28-09-2024
---
javaravishanker@gmail.com
9835647014

History of Java :

Java was developed by Sun microsystem but on 27th January 2010, Java was overtaken by Oracle Corporation so now java is the product of
Oracle Corporation.

The first version of java was released on 23rd January 1996.

Fisrt Name of Java : OAK (Tree Name)
Developed By : James Gosling and his friends
Project Name : Green Project
First release : 23rd Jan 1996 (JDK 1.0)
Java : It is an island in indonesia
Official Symbol : Coffee Cup

What is a function ?

A function is a self defined block which is used for performing some opertion, calculation or printing the data.

Function can be divided into two types :
---
1) Predefined OR Built-in function :

The function which is written by language createor OR user for some specific purpose is called predefined function.

2) User-defined Function :

The functions which are defined by user for performing some specific task are called User-defined function.

Advantages of Function :
---
1) Modularity :

Dividing the bigger task into number of smaller task.

2) Easy to understand :

Once the task is divided into number of independent modules then it is easy to understand the entire module.

3) Reusability :

We can reuse a particular module for ’n’ number of times.

Note :In java, we always reuse our java classes.

4) Easy debugging :

Each module is isolated from another module so the debugging is easy because we can debug only one module where we have syntax or semantics error.

==========
Why we pass parameter to a function ?

We should pass parameter to a function for getting more information regarding the function.

If We don’t pass parameter then the informations are not complete, It is partial information.

Example :

```
public void deposit(double amount)
{
}

public void doSum(int x, int y)
{
}

public void sleep(int hours)
{
}
```

==========
Why functions are called Method in java ?

In C++ language, there is a facility to write a function inside the class as well outside of the class by using scope resolution operator (::) but in java we can write a function inside the class only, we can’t define a function outside of the class, that is reason functions are called Method in java.

30-09-2024

**What is platform independency in java ?

C and C++ programs are platform dependent programs that means the .exe file created on one machine will not be executed on the another machine if the system configuration is different.

That is the reason C and C++ programs are not suitable for website development.

The role of java compiler :
---
1) Syntax verification.
2) Verify the compatibility issues (L.H.S = R.H.S)
3) Will Convert Source code into byte code.

Java is a platform independent language. Whenever we write a java program, the extension of java program must be .java.

Now this .java file we submit to java compiler (javac) for compilation process. After successful compilation the compiler will generate a very special machine code file i.e .class file (also known as bytecode). Now

this .class file we submit to JVM for execution purpose.

The role of JVM is to load and execute the .class file. Here JVM plays a major role because It converts the .class file into appropriate machine code instruction (Operating System format) so java becomes platform independent language and it is highly suitable for website development.[30-SEP-24]

Note :- We have different JVM for different Operating System that means JVM is platform dependent technology where as Java is platform Independent technology.

JVM internally contains an interpreter so it executes the code line by line. It is written in 'C'langugae hance platform dependent.

**What is the difference between bit code and byte code ?

Bit code is directly understood by Operating System but on the other hand byte code is understood by JVM, JVM is going to convert this byte code into appropriate machine understandable format.
[30-SEP-24]

Note : All the browsers internally contain JVM are known as
JEB (Java Enabled Browsers) browser.

01-10-2024

Difference between Compiler (javac) and Interpreter (JVM)

Paint Diagram [01-OCT]

*Difference between JDK, JRE, JVM and JIT Compiler :

Paint Diagram [01-OCT]

JDK :

It stands for Java Devleopment Kit. It is a developer version that
means by using JDK we can develop as well as execute our java
programs.
In order to develop and execute it supports various JDK tools which are as follows :

a) javac : java compiler, responsible for compilation.
b) java : Java launcher,responsible for executing java program.
c) jdb : java debugger, for debugging purpose
d) jconsole : java Console, to display the output in the console.
e) javap : java Profiler, To get the details of a class
f) javadoc : Java documention, for Generating java documentation.

JRE :

It stands for Java Runtime Environment. It is a client version so
by using JRE we can only execute our java program.

From java 11 version we don't have separate JRE folder, java software people removed this folder from

software so, from java 11 version we can directly execute our java program without compilation.

java FileName.java [This is the command to directly execute our
java code]

JVM :

The main purpose of JVM to load and execute the byte code. It provides, security, memory management by garbage collector, JIT compiler for fast execution and so on.

The .class file generated by java compiler first verified by ByteCode Verifier (One component of JVM) so java is the most
secure language in IT market.

What is JIT Compiler :

As we know, Our interpreter is slow in nature so to boost up the java execution, we have JIT (Just In time) compiler support.

It holds the repeated code instrauction and native code instruction, It will directly provide these two instrution at time
of line by line execution so our interpreter executes the code
in more efficient way hance the overall execution becomes very fast.

What is data type in java ?

A data type describes, what type of value the varaiable will hold.
In hava we have 2 types of data types :

Data type diagram is available in paint diagram [02-OCT]

What is the difference between statically typed and Dynamically typed language?

Statically Typed Language :

The languages where data type is compulsory before initialization of a variable are called statically typed language.
In these languages we can hold same kind of value during the execution of the program.

Ex:- C,C++,Core Java, C#

Dynamically Typed Langauge :

The languages where data type is not compulsory and it is optional before initialization of a variable then it is called dynamically typed language.

In these languages we can hold different kind of value during the execution of the program.
Ex:- Visual Basic, Javascript, Python

What is comments in java ?

Comments are used to enhance the readability of the code. It is ignored by the compiler.

In java we have 3 types of commants :

1) Single Line Comment
   //

2) Multiline Comment
   /*
     Java Source code.
   */

3) Documentation Comment
   /**
      Name of the Project : Online Shopping
        Number of Modules : 36
 Date of creation : 2nd Feb 2024
 Last Modification : 30th Sep 2024
            Author : Green Team
   */

WAP in Java to display welcome message :

```
public class Welcome
{
       public static void main(String[] args)
 {
  System.out.println("Welcome Batch 39!");
 }

}
```

Description of main() method :

---

public :-

---

public is an access modifier in java. The main method must be declared as public otherwise JVM cannot execute our main method or in other words JVM can't enter inside the main method for execution of the program.

If main method is not declared as public then program will compile but it will not be executed by JVM.

Note :- From java compiler point of view there is no rule (syntax rule) to declare our methods as public.

---

static :

As of now, In java we have 2 types of Methods :

 a) static method : Declared with static keyword
 b) non static method : Not declared with static keyword.

 static method (OBJECT IS NOT REQD) :

 Case 1:

Any static method we can call with the help of class name if the static method is declared in another class, Here Object is not
required.

```
public class Demo
{
public static void main(String[] args)
{
 System.out.println("Hello");
 Sample.greet();
}
}

class Sample
{
public static void greet()
{
 System.out.println("Good Morning All");
}
}
```

Case 2 :

If a static method is declared in the same class where main method is available then we can call the static method directly, Here class name is also not required.

```
public class Welcome
{
public static void main(String[] args)
{
 System.out.println("Welcome Batch 39!");
 m1();
}

public static void m1()
{
 System.out.println("Hi I am m1 method");
}

}
```

Our main method must be declared as static so object is not required, JVM can call this main method with the help of class name.
If we don't declare our main method as static then code will compile but it will not be executed by JVM.

03-10-2024
---
void :-

It is a keyword. It means no return type. Whenever we define any method in java and if we don't want to return any kind of value from that particular method then we should write void before the name of the method.

Eg:

```
public void input()                    public int accept()
{                                  {
}                                      return 15;
        }
```

Note :- In the main method if we don't write void or any other kind of return type then it will generate a compilation error.

In java whenever we define a method then compulsory we should define return type of method.(Syntax rule)

main method return type must be void because JVM will not accept any return value from the user.

---

main() method :

It is a user-defined method because a user is responsible to define some logic inside the main method.

main() method is very important method because every program execution will start from main() method only, as well as the execution of the program ends with main() method only.

Q) Can we write multiple method with same name ?

Yes, We can write multiple methods with same name but parameter must be different otherwise code will not compile.

Note :- We can also write multiple main methods with different parameter but JVM will always execute the main method which takes String [] args (String array) as a parameter as shown in the program below.

```
public class Welcome
{
 public static void main(String[] args)
 {
   System.out.println("String array Variable");
 }

 public static void main(int x)
 {
   System.out.println("int Ordinary Variable ");
 }
}
```

Output is : String array Variable

String [] args :

String is a predefined class in java available in java.lang package (just like header file) and args is an array variable
of type String so, it can hold multiple values.

IQ :

Why the main method of java accepts String array as a parameter ?

String is a collection of alpha-numeric character so it can accept all different kind of values. Java software people has

provided String array as a parameter so it can ACCEPT MULTIPLE
VALUES OF DIFFERENT TYPE, that means providing more wider scope to accept hetrogeneous types of values.

System.out.println() :

It is an output statement in java, By using this statement we can print different types of values on the console.

In this statement System is a predefined class available in java.lang pacakage, out is a reference variable of type PrintStream class available in java.io package and println() is
a predefined method available in PrintStream class.

Note :Actually It is HAS-A relation concept, System class has
PrintStream class as shown below.

```
class System
{
  final static PrintStream out;  //HAS-A Relation
}
```

==========

WAP in java to add two numbers :

```
public class Addition
{
 public static void main(String[] args)
 {
  int x = 10;
  int y = 20;
  int z = x + y;
  System.out.println(z);
 }
}
```

Here we are getting the output as 30 but it is not user-friendly message.

How to provide user-friendly message :

In order to provide user-friendly message to the end user we should use ’+’ operator i.e string concatenation operator.

Program

```
public class AdditionWithMessage
{
 public static void main(String[] args)
 {
  int x = 10;
  int y = 20;
  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)); //Sum is 300
 }
}
```

04-10-2024

IQ
--
```
public class StringConcatenationDemo
{
 public static void main(String[] args)
 {
  String str = 25 + 25 +"NIT"+ 50 + 50;
  System.out.println(str);
 }
}
```

Command Line Argument :

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

Example :

```
public static void main(String [] args) //Command Line Argument
{
}
```

By using comnmand line Argument, We can pass some value at runtime.

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

==========
```
//Program to accept the value from command line Argument
public class Command
{
 public static void main(String[] args)
 {
  System.out.println(args[0]);
 }
}
```

```
javac Command.java
java Command Scott Smith
Output is : Scott
```

```
//Program to pass some numberic value as a String value
public class CommandValue
{
 public static void main(String[] cmd)
 {
  System.out.println(cmd[1]);
 }
}

javac CommandValue.java
java CommandValue 100 200
Output is : 200
```

```
//Accepting the full name from command Line Argument

public class FullNameUsingCommand
{
 public static void main(String[] name)
 {
  System.out.println(name[0]);
 }
}

javac FullNameUsingCommand.java
java FullNameUsingCommand "Virat Kohli"

Output : Virat Kohli
```

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

javac Command.java
java Command        [Not passing any value at runtime]

While working with command line argument, if we are using the index in the program but not passing any value at runtime to
command line argument then we will get an exception
java.lang.ArrayIndexOutOfBoundsException.
```

How to find out the length of an array variable ?

In order to find out the length of an array variable, Arrays class has provided a predefined variable OR property called length as shown in the programs below.

```
//Finding the length of an array
public class ArrayLength
{
 public static void main(String[] args)
 {
```

```
  int []arr = {10, 20,30, 40};
  System.out.println(arr.length);
 }
}
```

//Finding the length of an array by using Command Line Argument

```
public class ArrayLengthUsingCommand
{
 public static void main(String[] args)
 {
  System.out.println("The length of array is :"+args.length);
 }
}
```

javac ArrayLengthUsingCommand.java
java ArrayLengthUsingCommand
Output : The length of array is 0
java ArrayLengthUsingCommand 12
Output : The length of array is 1
java ArrayLengthUsingCommand 12 14
Output : The length of array is 2

//WAP to add two numbers by using Command Line Argument

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

javac CommandAdd.java
java CommandAdd 100 200
Output is : 100 200 [Here '+' works as String Concatenation Optr]

==========

How to convert a String into integer value :

There is a predefined class called Integer available in java.lang pacakge, It provides a predefined static method called parseInt(String x) which accepts a single String type parameter and convert this String into int type becauase the return type of this parseInt(String x) method is int type.

```
public class CommandAddition
{
 public static void main(String[] num)
 {
  //Converting String to integer
  int a = Integer.parseInt(num[0]);
  int b = Integer.parseInt(num[1]);

  int sum = a + b;

  System.out.println("The Sum is :"+sum);
```

```
 }
}
```

javac CommandAddition.java
java CommandAddition 12 12

Output : The Sum is 24

Converting String to float

```
float f = Float.parseFloat(String x);
```

Converting String to double :

```
double d = Double.parseDouble(String x);
```

---05-10-2024

WAP to find out the square of the number by using Command line Argument.

```
public class FindSquare
{
 public static void main(String[] args)
 {
  int num = Integer.parseInt(args[0]);
  System.out.println("Square of "+num+" is :"+(num*num));
 }
}
```

javac FindSquare.java
java FindSquare 12

output is : Square of 12 is 144

Eclipse IDE :

IDE stands for "Integrated Development Environment". By using eclipse IDE, we can develop, compile and execute our java program in a single window.

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

How to create a Project in Eclipse IDE :

File -> new -> Project OR Java Project -> Provide the name for
the project (Batch39) -> Finish

What is a Package in java :

A package is nothing but folder in windows. It is used to arrange the classes and interfaces into a particular group.

If we arrange our java classes into a particular group by using pacakges (folders) then we will get the following two advantages :

1) Fast searching is possible.

2) Name can be reusable.

Program that describes a package is folder in windows :

A package is a keyword in java and it must be the first statement of any java program.

