ArrayList<Child>(); //error

                ArrayList<?> lp = new ArrayList<Child>(); //error

                ArrayList<Parent> lp1 = new ArrayList<Parent>();

                ArrayList<Child> lp2 = new ArrayList<>();

                System.out.println("Success");
        }
}
```

```java
//program on wild-card chracter
import java.util.*;
class Parent
{

}
class Child extends Parent
{
}
public class Test12
{
public static void main(String [] args)
        {
                List<?> lp = new ArrayList<Parent>();
                System.out.println("Wild card....");
        }
}
```

```java
import java.util.*;
public class Test13
{
        public static void main(String[] args)
        {
                List<? extends Number> list1 = new ArrayList<Float>();

                List<? super String> list2 = new ArrayList<Object>();
```

```java
                List<? super Gamma> list3 = new ArrayList<Alpha>();

                List list4 = new ArrayList();

                System.out.println("yes");
        }
}

class Alpha
{
}
class Beta extends Alpha
{
}
class Gamma extends Beta
{
}
------------------------------------------------------------
class MyClass<T>
{
        T obj;
        public MyClass(T obj)      //Student obj
        {
                this.obj=obj;
        }

        T getObj()
        {
                return this.obj;
        }
}
public class Test14
{
        public static void main(String[] args)
        {
                Integer i=12;
                MyClass<Integer> mi = new MyClass<>(i);
                System.out.println("Integer object stored :"+mi.getObj());

                Float f=12.34f;
                MyClass<Float> mf = new MyClass<>(f);
                System.out.println("Float object stored :"+mf.getObj());
```

```
                MyClass<String> ms = new MyClass<>("Rahul");
                System.out.println("String object stored :"+ms.getObj());

                MyClass<Boolean> mb = new MyClass<>(false);
                System.out.println("Boolean object stored :"+mb.getObj());

                Double d=99.34;
                MyClass<Double> md = new MyClass<>(d);
                System.out.println("Double object stored :"+md.getObj());

                MyClass<Student> std = new MyClass<>(new Student(1,"A"));
                System.out.println("Student object stored :"+std.getObj());
        }
}

record Student(int id, String name)
{

}
-----------------------------------------------------------------
package com.ravi.generics;

class Basket<E>  //E is of type Fruit
{
        private E element;   //Fruit element =

                public Basket(E element)  //Fruit element = new Apple();
                {
                        super();
                        this.element = element;
                }

                public E getElement()
                {
                        return element;
                }
}

class Fruit
{

}
```

```java
class Apple extends Fruit
{
        @Override
        public String toString()
        {
                return "Apple";
        }
}
class Orange extends Fruit
{
        @Override
        public String toString()
        {
                return "Orange";
        }
}


public class Test15 {

        public static void main(String[] args)
        {
                Basket<Fruit> basket = new Basket<>(new Apple());
        Apple apple = (Apple) basket.getElement();
        System.out.println(apple);


        basket = new Basket<>(new Orange());
        Orange orange = (Orange) basket.getElement();
        System.out.println(orange);
            }

}
```

29-01-2025

How to Develop our own functional interfaces :

```java
package com.ravi.custom_functionalInterface;

@FunctionalInterface
interface TriFunction<T,U,V,R>
{
```

```java
        R apply(T a, U b, V c);
}

public class CustomFunctionalInterface
{
        public static void main(String[] args)
        {
                TriFunction<Integer, Integer, Integer, String> fn1 = (a,b,c)-> ""+a+b+c;
                System.out.println("Concatenation is :"+fn1.apply(100, 200, 300));
        }

}
```

//Generic Method

```java
public class Test16
{
        public static void main(String[] args)
        {
                Integer []intArr = {10,20,30,40,50};
                printArray(intArr);

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

                String []cities = {"Hyderabad", "Banglore", "Mumbai", "Kolkata"};
                printArray(cities);

        }

        public static <T> void printArray(T[] array)
        {
                for(T element : array )
                {
                        System.out.println(element);
                }
        }

}
```

Stream API :