```
package sum;
public class Addition
{

}
```

Command for compilation of the classes which contains Package statement.

[javac -d . FileName.java]

javac -d . Addition.java

It will compile Addition.java, Addition.java contains sum package, one package i.e folder called sum will be created and automatically Addition.class file will be placed inside the package or folder called sum.

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;

WAP in eclipse IDE for finding the area of the circle :

```
package com.ravi.command_line_Argument;

public class FindingAreaOfCircle {

 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);


 }

}
```

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

---

WAP to find out the area of rectangle :

```
package com.ravi.command_line_Argument;

public class AreaOfRectangle {

 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 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

```
package com.ravi.command_line_Argument;

public class ArrayCalculationOnLength {

 public static void main(String[] args)
 {
  if(args.length==0)
  {
   System.err.println("Length is 0");
  }
  else if(args.length == 1)
  {
   int num = Integer.parseInt(args[0]);
   System.out.println("Cube of "+num+" is :"+(num*num*num));
  }
  else if(args.length ==2)
  {
   int a = Integer.parseInt(args[0]);
   int b = Integer.parseInt(args[1]);
   int sum = a + b;
   System.out.println("Sum is :"+sum);
  }

 }
```

```
}
```

WAP to show how exactly Integer.parseInt works internally ?

```
package com.ravi.command_line_Argument;

class Integer
{
 public static int getSquare(int num)
 {
  return num*num;
 }

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

public class IntegerClassDemo
{
 public static void main(String[] args)
 {
  int square = Integer.getSquare(10);
        System.out.println("Squre is :"+square);

        int cube = Integer.getCube(4);
        System.out.println("Cube is :"+cube);
 }

}
```

07-10-2024

Naming convention in java ?

Naming convention provides two important characteristics :

 a) Standard Code (Industry accepted code)
 b) Readability of the code will enhance.

1) How to write a class in java :

While writing a class in java, we should follow pascal naming convention, According to this each word first character must be capital and it should not contain any space. In java a class represents noun.

Example :
ThisIsExampleOfClass

System
String
Integer
CommandAddition
ArrayIndexOutOfBoundsException
DataInputStream.

2) How to write a method in java :

While writing a method in java we should follow camel case
naming convention, According to this naming convention first world will be in small and 2nd word onwards, each word first character must be capital. In java a method represents verb.

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

3) How to write a field/variable in java :

While writing a variable we should follow camel case naming convention but unlike method variable does not have () symbol.

Example :
---
rollNumber
customerName
customerBill
studentName
playerName

4) How to write a final and static variable :

While writing the final and static variable we should follow snake_case naming convention.

Example :
Integer.MIN_VALUE [MIN_VALUE is final and static variable]
Integer.MAX_VALUE [MAX_VALUE is final and static variable]

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 switchOn()
   {
   }
}
```

Here Fan(Name of the class), coil (Name of the variable) and switchOn(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 :

Any constant which we are assiging to variable is called Literal.

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 a literal

Integral Literal :

If any numberic literal does not contain decimal or fraction
then it is called Integral Literal.
Example : 12, 89, 45

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 four different forms

1) Decimal Literal (Base 10)
2) Octal Literal (Base 8)
3) Hexadecimal Literal (Base 16)
4) Binary Literal (Base 2) [Available from JDK 1.7v]

Decimal Literal :

By default our numeric literals are decimal literal. Here base is 10 so, It accepts 10 digits i.e. from 0-9.

Example :

```
int x = 20;
int y = 123;
int z = 234;
```

Octal Literal :

If any Integer literal starts with 0 (Zero) then it will become octal literal. Here base is 8 so it will accept 8 digits i.e 0 to 7.

Example :

```
int a = 018; //Invalid becuase it contains digit 8 which is out of range
int b = 0777; //Valid
int c = 0123; //Valid
```

Hexadecimal Literal :

If any integric literal starts with 0X or 0x (Zero capital X Or 0 small x) then it is hexadecimal literal. Here base is 16 so it will accept 16 digits i.e 0 to 9 and A to F OR [a to f]

Example :

```
int x = 0X12; //Valid
int y = 0xadd; //Valid
int z = 0Xface; //valid
```

int a = 0Xage; //Inavlid because character ’g’ out of range

Binary Literal :

If a numeric literal starts with 0B (Zero capital B) or 0b (Zero small b) then it will become Binary literal.
Binary literal is available from JDK 1.7v.
Here base is 2 so it will accept 2 digits i.e 0 and 1.

Example :
---
int x = 0B101; //valid
int y = 0b111; //Valid
int z = 0B112; //Invalis [2 is out of range]

The deafult type is decimal literal so to generate the output for any different literal JVM converts into decimal literal.

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

```
//Hexadecimal Litear
public class HexadecimalDemo
{
 public static void main(String[] args)
 {
  int a = 0xadd;
  System.out.println(a);//2781
 }
}
```

```
//Binary Literal
public class BinaryDemo
{
  public static void main(String [] args)
  {
     int x = 0B101;
  System.out.println(x); //5

  }
}
```

08-10-2024

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; //error
    System.out.println(b);

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

```
//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 Narrowing]
  System.out.println(b);
 }
}
```

```
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
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);
 }
}
```

09-10-2024

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)

```
//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);

 }
}
```

Providing _ (underscore) in integeral Literal :

In Order to enhance the readability of large numeric literals, Java software people has provided _ (underscore) 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;  //instead of const we should use final
  byte b = x;
  System.out.println(b);
 }
}
```

Note: It is a valid program will generate the output.

---
var keyword in java :
---
var keyword is introduced from java 10v.
It can be used inside the method only.
It must be initialized in the same line where we are declaring the variable with var keyword.
It is also known as local variable type inference.

```
//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 Test12
{
     public static void main(String[] argv)
```

```
    {
    //decimal to Binary
        System.out.println(Integer.toBinaryString(7)); //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 any numeric literal contains decimal or fraction then it is called floating point literal.
Example : 12.3, 90.7, 56.6

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 generate compilation error.

```
float f1 = 1.2;  //Invalid
```

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

```
double d1 = 12D;
double d2 = 78d;
```

A floating point literal we can represnt in exponent form.

```
double d1 = 15e2;  [15 X 10 to the power of 2]
```

* An integral literal we can represent in four different forms i.e decimal, octal, hexadecimal and binary but floating point literal we can represent in only one form i.e decimal.

* An integral literal i.e byte, short, int and long we can assign to floating point literal but floating point literal we can’t assign to integral literal.

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

```
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);

 }
}
```

```
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);
 }
}
```

```
public class Test3
{
 public static void main(String[] args)
 {
   double x = 0129.89;

   double y = 0167;

   double z = 0178;

  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; //error
    float c = 1.5e3F;
    double d = 10;
    int e = 10.0; //error
    long f = 10D; //error
    int g = 10F;   //error
    long l = 12.78F; //error
 }
}
```

```
//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);
 }
}
```

14-10-2024

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

//Programs :

```
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; //Invalid
      boolean d = 1; //Invalid
      System.out.println(c);
      System.out.println(d);
   }
}
```

```
public class Test3
{
 public static void main(String[] args)
 {
  boolean x = "true";
```

```
  boolean y = "false";
  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 Test1
{
 public static void main(String[] args)
```

```
 {
  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 = 45789;
        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);
 System.out.println(’A’+’A’);
   }
}
```

```
//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
  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);

  int ch2 = ’\uffff’;
  System.out.println(ch2);

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

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

```
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);
 }
}
```

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

```
//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

 }
}
```

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, String can be created by using 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);

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

 }
}
```

```
//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);
 }
}
```

```
//IQ
public class StringTest3
{
 public static void main(String []args)
 {
  String s = 15+29+"Ravi"+40+40;
  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]

15-10-2024

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

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

```
 }
}
```

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

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

```
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.

Local Variable in java ?

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 because local variable does not have default values.

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

As far as it's accessibility is concerned, It is accessible within the same method only.

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

```
 public static void main(String[] args)
 {
 System.out.println(x); //error
 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, Every methods are executed in a special memory called
Stack Memory.

Stack Memory works on LIFO (Last In First Out) basis.

In java, Whenever we call a method then a separate Stack Frame will be created for each and every method.[15-OCT]

Once the method execution is over then the corresponding method
Stack frame will also be deleted from Stack Area, that is the reason we can’t use local variable outside of the method.

Each stack frame contains 3 parts :
1) Local Variable Array
2) Frame Data
3) Operand Stack

Program that describes method Execution in Stack memory :

```
package com.ravi.method_demo;

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()
 {
  System.out.println("Hii I am m2 method");
 }
}
```

Limitation of Commnd 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.

```
package com.ravi.method_demo;

public class CommandLimitation
{
 public static void main(String[] args)
 {
  System.out.print("Enter your Gender :");
  char gender = args[0].charAt(0);
  System.out.println("Your Gender is :"+gender);

 }

}
```

Note : In the above program, after providing the gender value, It is asking for Gender which is not a recommended way.

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 the End use with user friendly mesasage :

There are so many ways to read the data from end user which are as follows :

1) java.io.DataInputStream
2) java.io.BufferedReader
3) System.in.read();
4) java.io.Console
5) java.util.Sacnner

How to read the data by using java.util.Scanner class :

---

java.util.Scanner is a predefined class to read the data from the clinet with user-friendly message available from JDK 1.5v.

static variables of System class :

System class has provided 3 final and static variables which are as follows :

1) System.out : Used to print normal message.

2) System.err : Used to print error message.

3) System.in : Used to read the data from the Source.

How to create the Object for Scanner class :

Scanner sc = new Scanner(System.in);

Methods of Scanner class :
---
1) public String next() : Used to read a Single Word.

2) public String nextLine() : Used to read multiple words or
complete line.

3) public byte nextByte() : Used to read byte value.

4) public short nextShort() : Used to read short value.

5) public int nextInt() : Used to read int value.

6) public long nextLong() : Used to read long value.

7) public float nextFloat() : Used to read float value.

8) public double nextDouble() : Used to read double value.

9) public boolean nextBoolean() : Used to read boolean value.

10) public char next().charAt(0) : Used to read character value
---
16-10-2024

WAP to read a name from the Scanner class.

```
import java.util.Scanner;
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);
 }
}
```

//WAP to read a character from Scanner class

```
package com.ravi.scanner_demo;
```

```
import java.util.Scanner;

public class ReadCharacter
{
 public static void main(String[] args)
 {
  var sc = new Scanner(System.in);
  System.out.print("Enter a Character :");
  char gen = sc.next().charAt(0);
  System.out.println("Your Gender is :"+gen);
                sc.close();
 }

}
```

//WAP to read employee data using Scanner class

```
package com.ravi.scanner_demo;

public class ReadEmployeeData {

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

  System.out.print("Enter Employee Id :");
  int id = sc.nextInt(); //123

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

  System.out.println("Printing Employee Data");

  System.out.println("Employee Id is :"+id);
  System.out.println("Employee Name is :"+name);
  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 format is required.

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

```
  byte b = 1;
  byte c = 2;
  byte d = b + c; //error
  System.out.println(d);

 }

}
```

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

```
class Test
{
 public static void main(String[] args)
 {
  byte b = 1;
  byte c = 2;
  byte d = (byte)(b + c);  //Valid
  System.out.println(d);

 }

}
```

Unary Minus Operator :

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


}
```

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

```
class Test
{
```

```
 public static void main(String [] args)
 {
  byte b = 1;
  b += 2;
  System.out.println(b);

 }

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

```
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

 }

}
```

Program on Boolean AND operator :

```
class Test
{
 public static void main(String [] args)
 {
  int z = 5;
  if(++z > 6 & ++z> 6)
  {
   System.out.println("Inside If");
   z++;
  }
  System.out.println(z);

 }

}
```

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

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

}
```

Note : If both the inputs are alternate of each other then we will get true otherwise we will get false.[Same input output will be false]

```
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 type.

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

 }

}
```

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

 }

}
```

Member access operator (.) :

It is called Member access operator, by using this we can access the member of the class.
In the following program we have static method in the Welcome class, in order to call the static method

we can use Welcome class and to access the static method of Welcome class we should use .(Dot) operator.

```
package com.ravi.operator;

class Welcome
{
 static int x = 100;
 public static void greet()
 {
  System.out.println("Hello batch 39");
 }
}

public class MemberAccessOperator
{
 public static void main(String[] args)
 {
      System.out.println(Welcome.x);
      Welcome.greet();
 }

}
```

---

new Keyword :

It is also an operator.
It is used to create the object and initialize the non static member with default value.

```
package com.ravi.operator;

class Welcome
{
 int x = 100;  //non static variable
 public  void greet() //non static method
 {
  System.out.println("Hello batch 39");
 }
}

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

}
```

17-10-2024

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 and 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.

```
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.*;
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;
  }
 }
}
-----------------------------------------------------------------

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;
  }
 }
}
```

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

    switch(l)
  {
          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;
  }
 }

}
```

```
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

do-while loop :

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

}
```

Note : x variable is declared inside the do block so we can’t use outside of the block.

```
class Test
{
 public static void main(String [] args)
 {
  int x = 1;
        do
        {
   System.out.println(x);
   x++;
        }
        while (x<=10);
 }

}
```

Progran on while loop :

```
class Test
{
 public static void main(String [] args)
 {
  int x = 10;
  while(x>=-1)
  {
   System.out.println(x);
   x--;
  }
 }

}
```

Program on for loop :

```
public class ForLoop
{
 public static void main(String[] args)
 {
  for(int i=1; i<=10; i++)
  {
                System.out.println(i);
  }
 }
}
```
---
for-each loop in java :

It is also known as enhaned for loop, introduced from JDK 1.5

It is used to retrieve the value from the Collection like array.

It will fetch the values one be one from the Collection data so, It is known as for each loop.

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

```
public class ForEachDemo1
{
 public static void main(String [] args)
 {
  int []arr = {50,40,30,20,10};

  Arrays.sort(arr);

  for(int x : arr)
  {
   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 ascending or alphabetical order.

Example :

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

```
public class ForEachDemo2
{
 public static void main(String[] args)
 {
  String cities[] = {"Hyd","Pune","Ajmer","Mumbai"};

  java.util.Arrays.sort(cities);

  for(String city : cities)
  {
   System.out.println(city);
  }
 }
}
```

In java, Array can also take hetrogeneous by using Object class.

```
public class ForEachDemo3
{
 public static void main(String[] args)
 {
  Object []arr = {12, 90.90, 'A', "NIT", new String("Hyd")};


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

}

Note : By using Object array we can also take hetrogeneous types
of datain java array but these data can't be sorted, If we try to sort then we will get a runtime Exception i.e java.lang.ClassCastException.

18-10-2024

What is BLC and ELC class in java ?

BLC :

BLC statnds for Business Logic class, In this class we are responsible to write the logic. This class will not contain main method.
The main purpose of this BLC class to reuse this class in various packages.

Example :

```
//BLC
public class Calculate
{
  //Here We are responsible to write the logic
}
```

ELC :

It stands for Executable Logic class, It will not contain any logic but the execution of the program will start from this ELC class because it contains main method.

Example :

```
//ELC
public class Main
{
  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 that must be our .java file 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]

Program that describes how to reuse a java BLC class in another package :

Here we have 2 packages :
---
1) com.nit.m1
2) com.nit.m2

Calculate.java [BLC]

```
package com.nit.m1;

//BLC
public class Calculate
{
   public static void getSquare(int x)
   {
    System.out.println("Square of "+x+" is :"+(x*x));
   }
}
```

Arithmetic.java [BLC]

```
package com.nit.m1;

public class Arithmetic
{
 public static void doSum(int x, int y)
 {
  System.out.println("Sum is :"+(x+y));
 }
}
```

Now we will write an ELC class in another package to reuse these
BLC classes which are declared in com.nit.m1 package.

ELC.java [ELC]

```
package com.nit.m2;

import com.nit.m1.Arithmetic;
import com.nit.m1.Calculate;

public class ELC
{
 public static void main(String[] args)
 {
  Calculate.getSquare(12);
  System.out.println(".....");
  Arithmetic.doSum(100, 200);

 }

}
```

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 :

```
public class Test
{

}

class A
{
}

class B
{
}

class C
{
}
```

Note : Here total 4 .class file will be generated.

Working with static method and method return type :
---
static method :

If a method is declared with static keyword (like main method) then it is called static method.

In order to call a static method, Object is not required, We can call static method directly with the help of class name.

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

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

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

Note : Any static method defined in the ELC class, we can direcytly call from main method.

---
2 files :

GetSquare.java

```
package com.ravi.pack2;

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

Test2.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 from BLC module to ELC module, ELC module is sending the value to BLC module but BLC module is not returining any kind of value.

---
2 files :

FindSquare.java

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

//BLC
public class FindSquare
{
 public static int getSquare(int x)
 {
  return (x*x);
 }
}
```

Test3.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 value = FindSquare.getSquare(side);
    System.out.println("Square of "+side+" is :"+value);
    sc.close();
 }
}
```
---
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
*/

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 squareAndCube = Calculate.getSquareAndCube(num);
 System.out.println("Result is :"+squareAndCube);
 sc.close();

   }

}
```
---
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.printf("Area of Rectangle is :%.2f",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);
    }
}
```

Test6.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();

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

}
```

---

2 files :

Circle.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;
   }

  }
}
```

Test7.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);
  System.out.println(areaOfCircle);

  sc.close();



 }
}
```

19-10-2024
---
2 files :

Student.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)
 {
    System.out.println(Student.getStudentDetails(101, "Smith", 12000.99));

 }

}
```

Note : we can call any method whose return type is not void by using System.out.println() method but we can't call a method whose return type is void.

```
class Alpha
{
  public static void m1()
  {
  }

  public static int m2()
  {
  }
}

System.out.println(Alpha.m1());  //error
System.out.println(Alpha.m2());  //Valid
```

---

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
public class Test9
{
 public static void main(String[] args)
 {
  for(int i=1; i<=10; i++)
  {
   Table.printTable(i);
  }
 }

}
```

Types of Variables in java :

In java, Based on the data types variables are divided into two types :

1) Primitive Variables
2) Reference Variables

1) Primitive Variables :

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

Example :

```
 byte x = 12;
 int y = 90;
 boolean isEmpty = false;
```

On primitive variables we can’t assign null literal as well as with primitive variables we can’t call a method.

```
int x = null;  //Invalid

int y = 12;
    y.m1(); //Invalid
```

Reference Variables :

In java, If a variable is declared with class name then it is called reference variable.

Example :

```
   Integer x = 19;
   String str = "India";
   Student st;
   Customer c = null;
```

On reference variable we can assign null as well as we can call a method.

```
 String str = null;  //Valid
 String y = "India";
 y.toUpperCase(); //valid
```

Based on the Declaration position, these two variables are further classified into 4 categories :

1) Class Variable OR Static Field
2) Instance Variable OR Non Static field

3) Local Variable
4) Parameter Variable

Program on Primitive Variables :

```
package com.ravi.variable_type;

class Test
{
 static int a; //static Field OR Class Variable
 int b;        //Non Static Field OR Instance Variable

 public void accept(int c)  //C is parameter Variable
 {
  int d = 400;           //d is a local Variable

  System.out.println("Class Variable :"+a);
  System.out.println("Instance Variable :"+b);
  System.out.println("Parameter Variable :"+c);
  System.out.println("Local Variable :"+d);

 }
}
public class PrimitiveVariablesDemo
{
 public static void main(String[] args)
 {
  Test t1 = new Test();
  t1.accept(300);
 }
```

```
}

Output : 0
         0
  300
  400
-----------------------------------------------------------------
Program on Reference Variable :
-------------------------------
package com.ravi.variable_type;

import java.util.Scanner;

class Student
{
 Student s1 = null; //Instance + Reference Variable
 static Scanner sc = new Scanner(System.in); //Static + Reference Var

 public void accept(Student st) //st parameter variable
 {
  Student s2 = new Student();  //s2 local Variable
  System.out.println("Executed");
 }
}

public class ReferenceVariable {

 public static void main(String[] args)
 {
  Student student = new Student();
  student.accept(student);

 }

}
=================================================================
21-10-2024
----------
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

==========

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 :

Student.java

```
package com.ravi.oop;

//BLC
public class Student
{
   String name;  //Instance Variable
   double height; //Instance Variable
   int rollNumber; //Instance Variable

   //Object Behavior
   public void talk()
   {
    System.out.println("Hello Everyone, My name is :"+name);
```

```
    System.out.println("My Roll number is :"+rollNumber);
    System.out.println("And my height is :"+height);

   }

   public void writeExam()
   {
    System.out.println("Every Saturday "+name+ " is writing exam for Enjoying sunday");
   }



}
```

StudentDemo.java

```
package com.ravi.oop;

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

  //Initialize the Object Properties using reference variable
  raj.name = "Raj Gourav";
  raj.rollNumber = 111;
  raj.height = 5.9;

  //Calling the behavior
  raj.talk();
  raj.writeExam();

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

  Student priya = new Student();
  priya.name = "Priya";
  priya.height = 5.6;
  priya.rollNumber =222;

  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)

22-10-2024

How to initialize the Object properties using Object reference
through Scanner class

2 files :

Product.java

```
package com.ravi.oop;

public class Product
{
 String laptopBrand;
 double laptopPrice;
 boolean isTouchScreen;

 public void getLaptopInformation()
 {
  System.out.println("Laptop Brand is :"+laptopBrand);
  System.out.println("Laptop Price is :"+laptopPrice);
  System.out.println("Is laptop touch screen ?:"+isTouchScreen);
 }
}
```

ProductDemo.java

```
package com.ravi.oop;

import java.util.Scanner;

public class ProductDemo {

 public static void main(String[] args)
 {
  Product laptop = new Product();
  Scanner sc = new Scanner(System.in);
  System.out.print("Enter Laptop Brand :");
  laptop.laptopBrand = sc.nextLine();

  System.out.print("Enter Laptop Price :");
  laptop.laptopPrice = sc.nextDouble();

  System.out.print("Is it touch screen Laptop :");
  laptop.isTouchScreen = sc.nextBoolean();

  laptop.getLaptopInformation();
  sc.close();
```

```
 }

}
```

What is instance OR Non static variable :

It is a class level variable so It has default value.

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

Example :

```
public class Student
{
   int rollNumber;  //Instance Variable [Object Properties]

   public void setStudentData()
   {
   }
}
```

An instance variable life starts at the time of creating the object, Without object we can't think about instance variable.

```
public class Test
{
 int x = 100;

 public static void main(String[] args)
 {
  System.out.println(x); //error
 }
}
```

As far as its accessibility is concerned, It is accessible within the same class as well as depends upon the access modifier applied on the instance variable.

Parameter Variable :

It is a method level variable hence does not have default value.

If a variable is declared inside a method parameter (not inside method body) then it is called Parameter Variable.

End user is responsible to initialize the parameter variable.

As far as it's scope is concerned, It is accessible within the same method body only.

```
public class Employee
{
   int employeeId;

   public void set(int id)  //id is parameter variable
   {
   }
```

```
}
============================================================
2 files :
---------
Employee.java
---------------
package com.ravi.oop;

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

   public void setEmployeeData(int id, String name, double sal, String addr)
   {
    employeeId = id;
    employeeName = name;
    employeeSalary = sal;
    employeeAddress = addr;
   }

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


}

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

public class EmployeeDemo
{
 public static void main(String[] args)
 {
  Employee scott = new Employee();
  scott.setEmployeeData(111, "Scott", 55000, "S R Nagar");
  scott.getEmployeeData();

  System.out.println("================");
  Employee smith = new Employee();
  smith.setEmployeeData(222, "Smith", 56000, "Ameerpet");
  smith.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.

```
public class Example
{
   public Example() //Constructor
   {
   }
}
```

Default Constructor added by java compiler :

In java, Whenever we write a class and if a programmer does not
write any kind of constructor in the class then automatically one default constructor will be added by the compiler in the class.

```
public class Example
{

}
```

javac Example.java

```
public class Example
{
    public Example() //Default Constructor added by compiler
    {
    }
}
```

Every java class must have at-least one constructor [We can't think about java class without constructor] either implicitly added by java compiler OR explicitly written by programmer.

The access modifer of default constructor (added by compiler) depends upon class access modifier, if class is public then the access modifier of default constructor is also public.

Example :

```
public class Demo
{

}
```

javac Demo.java
javap Demo.class [You Can see the constructor added by compiler]

Why compiler is adding default constructor to our class :

We have 2 reasons that why compiler is adding default constructor :

1) Without default constructor, Object creation is not possible in java by using new keyword.

2) As we know only class level variables are having default values so, default constructor will initialize all the instance variables with default values with the help of new keyword.

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)

==========

```
package com.ravi.oop;

public class Employee
{
        int id;
 String name;
 double salary;

 public void show()
 {
  System.out.println(id);
  System.out.println(name);
  System.out.println(salary);
 }
 public static void main(String[] args)
 {
    Employee emp = new Employee();
    emp.show();
 }
}
```

Output is : 0 null 0.0

23-10-2024

Initializing the instance variable using parameter variable as per requirement :

2 files :

Employee.java

```
package com.nit.oop;

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

```
    public void setEmployeeData(int id, String name, double salary)
    {
     employeeId = id;
     employeeName = name;
     employeeSalary = salary;
    }

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

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

}
```

EmployeeDemo.java

```
package com.nit.oop;

public class EmployeeDemo
{
 public static void main(String[] args)
 {
  Employee scott = new Employee();
  scott.setEmployeeData(101, "Scott", 74000);
  scott.calculateEmployeeGrade();
  scott.getEmployeeData();

 }

}
```

Note : In the above program all the object properties are not

initialized with parameter variable, actually employeeGrade is initialized by employeeSalary.

Note : Upto Here we have alreday learned the followinbg ways to initialize the object properties :

1) By using Object Reference (Reference Variable)
2) By using Method without parameter
3) By using Method Parameter (Initializing the Object Properties through parameter variable)

What is Variable Shadow in java ?

[Method Level variables are having more priority than class level variables inside method, block OR Constructor]

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 (Method, block Or Constructor), when we refer to the variable directly by name, We are actually referring to the inner variable, not the outer variable.

2 files :

Student.java

```
package com.nit.oop;

public class Student
{
    int id = 100;
 String name = "Scott";

 public void accept()
 {
  int id = 200;
  String name = "Smith";
  System.out.println(id +" : "+name);
 }

 public void input(int id, String name)
 {
  System.out.println(id +" : "+name);
 }

}
```

StudentDemo.java

```
package com.nit.oop;

public class StudentDemo {

 public static void main(String[] args)
 {
  Student s = new Student();
```

```
  s.accept();
  s.input(300, "John");
 }

}
```
---
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/local 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 represent instance variable.

We cannot use this (non static member) keyword from static area (Static context).

2 files :

Manager.java

```
package com.nit.oop;

public class Manager
{
   int managerId;
   String managerName;

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

   }

   public void getManagerData()
   {
    System.out.println("Manager id is :"+managerId);
    System.out.println("Manager name is :"+managerName);
   }

}
```

ThisKeywordExample.java

```
package com.nit.oop;

public class ThisKeywordExample
{
 public static void main(String[] args)
 {
  Manager m1 = new Manager();
  m1.setManagerData(111, "Mr Smith");
```

```
    m1.getManagerData();
 }
}
```

Note : this keyword is used to represent instance variable.

Local Search algorithm [Paint Diagram 24-OCT]:

Program :

```
package com.ravi.inner_class_concept;

class Test
{
 static int a = 100;  //Class Variable OR Static Field
 int b = 200;         //Instance Variable OR Non static Field

 public void accept(int c)
 {
  int d = 400;
  System.out.println(Test.a);
  System.out.println(this.b);
  System.out.println(c);
  System.out.println(d);
 }
}

public class Main {

 public static void main(String[] args)
 {
  Test t1 = new Test();
  t1.accept(300);
 }

}
```

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

If we want to print our object properties (Instance Variables) 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 :

Product.java

```
package com.ravi.to_string_demo;

public class Product
{
 int productId;
 String productName;

 public void setProductData(int productId, String productName)
 {
  this.productId = productId;
  this.productName = productName;
 }

 @Override
 public String toString()
 {
  return "Product [productId=" + productId + ", productName=" + productName + "]";
 }


}
```

ProductDemo.java

```
package com.ravi.to_string_demo;

public class ProductDemo
{
 public static void main(String[] args)
 {
  Product p1 = new Product();
  p1.setProductData(111, "Laptop");
  System.out.println(p1);  //toSting() method of Product class

  Product p2 = new Product();
  p2.setProductData(222, "Smart Phone");
  System.out.println(p2);
 }

}
```

25-10-2024

Role of instance variable while creating the Object :

Whenever we create an object in java then a separate copy of all the instance variables will be created with each and every object as shown in the program.[25-OCT]

```
package com.ravi.variable_copy_demo;
```

```
public class Test
{
 int x = 100;   //Non static field

 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 with Object creation :

Whenever we create an object then a single copy of static variable will be created for all the objects and the same single copy of static variable will be sharable by all the objects as shown in the program.[25-OCT]

```
package com.ravi.variable_copy_demo;

public class Demo
{
 static int x = 100; //static field
 public static void main(String[] args)
 {
  Demo d1 = new Demo();
  Demo d2 = new Demo();

  --d1.x;   --d2.x;

  System.out.println(d1.x);  //98
  System.out.println(d2.x); //98
 }
}
```

So, final conclusion is :

Instance Variable = Multiple Copies

Static Variable = Single Copy for all the objects

Basically static variables are used to save the memory.

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

Non static Variable :

Multiple copies will be created with each and every object.

If the value of the variable is different with respect to object then we should use instance variable OR non static field.

Static Field :

If the value of the variable is common with respect to object
then we should use static field OR class variable.

Example1 :

```
public class Student
{
  int rollNumber;  //NSV
  String studentName; //NSV
  String studentAddress//NSV
  static String collegeName = "VIT"; //SV
  static String courseName = "Java"; //SV
}
```

Example 2 :

```
class Customer
{
   long accountNumber; //NSV
   String customerName; //NSV
   long mobileNumber; //NSV
   String customerAddress; //NSV
   static String IFSCCode = "SBIHYD08590"; //SV
   static String branchLocation = "Ameerpet"; //SV
}
```

Program :

---

2 files :

Student.java

```
package com.ravi.variable_copy_demo;

public class Student {
 int rollNumber;
 String studentName;
 String studentAddress;
 static String collegeName = "JNTU";
```

```
 static String courseName = "B.Tech";

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

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

}
```

StudentDemo.java

```
package com.ravi.variable_copy_demo;

public class StudentDemo
{
 public static void main(String[] args)
 {
  Student raj = new Student();
  raj.setStudentData(101, "Raj", "Ameerpet");

  Student priya = new Student();
  priya.setStudentData(102, "Priya", "S R Nagar");

  Student scott = new Student();
  scott.setStudentData(103, "Scott", "Koti");

  System.out.println(raj);
  System.out.println(priya);
  System.out.println(scott);

 }

}
```

Assignment :

Develop Bank and Customer application with valid SV and NSV

**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

```
package com.ravi.data_hiding;

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

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

   public void withdraw(double amount)
   {
    this.balance = this.balance - amount;
    System.out.println("After Withdraw :"+this.balance);
   }


}
```
BankingApplication.java

```
package com.ravi.data_hiding;

public class BankingApplication
{
 public static void main(String[] args)
 {
  Customer c1 = new Customer();
  c1.deposit(10000);
  c1.withdraw(5000);

 }

}
```

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. [26-OCT]

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

Defination of Constructor :

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

The main purpose of constructor to initialize the object properties (Instance Variables) with user-defined value.

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

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

A constructor never contain any return type including void also.

Example :

```
package com.ravi.command;

class Student
{
  public void Student() //Method
  {
   System.out.println("I am Method");
  }

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

A constructor may contain return keyword but not return keyword with value.

```
package com.ravi.command;

class Student
{
  public Student()
  {
   System.out.println("I am Constructor");
   return;
  }

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

```
  }
}
```

A constructor is automatically called and executed at the time of creating the object.

==========

Types of constructor in java :

We have 3 types of Constructors in java :

1) Default No argument Constructor [Added by compiler]
2) No Argument OR Parameter-less OR Non parameterized OR Zero argument Constructor.
3) Parameterized Constructor.

Default No Argument Constructor :

Whenever we write a class and if we don't write any type of constructor then automatically one default constructor is added by the compiler.

The access modifier of default constaructor would be same as class access modifier.

It does not accept any parameter.

Example.java

```
public class Example
{

}
```

```
javac Example.java
```

Example.class

```
public class Example
{
   public Example() //default constructor
   {
   }
}
```

---

2) No Argument OR Parameter-less OR Non parameterized OR Zero argument Constructor.

If a user defines a constructor inside a class without argument then it is called no argument constructor.

No argument constructor and default constructor, both look like same the only difference is, default constructor means added by compiler and no argument constructor means written by user.

```
public class Student
{
  private int rollNumber;
  private String studentName;
```

```
  public Student() //No Argument Constructor
  {
     rollNumber = 111;
     studentName = "Raj";
  }

}
```

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

```
package com.ravi.constructor;

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.constructor;

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

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

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

 }

}
```

Note : Actually No argument constructor is used to initialize the object properties with default values.

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 :

Dog.java

```
package com.ravi.constructor;

public class Dog
{
 private String dogName;
 private double dogHeight;
 private int dogAge;

 public Dog(String dogName, double dogHeight, int dogAge) {
  super();
  this.dogName = dogName;
  this.dogHeight = dogHeight;
  this.dogAge = dogAge;
 }

 @Override
 public String toString() {
  return "Dog [dogName=" + dogName + ", dogHeight=" + dogHeight + ", dogAge=" + dogAge + "]";
 }

}
```

ParameterizedConstructor.java

```
package com.ravi.constructor;

public class ParameterizedConstructor
{
```

```
 public static void main(String[] args)
 {
  Dog tommy = new Dog("Tommy", 3.5, 4);
  System.out.println(tommy);

  Dog tiger = new Dog("Tiger", 4.5, 7);
  System.out.println(tiger);

 }

}
```

What is setter and getter :

---

setter : Used to modify the existing object data.
getter : Used to read the private data from BLC class.

2 files :

Employee.java

```
package com.ravi.setter_getter;

public class Employee
{
 private double employeeSalary;

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

 public double getEmployeeSalary()
 {
  return this.employeeSalary;
 }

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

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

Main.java

```
package com.ravi.setter_getter;

public class Main {
```

```
 public static void main(String[] args)
 {
  Employee scott = new Employee(40000);
  System.out.println(scott);

  System.out.println("After 1 year, Salary Updated");

  scott.setEmployeeSalary(scott.getEmployeeSalary()+10000);
  System.out.println(scott);

 }

}
```

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]

28-10-2024

*** 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.

Program on Setter and Getter using Encapsulation :

---
2 files :

Student.java

```
package com.ravi.setter_getter;

public class Student
{
 private int studentId;
 private String studentName;
 private int studentMarks;
 private String studentAddress;

 public Student(int studentId, String studentName, int studentMarks, String studentAddress)
 {
  super();
  this.studentId = studentId;
  this.studentName = studentName;
  this.studentMarks = studentMarks;
  this.studentAddress = studentAddress;
 }

 @Override
 public String toString() {
  return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentMarks=" +
studentMarks
    + ", studentAddress=" + studentAddress + "]";
 }

 public int getStudentId() {
  return studentId;
 }

 public void setStudentId(int studentId) {
  this.studentId = studentId;
 }

 public String getStudentName() {
  return studentName;
 }

 public void setStudentName(String studentName) {
  this.studentName = studentName;
 }

 public int getStudentMarks() {
  return studentMarks;
 }

 public void setStudentMarks(int studentMarks) {
  this.studentMarks = studentMarks;
 }

 public String getStudentAddress() {
```

```
  return studentAddress;
 }

 public void setStudentAddress(String studentAddress)
 {
  this.studentAddress = studentAddress;
 }
}
```

StudentDemo.java

```
package com.ravi.setter_getter;

public class StudentDemo {

 public static void main(String[] args)
 {
  Student raj = new Student(111, "Raj", 90, "Ameerpet");
  System.out.println(raj);
  //Address Updated
  raj.setStudentAddress("S R Nagar");
  System.out.println(raj);

  int marks = raj.getStudentMarks();

  if(marks >= 90)
  {
   System.out.println(raj.getStudentName()+" is Excellent in Subject");
  }
  else if(marks >=75)
  {
   System.out.println(raj.getStudentName()+" is Very good in Subject");
  }
  else
  {
   System.out.println(raj.getStudentName()+" is good in Subject");
  }



 }

}
```

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.

Case 1 :

```
package com.ravi.method_return_type;

public class Test
{
 public Test accept()
 {
  return this;
     OR
   return null;
     OR
   return new Test();
 }
}
```

Case 2 :

```
package com.ravi.method_return_type;

public class Demo
{
   private int id;

   public Demo(int id)
   {
    this.id = id;
   }

   public Demo get()
   {
      return new Demo(5);
   }



}
```

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 :

Product.java

```
package com.ravi.method_return_type;

public class Product
{
 private int productId;
 private String productName;
```

```
 private double productPrice;

 public Product(int productId, String productName, double productPrice) {
  super();
  this.productId = productId;
  this.productName = productName;
  this.productPrice = productPrice;
 }

 @Override
 public String toString() {
  return "Product [productId=" + productId + ", productName=" + productName + ", productPrice=" +
productPrice
    + "]";
 }

 public static Product getProductObject()
 {
  return  new Product(111, "Laptop", 92000.90);
 }

}
```

ProductDemo.java

```
package com.ravi.method_return_type;

public class ProductDemo {

 public static void main(String[] args)
 {
  Product product = Product.getProductObject();
  System.out.println(product);
 }

}
```

In the avove program getProductObject() is providing only one product object so it is not recommended because the main purpose of any method to provide re-usability as shown in the program below.

---

2 files :

Book.java

```
package com.ravi.method_return_type;

import java.util.Scanner;

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

 public Book(String bookTitle, String authorName)
 {
  super();
```

```
  this.bookTitle = bookTitle;
  this.authorName = authorName;
 }

 @Override
 public String toString() {
  return "Book [bookTitle=" + bookTitle + ", authorName=" + authorName + "]";
 }

 public static Book getBookObject()
 {
  Scanner sc = new Scanner(System.in);
  System.out.print("Enter Book Title :");
  String title = sc.nextLine();
  System.out.print("Enter Author Name :");
  String author = sc.nextLine();

  return new Book(title, author);
 }

}
```

BookDemo.java

```
package com.ravi.method_return_type;

import java.util.Scanner;

public class BookDemo {

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

  for(int i=1; i<=noOfObj; i++)
  {
   Book object = Book.getBookObject();
   System.out.println(object);
  }

  sc.close();
 }

}
```

29-10-2024

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;
 }

 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 + "]";
 }
}
```

ShallowCopyDemo.java

```
package com.ravi.shallow_copy;

public class ShallowCopyDemo {

 public static void main(String[] args)
 {
  Laptop laptop1 = new Laptop("Acer", 60000);
```

```
  System.out.println(laptop1);

  Laptop laptop2 = laptop1;
  laptop2.setLaptopBrand("HP");
  laptop2.setLaptopPrice(80000);

  System.out.println(laptop1);
  System.out.println(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 :

Product.java

```
package com.ravi.deep_copy;

public class Product
{
 private int productId;
 private String productName;

 public Product()
 {
  productId = 0;
  productName = null;
 }

 public Product(int productId, String productName)
 {
  super();
  this.productId = productId;
  this.productName = productName;
 }

 public int getProductId() {
  return productId;
 }

 public void setProductId(int productId) {
  this.productId = productId;
 }

 public String getProductName() {
  return productName;
 }
```

```
 public void setProductName(String productName) {
  this.productName = productName;
 }

 @Override
 public String toString() {
  return "Product [productId=" + productId + ", productName=" + productName + "]";
 }

}
```

DeepCopyDemo.java

```
package com.ravi.deep_copy;

public class DeepCopyDemo
{
 public static void main(String[] args)
 {
  Product p1 = new Product(111, "Laptop");

  Product p2 = new Product();
  p2.setProductId(p1.getProductId());
  p2.setProductName(p1.getProductName());

  System.out.println("Before Modification...");
  System.out.println(p1);
  System.out.println(p2);

  System.out.println("After Modification...");
  p1.setProductId(222);
  p1.setProductName("Camera");
  System.out.println(p1);
  System.out.println(p2);


 }

}
```

Note : Here only object content will be modified.

**Pass by Value :

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

Pass by value means we are sending the copy of orginal data to the method.

```
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;
 }

}
```

```
package com.ravi.pass_by_value;

public class PassByValueDemo2
{
 public static void main(String[] args)
 {
  int x = 100;
  x = accept(x);
  System.out.println(x);
 }

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

```
package com.ravi.pass_by_value;

class Customer
{
 private double customerBill = 12000;

 public double getCustomerBill()
 {
  return customerBill;
 }

 public void setCustomerBill(double customerBill)
 {
  this.customerBill = customerBill;
 }

}

public class PassByValueDemo3
{
 public static void main(String[] args)
 {
  Customer c1 = new Customer();  //customerBill = 12000
  accept(c1);
  System.out.println(c1.getCustomerBill()); ////customerBill = 18000
 }
```

```
 public static void accept(Customer cust)
 {
  cust.setCustomerBill(18000);
 }
}
```

Output : 18000

```
package com.ravi.pass_by_value;

class Customer
{
 private double customerBill = 12000;

 public double getCustomerBill()
 {
  return customerBill;
 }

 public void setCustomerBill(double customerBill)
 {
  this.customerBill = customerBill;
 }

}

public class PassByValueDemo4
{
 public static void main(String[] args)
 {
  Customer c1 = new Customer();  //customerBill = 12000
  accept(c1);
  System.out.println(c1.getCustomerBill()); //12000
 }

 public static void accept(Customer cust)
 {
  cust = new Customer();
  cust.setCustomerBill(18000);
 }
}
```

Output : 12000

==========

What is Garbage Collector in java ----------

It is an automatic memory management technique in java.

In C++ language, A programmer is responsible to allocate as well as de-allocate the memory otherwise we will get OutOfMemoryError.

In java language, Programmer is only responsible to allocate the memory, Memory de-allocation is automatically done by garbage collector.

Garbage Collector is a daemon thread which is responsible to delete the objects from the HEAP Memory.

Actually It scans the heap memory and identifying which objects are eligible for Garbage Collector.[THE OBJECTS WHICH DOES NOT CONTAIN ANY REFERENCES ONLY THOSE OBJECTS ARE ELIGIBLE FOR GC]

It internally uses an algorithm called Mark and Sweep algorithm to delete the un-used objects.

As a developer we can also explicitly call garbage collector by writing the following code

System.gc();

==========
How many ways we can make an object eligible for Garbage Collector :

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.
==========
30-10-2024

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.

HEAP and STACK Diagram for CustromerDemo.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;
 }
}

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

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

        m1(c);

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

  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
============================================================
HEAP and STACK Diagram for Sample.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 are eligible for 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
-------------------------------------------------------------
Heap and Stack Digram for Test.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;
 }

 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);
 }

}
```

01-11-2024

HEAP and STACK Diagram for EmployeeDemo.java

```
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, 1000x

    //GC [2 objects, 2000x and 4000x both are eligible]

   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

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

void m1(Test t)  //t -> 2000x
{
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 , y , m1()
Test t2=new Test(); //x , y , m1()

t1.m1(t2);

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

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);
}
}
```

==========

Passing an Object reference to the Constructor :(Copy Constructor)

We can pass an object reference to the constructor so we can copy the content of one object to another object.

The following program explains how to copy the content of Employee object to initialize Manager class properties :

3 files :

Employee.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;
 }

 public int getEmployeeId() {
  return employeeId;
 }

 public String getEmployeeName() {
  return employeeName;
 }

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

}
```

Manager.java

```
---------
package com.ravi.copy_constructor;

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 + "]";
 }

}
```

CopyConstructor.java

```
--------------------
package com.ravi.copy_constructor;
```

```
public class CopyConstructor
{
 public static void main(String[] args)
 {
  Employee e1 = new Employee(111, "Scott");
        Manager m1 = new Manager(e1);
        System.out.println(m1);
 }

}
```

Note : Here by using Employee class properties, we are initializing the Manager class properties.

The following program explains how to copy the content of one object of same class to another object of same class only.

Product.java

```
package com.ravi.copy_constructor;

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 p)  //p = prod
 {
  this.productId = p.productId;
  this.productName = p.productName;
 }

 @Override
 public String toString() {
  return "Product [productId=" + productId + ", productName=" + productName + "]";
 }
}
```

CopyConstructorDemo.java

```
package com.ravi.copy_constructor;

public class CopyConstructorDemo
{
 public static void main(String[] args)
 {
  Product prod = new Product(101, "HP Laptop");
  Product p1 = new Product(prod);
```

```
  System.out.println(prod);
  System.out.println(p1);

 }

}
```
==========
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.

Calculate.java

```
package com.ravi.constructor_overloading;

public class Calculate
{
   public Calculate(int x, int y)
   {
    System.out.println("Sum of "+x+" and "+y+ " is :"+(x+y));
   }

   public Calculate(int x)
   {
    this(100,200);
    System.out.println("Square of "+x+" is :"+(x*x));

   }
}

package com.ravi.constructor_overloading;

public class ConstructorOverloading
{
 public static void main(String[] args)
 {
  Calculate c1 = new Calculate(5);
 }

}
```

02-11-2024

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 blocks 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_block;

class Sample
{
 {
  System.out.println("Instance OR Non static block");
 }
}

public class InstanceBlockDemo1
{
 public static void main(String[] args)
 {
  new Sample(); //Nameless OR Anonymous Object
  new Sample();
 }

}
```

Note : Instance block will be executed with object creation

```
package com.ravi.instance_block;

class Demo
{
 public Demo()
 {
  System.out.println("Demo class Constructor");
 }

 {
  System.out.println("NSB");
```

```
 }

}

public class InstanceBlockDemo2
{

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

 }

}
```

Note : NSB will be executed before the constructor body.

```
package com.ravi.instance_block;

class Foo
{
 Foo()
 {
  System.out.println("No Argument Constructor");
 }

 Foo(int x)
 {
  System.out.println("Parameterized Constructor");
 }

 {
  System.out.println("NSB");
 }

}


public class InstanceBlockDemo3 {

 public static void main(String[] args)
 {
     new Foo();
     new Foo(10);

 }

}
```

Note : NSB will be placed inside all the constructors which contains super() in the first line.

```
package com.ravi.instance_block;

class Student
{
```

```
 public Student()
 {
  this(101,"Scott");
  System.out.println("No Argument Constructor");
 }

 public Student(int id, String name)
 {
  System.out.println("Parameterized Constructor");
 }

 {
  System.out.println("Object creation is in process");
 }
}


public class InstanceBlockDemo4 {

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

 }

}
```

NOte : NSB will not be added to the constructor which contains this() as a first line of constructor.

```
package com.ravi.instance_block;

class Test
{
 int x;

 public Test()
 {
  x = 590;
  System.out.println("x value is :"+x);
 }

 {
  x = 190;
  System.out.println("x value is :"+x);
 }
}

public class InstanceBlockDemo5
{
 public static void main(String[] args)
 {
  Test t1 = new Test();
 }
```

```
}
```

Note : NSM is also used to initialize the instance variable.

```
package com.ravi.instance_block;

class Customer
{
 private double bill;

 public Customer()
 {
  bill = 10000;
  System.out.println(bill);
 }
 {
  bill = 1000;
  System.out.println(bill);
 }

 {
  bill = 2000;
  System.out.println(bill);
 }

 {
  bill = 3000;
  System.out.println(bill);
 }

 {
  bill = 4000;
  System.out.println(bill);
 }

}

public class InstanceBlockDemo6
{
   public static void main(String[] args)
   {
    new Customer();
   }
}
```

Note : It is executed top to bottom.

```
package com.ravi.instance_block;

class Manager
{
 int x = 10;

 {
```

```
  System.out.println("Instance Initializer");
  //return;
 }



}

public class InstanceBlockDemo7
{

 public static void main(String[] args) {
  // TODO Auto-generated method stub

 }

}
```

Note : return statement not allowed.

```
package com.ravi.instance_block;

class Hello
{
 public Hello()
 {
  System.out.println("Constructor");
  {
   System.out.println("NSB2");
  }
 }
 {
  System.out.println("NSB1");
 }
}


public class InstanceBlockDemo8
{
 public static void main(String[] args)
 {
  new Hello();

 }

}
```

Note : If we wtite NSB after the body of the constructor 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 with default value, 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 [new keyword] => At the time of declaration => in the body of non static block => in the body of constructor => Inside method body [Not Recommended]

```
package com.ravi.nsv_life_cycle;

class Test
{
 int x = 100;   //STEP 1

 {
  x = 200;  //STEP 2
 }

 Test()
 {

  x = 300; //STEP 3
 }

}


public class LifeCycle
{
 public static void main(String[] args)
 {
  Test t1 = new Test();
  System.out.println(t1.x); //300

 }

}
```

==========

12-11-2024

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 as shown in the program.

```
class Test
{
 final int A;  //Blank final field

 public static void main(String[] args)
 {
  Test t1 = new Test();
  System.out.println(t1.A);
 }
}
```

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 completed]

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.

```
package com.ravi.blank_final_field;

class Sample
{
 final int x;

 {
  x = 123;
 }

 public Sample()
 {
  // x = 234;
 }
}


public class BlankFinalFieldDemo {

 public static void main(String[] args)
 {
  Sample s1 = new Sample();
  System.out.println(s1.x);

 }
```

```
}
```

A blank final field must be initialized by the non static block constructor body

==========

```
package com.ravi.blank_final_field;

class Test
{
 final int x; //blank final field

 {
  m1();
  x = 100;
 }

 public void m1()
 {
  System.out.println("Default value :"+x);
 }

}

public class BlankFinalFieldDemo1 {

 public static void main(String[] args)
 {
  Test t1 = new Test();
  System.out.println("User Value :"+t1.x);

 }

}
```

Note : A blank final field can also have default value.

```
class Alpha
{
  final int x ; //Blank final field

 public Alpha()
 {
    x = 100;
    System.out.println(x);
 }

 public Alpha(int y)
 {
  x = y;
  System.out.println(y);
 }

}
public class Test
{
```

```
 public static void main(String[] args)
 {
  Alpha a1 = new Alpha();
  Alpha a2 = new Alpha(200);
 }

}
```

A blank final filed must be initialized in all the objects.

Relationship between the classes :

In java we have 2 types of relation in between the classes :

a) IS-A Relation
b) HAS-A Relation

IS-A Relation :

```
class Car
{
}
class Ford extends Car //[Ford IS-A car ]
{
}
```

HAS-A Relation :

```
class Engine
{
}

class Car
{
   private Engine engine;  //Car HAS-A Engine.
}
```

IS-A relation we can achieve by using Inheritance Concept.
HAS-A relation we can achieve by using Association Concept.

==========

13-11-2024

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.

Types of Inheritance in java :

Java supports 5 types of inheritance :

1) Single level Inheritance
2) Multi level Inheritance
3) Hierarchical Inheritance
4) Multiple Inheritance (Not supported using class)
5) Hybrid Inheritance (Combination of two)

//Program on Single Level Inheritance :

```
package com.ravi.single_level_inheritance;

class Father
{
 public void house()
 {
  System.out.println("2 BHK house");
 }

}
class Son extends Father
{
 public void car()
 {
  System.out.println("Audi car");
 }
}

public class SingleLevel {

 public static void main(String[] args)
 {
  Son s1 = new Son();
  s1.car();
  s1.house();

 }

}
```

//Program on Single Level Inheritance :

```
package com.ravi.single_level_inheritance;
```

```
class Super
{
 private int x,y;

 public void setData(int x, int y)
 {
  this.x = x;
  this.y = y;
 }

 public int getX() {
  return x;
 }


 public int getY() {
  return y;
 }


}
class Sub extends Super
{
 public void showData()
 {
  System.out.println("x value is :"+getX());
  System.out.println("y value is :"+getY());
 }
}

public class SingleLevelDemo {

 public static void main(String[] args)
 {
    Sub s1 = new Sub();
    s1.setData(100, 200);
    s1.showData();

 }

}
```

Note : By default private varaible of super class is not available to sub class, getter is required.

How to initialize the super class properties :

---

super keyword is used to access the member or to access the memory of super class.

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 (Variable Hiding)
2) To access super class method (Method Overriding)
3) To access super class constructor. (Constructor Chaining)

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.

SupervarDemo.java

```
package com.ravi.super_keyword;

class Father
{
 protected double balance = 50000;

}
class Daughter extends Father
{
 protected double balance = 18000; //Variable Hiding

 public void getBalance()
 {
  System.out.println("Daughter balance is :"+this.balance);
  System.out.println("Father balance is :"+super.balance);
 }


}
public class SupervarDemo {

 public static void main(String[] args)
 {
  Daughter d = new Daughter();
  d.getBalance();

 }

}
```

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 (bottom 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.

```
class Alpha
{
 public String toString()
 {
  return "India";
 }
}
class Beta extends Alpha
{
 public String toString() //Method Overriding
 {
  System.out.println(super.toString());
  return "Hyderabad";
 }
}
public class SuperMethodCall
{
 public static void main(String[] args)
 {
  Beta b = new Beta();
  System.out.println(b.toString());
 }

}
```

Note :

From the above program, We will get two concepts

1) Compiler and JVM both will search the member of the class from bottom to top
2) In order to access super class method (super class memory) we should use super keyword in the sub class method body.

---

3) To access the 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 no argument 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 java 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.

CallingNoArgument.java

```
class Alpha
{
 public Alpha()
 {
  super();
  System.out.println("Alpha class");
 }
}
class Beta extends Alpha
{
 public Beta()
 {
  super();
  System.out.println("Beta class");
 }
}
public class CallingNoArgument
{
 public static void main(String[] args)
 {
  Beta b = new Beta();

 }

}
```

Case 2 :
---

super("Java") : Must be explicitly written by user in the first line of constructor [not inside a method]. It is used to call the parameterized constructor of super class.

```
package com.ravi.constructor_test;

class Super
{
 public Super(String str)
 {
  System.out.println("My Institute name is :"+str);
 }
}
class Sub extends Super
{
   public Sub()
   {
    super("NIT");
    System.out.println("No argument constructor of sub class");
   }
}
```

```
public class ParameterizedConstructor {

 public static void main(String[] args)
 {
  Sub s = new Sub();
 }

}
```

Program that describes default constructor and super() will be added by the compiler.

ConstructorDemo.java

```
package com.ravi.super_demo;

class Alpha
{
 public Alpha()
 {
  System.out.println("Alpha class Constructor!!!");
 }
}
class Beta extends Alpha
{}

class Gamma extends Beta
{
 public Gamma()
 {
  System.out.println("Gamma class Constructor!!!");
 }
}

public class ConstructorDemo {

 public static void main(String[] args)
 {
  new Gamma();
 }

}
```

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_test;

class Super
{
 public Super()
 {
   System.out.println("No argument constructor of Super class");
```

```
 }

 public Super(String str)
 {
  this();
  System.out.println("My Institute name is :"+str);
 }
}
class Sub extends Super
{
   public Sub()
   {
   super("NIT");
    System.out.println("No argument constructor of sub class");
   }
}
public class NoArgumentOfCurrentClass {

 public static void main(String[] args)
 {
  Sub s = new Sub();
 }

}
```

Case 4 :

---

this(15) : Must be explicitly written by user in the
first line of constructor. It is used to call
parameterized constructor of current class.

```
package com.ravi.constructor_test;

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()
 {
  System.out.println("No Argument Constructor of Derived class");
 }
```

```
}

public class ParameterizedConstructorOfCurrent
{
 public static void main(String[] args)
 {
  Derived d = new Derived();
 }

}
```

==========

Program on super keyword :

```
package com.ravi.constructor_test;

class Shape
{
 protected int x;

 public Shape(int x)
 {
  this.x = x;
  System.out.println("x value is :"+this.x);
 }
}

class Square extends Shape
{
 public Square(int side)
 {
  super(side);
 }

 public void getAreaOfSquare()
 {
  double area = x * x;
  System.out.println("Area of Square is :"+area);
 }
}



public class SuperDemo1 {

 public static void main(String[] args)
 {
  Square ss = new Square(5);
  ss.getAreaOfSquare();

 }

}
```

15-11-2024

//Program on Hierarchical Inheritance

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)

```
package com.ravi.hierarchical_demo;

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

class Shape
{
 protected int x;

 public Shape(int x)
 {
   this.x = x;
   System.out.println("x value is :"+x);
 }
}
class Circle extends Shape
{
 final double PI = 3.14;

 public Circle(int radius)
 {
  super(radius);
 }

 public void areaOfCircle()
 {
  double area = PI * x * x;

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

class Rectangle extends Shape
{
  protected int breadth;

  public Rectangle(int length, int breadth)
  {
```

```
  super(length);
  this.breadth = breadth;
  }

  public void areaOfRectangle()
  {
   double area = super.x * this.breadth;
   System.out.println("Area of Rectangle is :"+area);
  }
}

public class HierarchicalDemo {

 public static void main(String[] args)
 {
  Scanner sc = new Scanner(System.in);
  System.out.print("Enter the radius of Circle :");
  int radius = sc.nextInt();

  Circle circle = new Circle(radius);
  circle.areaOfCircle();

  System.out.print("Enter the length of the Rectangle :");
  int length = sc.nextInt();
  System.out.print("Enter the breadth of the Rectangle :");
  int breadth = sc.nextInt();

  Rectangle rectangle = new Rectangle(length, breadth);
  rectangle.areaOfRectangle();

  sc.close();

 }

}
```

//Program on Single Level Inheritance :

```
package com.ravi.inheritance;

class TemporaryEmployee {
 protected int employeeId;
 protected String employeeName;
 protected String employeeAddress;

 public TemporaryEmployee(int employeeId, String employeeName, String employeeAddress) {
  super();
  this.employeeId = employeeId;
  this.employeeName = employeeName;
  this.employeeAddress = employeeAddress;
 }

}

class PermanentEmployee extends TemporaryEmployee {
```

```
 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 + "]";
 }




}

public class SingleLevelInheritanceDemo
{
 public static void main(String[] args)
 {
  PermanentEmployee p = new PermanentEmployee(1, "John", "Ameerpet", "IT", "developer");
  System.out.println(p);
 }

}
```

```
//Program on Hierarchical Inheritance :
package com.ravi.hierarchical_demo;

class Employee
{
 protected double salary;
 public Employee(double salary)
 {
  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 HierarchicalDemo1 {

 public static void main(String[] args)
 {
  Developer developer = new Developer(45000);
  System.out.println(developer);

  Designer designer = new Designer(20000);
  System.out.println(designer);


 }

}
```

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)

```
    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)

```
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();
```

==========

**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 [15-NOV-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]

18-11-2024

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.

2 files :

Test.java [It is available in com.ravi.m1 package]

```
package com.ravi.m1;

public class Test
{
   protected int x = 500;
}
```

ELC.java   [It is 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); //x variable is accessible
 }                              because declared with
                                      Protected AM.
}
```

---

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.

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.

How Delegation Hierarchy algorithm works :-

Whenever JVM makes a request to class loader sub system to load the required .class file into JVM memory, first of all, class loader sub system makes a request to Application class loader, Application class loader will delegate(by pass) the request to the Extension class loader, Extension class loader will also delegate the request to Bootstrap class loader.

Bootstrap class loader will load the .class file from lib folder(jrt-fs.jar) and then by pass the request back to extension class loader, Extension class loader will load the .class file from ext folder(*.jar) and by pass the request back to Application class loader, It will load the .class file from environment variable into JVM memory.

Note :-

If all the class loaders are failed to load the .class file into JVM memory then we will get a Runtime exception i.e java.lang.ClassNotFoundException.

Note : java.lang.Object is the first class to be loaded into JVM Memory.

Note : Always Super class will be loaded before sub class loading.
[A child cannot exist without parent]

==========

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.

MethodChainingDemo1.java

```
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(0);
  System.out.println(ch);

 }

}
```

MethodChainingDemo2.java

```
package com.ravi.method_chaining;

public class MethodChainingDemo2 {

 public static void main(String[] args)
 {
  String str = "Hyderabad";
  int len =  str.concat(" is an IT city").toLowerCase().length();
  System.out.println(len);
 }

}
```

20-11-2024

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 class object.

```
   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.

```
package com.ravi.method_area;

class Employee{}

class Student{}

class Sample{}

public class ClassLoadingInformation
{
 public static void main(String[] args)
 {
  Class cls = Employee.class;
  System.out.println(cls.getName()); //FQN (Package name + class name)

  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 class loader name which is responsible to load the .class file into JVM Memory.

```
public ClassLoader getClassLoader()
```

ApplicationClassLoaderDemo.java

```
package com.ravi.method_area;

class Customer
{

}

public class ApplicationClassLoaderDemo
{
 public static void main(String[] args)
 {
  System.out.println("Customer.class file will be loaded by :");
  System.out.println(Customer.class.getClassLoader());

 }

}
```

WAP to describe that 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();
```

PlatformClassLoaderDemo.java

```
package com.ravi.method_area;

class Foo
{

}
public class PlatformClassLoaderDemo
{
 public static void main(String[] args)
 {
  System.out.println("Super class of application class loader is :");
  System.out.println(Foo.class.getClassLoader().getParent());

 }

}
```

//Program to show Bootstarp class loader

```
package com.ravi.method_area;

class Foo
```

```
{

}
public class PlatformClassLoaderDemo
{
 public static void main(String[] args)
 {
  System.out.println("Super class of platform class loader is :");
  System.out.println(Foo.class.getClassLoader().getParent().getParent());

 }

}
```

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.

Linking Phase :

---

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 even the variable is final]

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 (like #7) will be converted into direct references OR actual reference.

javap -verbose FileName.class

Note :- By using above command we can read the internal details of .class file.

Initialization :

Here class initialization will take place. All the static data member will get their actual/original 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)

21-11-2024

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.

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 variable is coming from class area OR Method area.

```
//static block
class Foo
{
```

```
 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.

```
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);
 }

 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.

```
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.

```
class Demo
{

 final static int a ; //Blank static final field

 static
 {
  m1();
  a = 100;
  System.out.println("User Value :"+a);
 }

 public static void m1()
 {
  System.out.println("Default Value :"+a);
 }


}
public class  StaticBlockDemo3
{
 public static void main(String[] args)
  {
        System.out.println("a value is :"+Demo.a);
    }
}
```

A static black final field must be initailized inside static block only and it also contains default value.

```
class  A        //AD  BC  EF
```

```
{
 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();
 }
}
```

22-11-2024

```
//illegal forward reference

class Demo
{
    static
 {
  i = 100;  //valid
 }

    static int i;
}
```

```
public class StaticBlockDemo5
{
    public static void main(String[] args)
 {
        System.out.println(Demo.i);
 }
}
```

```
class Demo
{
    static
 {
  i = 100;
  //System.out.println(i);  //Invalid
  System.out.println(Demo.i); //Valid
    }

    static int i;
}

public class StaticBlockDemo6
{

    public static void main(String[] args)
 {
  System.out.println(Demo.i);
 }
}
```

```
class StaticBlockDemo7
{
 static
 {
      System.out.println("Static Block");
   return;
 }

 public static void main(String[] args)
 {
  System.out.println("Main Method");
 }
}
```

Note : All the initializer must be executed normally so we can’t write return statement OR any transfer statement.

```
public class StaticBlockDemo8
{
    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);
 }
}
```

Note : A static blank final field must be initialized inside the
static block only as well as it has also default value.

```
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");
 }

}

public class StaticBlockDemo9
{
 public static void main(String[] args)
 {
  new Test();  //2 steps (class loading + Object creation)
 }
}
```

Note : First non static block, constructor then only static block will be executed.

==========

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.

```
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]

```
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 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

```
class Demo
{
   static
   {
      System.out.println("static block");
   }
}
public class Main
{
   public static void main(String [] args) throws ClassNotFoundException
   {
      Class.forName("Demo");
   }
}
```

---
loading .class file by using Class.forName(String className) at runtime using Eclipse IDE :

```
package com.ravi.static_block;

class Ravi
{
 static
 {
  System.out.println("Static Block of Ravi class");
 }
}

public class ClassLoading
{
 public static void main(String[] args) throws ClassNotFoundException
 {
  Class.forName("com.ravi.static_block.Ravi");
 }

}
```

Note : In eclipse IDE a class is represented by (FQN) Fully Qualified Name (Package Name + class name)

** 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.

```
package com.ravi.static_block;

class Ravi
{
 static
 {
  System.out.println("Static Block of Ravi class");
 }
}

public class ClassLoading
{
 public static void main(String[] args) throws ClassNotFoundException
 {
  Class.forName("Ravi");
 }

}
```

Note : In the above program we will get get java.lang.ClassNotFoundException because Ravi class is not identified by Application class loader, In Eclise IDE Fully Qualified Name is reqd.

---
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 Hello
{
 public void greet()
 {
  System.out.println("Hello Batch 39");
 }
}
public class NoClassDefFoundErrorDemo
{
 public static void main(String[] args)
 {
  Hello h = new Hello();
  h.greet();
 }
}
```

Note : 1) After compilation delete Hello.class file from the current folder and execute the program, we will get java.lang.NoClassDefFoundError

2) In order to solve the issue , we need to re-compile the code to generate a new .class file.

** A static method does not act on instance variable directly why?

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]

Test.java

```
public class Test
{
 int x = 100;

 public static void main(String[] args)
 {
  System.out.println(x); //error
 }
}


class Test
{
 private int x;

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

 public static void access()
 {
  System.out.println(x); //error
 }
}
public class StaticDemo
{
 public static void main(String[] args)
 {
  Test t1 = new Test(10);
  Test.access();
 }
}
```

==========

Accessing variable of Sub class and Super class by using static method.

```
package com.ravi.inheritance;

class Super
{
 protected int x = 100;
}
class Sub extends Super
{
 protected int x = 200; //Variable Hiding

 public static void access()
 {
  Sub s1 = new Sub();
  System.out.println(s1.x);

  Super s2 = s1;  //Up casting
  System.out.println(s2.x);
 }
}

public class StaticTest
{
 public static void main(String[] args)
 {
  Sub.access();
 }

}
```

==========

25-11-2024

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

```
package com.ravi.class_info;

import java.util.Scanner;

public class Test
{
   int x = 100;
   static Scanner sc = new Scanner(System.in);
   static int y = 200;
   int z = 300;

   public void input() {}

   public static void accept() {}

   public void display() {}

   public void show() {}

   public void m1() {}

}
```

ClassInformationDemo.java

```
package com.ravi.class_info;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ClassInformationDemo
{
 public static void main(String[] args) throws ClassNotFoundException
 {
      Class cls =  Class.forName(args[0]);

      System.out.println("Class Name is :"+cls.getName());
      System.out.println("Package Name is :"+cls.getPackageName());

      Method[] methods = cls.getDeclaredMethods();
      System.out.println("Method names are :");
      int count = 0;
      for(Method method : methods)
      {
       System.out.println(method.getName());
       count++;
      }
```

```
    System.out.println("Total number of Methods are :"+count);

    count = 0;
    Field[] fields = cls.getDeclaredFields();

    System.out.println("Available fields are :");

    for(Field field : fields)
    {
     System.out.println(field.getName());
     count++;
    }
    System.out.println("Total number of fields are :"+count);
 }
}
```

javac ClassInformationDemo.java
java ClassInformationDemo FQN of the class

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.

Everytime we create a thread in java then JVM will create a separate Runtime Stack.[Multithreading]

==========
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;
 }
}

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;
        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;
 }
}
```

26-11-2024

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.

Execution Engine : [Interpreter + JIT Compiler]

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 (Engine class) as a property to another class (Car 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

```
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 void setStudentId(int studentId) {
  this.studentId = studentId;
 }

 public String getStudentName() {
  return studentName;
 }

 public void setStudentName(String studentName) {
  this.studentName = studentName;
 }

 public int getStudentMarks() {
  return studentMarks;
 }

 public void setStudentMarks(int studentMarks) {
  this.studentMarks = studentMarks;
 }

 @Override
 public String toString() {
  return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentMarks=" +
studentMarks
    + "]";
 }

}
```

Trainer.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! No such student with given id");
  }
  sc.close();

 }

}
```

AssociationDemo.java

```
package com.ravi.association;

public class AssociationDemo {

 public static void main(String[] args)
 {
  Student raj = new Student(1, "Raj", 88);
  Trainer.viewStudentProfile(raj);

 }

}
```

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

```
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

 public Car(String carName, int carModel)
 {
  super();
  this.carName = carName;
  this.carModel = carModel;
  this.engine = new Engine("Battery", 1200); // Composition
 }

 @Override
 public String toString()
 {
  return "Car [carName=" + carName + ", carModel=" + carModel + ", engine=" + engine + "]";
 }

}
```

CompostionDemo.java

```
package com.ravi.composition;

public class CompostionDemo
{
 public static void main(String[] args)
 {
  Car naxon = new Car("Tata Naxon", 2024);
  System.out.println(naxon);

 }

}
```

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.

College.java

```
package com.ravi.aggregation;

public class College
{
 private String collegeName;
 private String collgeLocation;

 public College(String collegeName, String collgeLocation)
 {
  super();
  this.collegeName = collegeName;
  this.collgeLocation = collgeLocation;
 }

 @Override
 public String toString() {
  return "College [collegeName=" + collegeName + ", collgeLocation=" + collgeLocation + "]";
 }




}
```

Student.java

```
package com.ravi.aggregation;

public class Student {
```

```
 private int studentId;
 private String studentName;
 private College collge; // HAS-A Relation

 public Student(int studentId, String studentName, College collge)
 {
  super();
  this.studentId = studentId;
  this.studentName = studentName;
  this.collge = collge;
 }

 @Override
 public String toString() {
  return "Student [studentId=" + studentId + ", studentName=" + studentName + ", collge=" + collge + "]";
 }


}
```

AggregationDemo.java

```
package com.ravi.aggregation;

public class AggregationDemo {

 public static void main(String[] args)
 {
  College c1 = new College("VIT", "Vellore");


  Student s1 = new Student(1, "Scott", c1);

  Student s2 = new Student(2, "Martin", c1);
  Student s3 = new Student(3, "Smith", c1);

  System.out.println(s1);
  System.out.println(s2);
  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.

==========

27-11-2024

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 :

```
package com.ravi.s_o_p;

class Test
{
 static final String str = "Hyderabad";
}

public class Description {

 public static void main(String[] args)
 {
  System.out.println(Test.str.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

==========
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 us to write two methods with same name but 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)
{
}
```

28-11-2024

Program on Constructor Overloading :

```
package com.ravi.overloading;

class Calculate
{
  public Calculate()
  {
   this(10,20);
  }
  public Calculate(int x, int y)
  {
   this(100,200,300);
   System.out.println("Sum of two integer is :"+(x+y));
  }
  public Calculate(int x, int y, int z)
  {
   System.out.println("Sum of three integer is :"+(x+y+z));
  }
}

public class ConstructorOverloading {

 public static void main(String[] args)
 {
  new Calculate();
 }

}
```

Program on Method Overloading :

```
package com.ravi.overloading;

class Addition
```

```
{
 public int add(int x, int y)
 {
  return x+y;
 }

 public double add(double x, double y)
 {
  return x+y;
 }

 public String add(String x, String y)
 {
  return x+y;
 }
}

public class MethodOverloading
{
 public static void main(String[] args)
 {
  Addition a1 = new Addition();
  int sum = a1.add(12, 24);
  System.out.println("Sum of two integer is :"+sum);

  double add = a1.add(2.3, 8.9);
  System.out.println("Sum of two double is :"+add);

  String concat = a1.add("Data", "base");
  System.out.println("String after Concatenation :"+concat);

 }

}
```

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.

```
package com.ravi.overloading;

class Test
{
 public void input(int  ...x)
```

```
 {
  System.out.println("Var args executed");
 }
}

public class VarArgsDemo1 {

 public static void main(String... args)
 {
  Test t1 = new Test();
  t1.input();
  t1.input(5);
  t1.input(5,10);
  t1.input(5,10,20);
 }

}
```

Note : We can pass 0 to n numbewr of parameters.

//Finding the sum of parameters

```
package com.ravi.overloading;

class AddParameter
{
 public void acceptAndAddParameter(int ...values)
 {
  int sum = 0;
  for(int value : values)
  {
   sum = sum + value;
  }
  System.out.println("Sum of parameter is :"+sum);
 }
}

public class VarArgsDemo2
{
 public static void main(String[] args)
 {
  AddParameter a = new AddParameter();
  a.acceptAndAddParameter(10,20,30);
  a.acceptAndAddParameter(100,200,300,400);

 }

}
```

//We can hetrogeneous types of data

```
package com.ravi.overloading;

class Hetro
{
```

```
 public void acceptHetro(Object ...x)
 {
  for(Object y : x)
  {
   System.out.println(y);
  }
 }
}

public class VarArgsDemo3 {

 public static void main(String[] args)
 {
  Hetro h = new Hetro();
  h.acceptHetro(12,89.90,'C',34.89, new String("NIT"));

 }

}
```

```
//Var args must be only one and last argument.
package com.ravi.overloading;

class Demo
{
 // 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);
  }
 }
}


public class VarArgsDemo4
{
   public static void main(String[] args)
   {
   Demo d = new Demo();
   d.accept(12,10,20,30,40);
   }
```

}

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.It is only possible with objects.

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() methods are overridden in all the Wrapper classes.

WAP to show that Wrapper classes are immutable so if we modify try to modify the object then original object will not be modified.

```
package com.ravi.aggregation;

public class ImmutableDemo {

 public static void main(String[] args)
 {
  Integer i = new Integer(12);
        accept(i);
        System.out.println(i);
 }

 public static void accept(Integer y)
```

```
 {
  y = 22;
 }

}
```

String is also an immutable class as shown in the program.

```
package com.ravi.aggregation;

public class ImmutableDemo2
{
 public static void main(String[] args)
 {
  String str = "india";
  accept(str);
  System.out.println(str);
 }

 public static void accept(String s1)
 {
  s1 = "Hyd";
 }

}
```

```
//Integer.valueOf(int);
public class AutoBoxing1
{
 public static void main(String[] args)
 {
  int a = 12;
  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 :

We have 3 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 specified 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

System.out.println(Character.MAX_RADIX); //36
MAX_RADIX is a final and static variable of Character class.

```
-----------------------------------------------------------------
//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);
 }
}
```

30-11-2024

Unboxing :

Converting wrapper object to corresponding primitive type is called Unboxing.

Wrapper Object	Primitive 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 (Byte, Short, Integer, Long, Float and Double) are the sub class of java.lang.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()

```
//Converting Wrapper object into primitive
public class AutoUnboxing1
{
   public static void main(String args[])
    {
           Integer obj = 15;   //Upto 1.4
   int x = obj.intValue();
   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);
 }
}
-----------------------------------------------------------------
class BufferTest
{
 public static void main(String[] args)
 {
  Integer i = 127;
  Integer j = 127;
  System.out.println(i==j);
  System.out.println(i.equals(j));

  Integer a = 128;
  Integer b = 128;
  System.out.println(a==b);
  System.out.println(a.equals(b));

  Integer p = 130;
  Integer q = 130;
```

```
  System.out.println(p.equals(q));

 }
}
```

Note : 1) While comparing the Wrapper object we should always use
equals(Object obj) method which is overridden method
in the Wrapper classes.

2) Here when we write the statement Integer i = 128 then it is out of the range of byte (-128 to 127) hence == opertor will provide false if we compare two Integer object.

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 case of ambiguity where compiler can select more than one method then compiler will provide the priority in 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 //[No relation between short and char]
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);

 }
}
```

Note : Here float will be executed becuase float is the most specific type.

```
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 6 is int type so int will be executed.

```
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();
 }
}
```

char will be executed becoz char is more 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);  //error
  t1.accept((byte)15);
  t1.accept((short)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");
 }
}
public class AmbiguityIssue
{

 public static void main(String[] args)
 {
  Test t = new Test();
  t.accept(9,9);
 }
}
```

Here We will get ambiguity issue.

```
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

```
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("NIT");

 }
}
```

Here String will be executed

```
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.

```
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.

```
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.

```
class Test
{
 public void accept(Number n)
 {
  System.out.println("Number");
 }
 public void accept(Double d)
 {
  System.out.println("Double");
 }
}
public class AmbiguityIssue {

 public static void main(String[] args)
 {
  Test t = new Test();
  t.accept(12);

    }
}
```
Here Number will be executed.

02-12-2024

***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 as well as return type must be compaitable 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 (compatibility issue) 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.

**What is upcasting and downcasting ?

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
```

Note : To avoid this ClassCastException we should use instanceof opertor.

Downcasting is a technique to assign sub class object (Only reference is super type) to sub class reference variable as shown below.

```
Animal a1 = new Lion();
Lion l = (Lion) a1;  //Downcasting
```

Downcasting is not possible without upcasting.

```
package com.ravi.mor;

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 OverridingDemo1
{
 public static void main(String[] args)
 {
  Bird b1 = null;

  b1 = new Parrot();  b1.fly();  //Dynamic Method Dispatch
  b1 = new Sparrow();  b1.fly();  //Dynamic Method Dispatch

 }

}
------------------------------------------------------------
package com.ravi.mor;

class Animal
{
 public void eat()
 {
  System.out.println("Generic Animal is eating");
 }
}
class Dog extends Animal
{
 public void eat()
 {
  System.out.println("Dog Animal is eating");
 }
}
class Puppy extends Dog
{

}

public class OverridingDemo2 {

 public static void main(String[] args)
       {
  Animal a1 = new Puppy();
  a1.eat();

 }

}
```

Here compiler will search the eat method in Animal class where as JVM will start executing from Puppy class, Dog class, Animal class, Object class.

@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.

```
package com.ravi.mor;

class Shape
{
 public void draw()
 {
    System.out.println("Generic Draw");
 }
}
class Rectangle extends Shape
{
 @Override
 public void draw()
 {
    System.out.println("Drawing Rectangle");
 }
}

class Square extends Shape
{
 @Override
 public void draw()
 {
    System.out.println("Drawing Square");
 }
}

public class OverridingDemo3
{
 public static void main(String[] args)
 {
  Shape s = null;
  s = new Rectangle();  s.draw();
  s = new Square(); s.draw();
 }

}
```

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.

Note : static variable, non static variable and static method
are always executed using current reference.

```
package com.ravi.mor;

class RBI
{
 protected String ifscCode = "RBIHYD09675";

 public String loan()
 {
  return "Bank should provide loan";
 }
}
class SBI extends RBI
{
 protected String ifscCode = "SBIAMPT78645";  //Variable Hiding

 @Override
 public String loan()
 {
  return "Providing loan @ 9.2% ROI";
 }
}

public class OverridingDemo4
{
 public static void main(String[] args)
 {
  RBI r = new SBI();

  System.out.println(r.ifscCode+" : "+r.loan());

 }
```

```
}
```

Can we override private method ?

No, We can’t override private method 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.

```
package com.ravi.mor;

class Super
{
 private void m1()
 {
  System.out.println("Private Method of super class");
 }
}
class Sub extends Super
{
 protected void m1()  //Re-declaration of Method
 {
  System.out.println("Method has re-declared");
 }
}


public class OverridingDemo5 {

 public static void main(String[] args)
 {
  new Sub().m1();

 }

}
```

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.

04-12-2024

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.

```
class Super
{
 public void m1()
 {
 }
}
class Sub extends Super
{
 @Override
 protected void m1()  //error [super class method AM
                          is public ]
 {
 }
}
public class OverridingDemo6
{
 public static void main(String[] args)
 {
  Super s1 = new Sub();
  s1.m1();
 }
}
```

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 m1()
 {
 }
}
class Sub extends Super
{
 @Override
 public int m1()  //error [int is not compaitable with
                       void]
 {
  return 0;
 }
}
public class OverridingDemo7
{
 public static void main(String[] args)
 {
  Super s1 = new Sub();
  s1.m1();
 }
}
```

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.

```
class Alpha
{
}
class Beta extends Alpha
{
}

class Super
{
 public Alpha m1()
 {
  System.out.println("Super class Method");
  return new Alpha();
 }
}
class Sub extends Super
{
 @Override
 public Beta m1()
 {
  System.out.println("Sub class Method");
  return new Beta();
 }
}
public class OverridingDemo8
{
 public static void main(String[] args)
 {
  Super s1 = new Sub();
  s1.m1();
 }
}
```

Note : Here we need to verify one concept, can we assign Beta class Object to Alpha class, if yes then it is compitable.

```
class Super
{
 public Super m1()
 {
  System.out.println("Super class Method");
  return this;
 }
}
class Sub extends Super
{
 @Override
```

```
 public Sub m1()
 {
  System.out.println("Sub class Method");
  return this;
 }
}
public class OverridingDemo9
{
 public static void main(String[] args)
 {
  Super s1 = new Sub();
  s1.m1();
 }
}
```

```
package com.ravi.execution;

class A
{
 public Object m1()
 {
  System.out.println("Super class m1 method");
  return this;
 }
}
class B extends A
{
 @Override
 public System m1()
 {
  System.out.println("Sub class m1 method");
  return null;
 }


}
public class OverridingDemo10 {

 public static void main(String[] args)
 {
  A a1 = new B();
  a1.m1();
 }

}
```

While working with co-variant (In the same direction), sub class method return type object, if we can assign to super class method return then only it is compatible and it is co-variant

---

IQ :

```
package com.ravi.polymorphic_behavior;

class Vehicle
```

```
{
 public int getHorsePower()
 {
  return 1000;
 }

 public void printHorsePower()
 {
  System.out.println(this.getHorsePower());
 }

}
class Car extends Vehicle
{
 public int getHorsePower()
 {
  return 1200;
 }

 public void printHorsePower()
 {
  System.out.println(super.getHorsePower());
 }

}


public class IQ {

 public static void main(String[] args)
 {
  Vehicle v = new Car();
  v.printHorsePower();
 }

}
```

Progrm that describes Polymorphic behaviour of sub classes :

Case 1 :

```
package com.ravi.polymorphic_behavior;

class Animal
{
 public void roam()
 {
  System.out.println("Generic Animal is roaming");
 }
}
class Lion extends Animal
{
 public void roam()
 {
  System.out.println("Lion Animal is roaming");
```

```
 }
}
class Dog extends Animal
{
 public void roam()
 {
  System.out.println("Dog Animal is roaming");
 }
}
public class PolymorphicDemo1
{
 public static void main(String[] args)
 {
       Animal a = null;
       a = new Lion();
       animalRoam(a);

       a = new Dog();
       animalRoam(a);
 }

 public static void animalRoam(Animal animal)
 {
  animal.roam();
 }

}
```

Case 2 :

How to call specific method of sub class :

```
package com.ravi.polymorphic_behavior;

class Animal
{
 public void roam()
 {
  System.out.println("Generic Animal is roaming");
 }
}
class Lion extends Animal
{
 public void roam()
 {
  System.out.println("Lion Animal is roaming");
 }

 public void roar()
 {
  System.out.println("Lion is roaring");
 }
}
class Dog extends Animal
{
```

```
 public void roam()
 {
  System.out.println("Dog Animal is roaming");
 }

 public void bark()
 {
  System.out.println("Dog is Barking");
 }
}
public class PolymorphicDemo1
{
 public static void main(String[] args)
 {
       Animal a = new Lion();
       animalRoam(a);

       a = new Dog();
       animalRoam(a);
 }

 public static void animalRoam(Animal animal)
 {

     Lion lion = (Lion) animal;
     lion.roam();
     lion.roar();
 }

}
```

In the above program when we pass Dog object then we will get Runtime Exception java.lang.ClassCastException becuase Dog can’t be converted into Lion.

In order to resolve this issue we should use instanceof Operator.

---

instanceof Operator :

It is an operator as well as keyword.
It is relational operator which provides true/false.
It is used to verify whether a reference variable is pointing to a particular type of object or not ?
We must have IS-A relation between the reference variable and class or interface type.

Programs :

```
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 pointing to Test Object");
    }

 }

}
```

```
package com.ravi.instance_of;

class Alpha
{
}
class Beta extends Alpha
{
}
class Gamma extends Beta
{
}

public class InstanceDemo2
{
 public static void main(String[] args)
 {
  Gamma g = new Gamma();

  if(g instanceof Gamma)
  {
   System.out.println("g is pointing to Gamma Object");
  }

  if(g instanceof Beta)
  {
   System.out.println("g is pointing to Beta Memory");
  }

  if(g instanceof Alpha)
  {
   System.out.println("g is pointing to Alpha Memory");
  }

  if(g instanceof Object)
  {
   System.out.println("g is pointing to Object Memory");
  }
 }


}
```

```
package com.ravi.instance_of;

public class InstanceDemo3 {

 public static void main(String[] args)
 {
  String str = "india";

  if(str instanceof String)
  {
   System.out.println("str is pointing to String Object");
  }

  Integer i = 90;

  if(i instanceof Number)
  {
   System.out.println("i is pointing to Integer Object");
  }

 }

}
```

```
package com.ravi.instance_of;

class Bird
{

}
class Parrot extends Bird{}

class Sparrow extends Bird{}

public class InstanceDemo4
{
 public static void main(String[] args)
 {
  Parrot p = new Parrot();
  Sparrow s = new Sparrow();

  acceptBirdType(s);
 }

 public static void acceptBirdType(Bird bird)
 {
  if(bird instanceof Parrot)
  {
   System.out.println("It is a Parrot object");
  }
  else
  {
   System.out.println("Another Object");
  }
```

```
 }

}
```

Dynamic Polymorphism with the help of instanceof Operator.

```
package com.ravi.mor;

class Payment
{
 public double makePayment(double amount)
 {
  return amount;
 }
}

class UPI extends Payment
{
 @Override
 public double makePayment(double amount)
 {
  System.out.println("Making a payment of "+amount+" through UPI");
  return amount;
 }

 public void offer()
 {
  System.out.println("Make first payment and get 100 RS");
 }
}

class CreditCard extends Payment
{
 @Override
 public double makePayment(double amount)
 {
  System.out.println("Making a payment of "+amount+" through Credit Card");
  return amount;
 }

 public void offer()
 {
  System.out.println("Make first payment and get holiday ticket");
 }
}

public class DynamicPolyInstanceOf
{
 public static void main(String[] args)
 {
  Payment p = null;
  p = new UPI();
  acceptPayment(p);
```

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

  p = new CreditCard();
  acceptPayment(p);

 }

 public static void acceptPayment(Payment payment)
 {
  if(payment instanceof UPI)
  {
   UPI u = (UPI) payment;
   u.makePayment(20000);
   u.offer();
  }

  else if(payment instanceof CreditCard)
  {
   CreditCard cc = (CreditCard) payment;
   cc.makePayment(35000);
   cc.offer();
  }
 }

}
```

**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

```
class Parent
{
 public static void show()
 {
   System.out.println("Show method of Parent class");
 }
}
class Child extends Parent
{
}
public class MethodHidingDemo1
{
 public static void main(String[] args)
 {
```

```
  Child.show();

  Child c1 = new Child();
  c1.show();
 }
}
```

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 Super
{
 public static void m1()  //class
 {
 }
}
class Sub extends Super
{
 public void m1()  //object
 {
 }
}
public class MethodHidingDemo2
{
 public static void main(String[] args)
 {
  System.out.println("Hello World!");
 }
}
```

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 Super
{
 public  void m1()  //Object
 {
 }
}
class Sub extends Super
{
 public static void m1()  //class
 {
 }
}
public class MethodHidingDemo3
{
 public static void main(String[] args)
 {
  System.out.println("Hello World!");
```

```
 }
}
```

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.

Case 4 :

Program that describes method hiding concept as well as sub class method can’t hide super class method because return type is not compaitable.

```
class Super
{
 public static void m1()  //class
 {
 }
}
class Sub extends Super
{
 public static int m1()  //class
 {
  return 0;
 }
}
public class MethodHidingDemo2
{
 public static void main(String[] args)
 {
  System.out.println("Hello World!");
 }
}
```

Note : sub class method can’t hide super class method becuase return type is not compaitable

---

case 5 :

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 and common for 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.

```
package com.ravi.mor;

class Base
{
 public static void m1()
 {
  System.out.println("Static Method of Base class");
 }
```

```
}
class Derived extends Base
{

 public static void m1() //Method Hiding [Static Polymorphism]
 {
  System.out.println("Static Method of Derived class");
 }
}
public class MethodHidingEx {

 public static void main(String[] args)
 {
  Base b1 = new Derived();
  b1.m1();


 }

}
```

06-12-2024

---

*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.

newInstance() method creates the object internally by using new keyword only and the class must contain either default OR no
argument constructor.

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. The return type of this method is java.lang.Class so further we can call any method of java.lang.Class class object

```
getClass().getName();
```

ObjectAtRuntime.java

```
class Student
{
}
class Employee
{
}
public class ObjectAtRuntime
{
 public static void main(String[] args) throws Exception
 {
  Object obj = Class.forName(args[0]).newInstance();
  System.out.println("Object created for :"+obj.getClass().getName());
 }
}
```

```
class Student
{
 public void greet()
 {
  System.out.println("Welcome Student");
 }
}
class Sample
{
 public void greet()
 {
   System.out.println("Hello Batch 39!!!!");
 }
}
public class ObjectAtRuntime1
{
 public static void main(String[] args) throws Exception
 {
  Object obj = Class.forName(args[0]).newInstance();

  if(obj instanceof Sample)
  {
     Sample s1 = (Sample) obj;
     s1.greet();
  }
  else if(obj instanceof Student)
  {
     Student s1 = (Student) obj;
     s1.greet();
  }
 }
}
```

---

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.

```
final class A
{
 private int x = 100;

 public void setData()
 {
  x = 120;
  System.out.println(x);
 }
}
class B extends A
{
}
public class FinalClassEx
{
 public static void main(String[] args)
 {
  B b1 = new B();
  b1.setData();
 }
}
```

Note : class A is final so we can't inherit hence we will get compilation error.

```
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);

 }
}
```

Note : for final class we can create object as well as we can modify the data.

Whenever we declare a constructor as private then we should declare the class with final modifier. If constructor is private then we can't create a sub class because super class constructor is not visible from sub class constructor.

```
final class Sample
{
 private Sample()
 {
 }
 public void m1()
 {
  System.out.println("Sample class m1 method");
 }
}

public class FinalClassEx2
{
 public static void main(String[] args)
 {

 }
}
```

07-12-2024

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 OR sealed interface

4) final : we can declare permitted sub class as final so, it cannot be extended further.

```
package com.ravi.sealed_ex;

sealed class Bird permits Parrot, Sparrow
{
 public void roam()
 {
  System.out.println("Generic Bird is roaming");
 }
}
non-sealed class Parrot extends Bird
{
 @Override
 public void roam()
 {
  System.out.println("Parrot Bird is roaming");
 }
}
final class Sparrow extends Bird
{
 @Override
 public void roam()
 {
  System.out.println("Sparrow Bird is roaming");
 }
}

public class SealedDemo1
{
 public static void main(String[] args)
 {
  Bird b = null;
  b = new Parrot(); b.roam();
  b = new Sparrow(); b.roam();
 }

}
```

```
package com.ravi.sealed_ex;

sealed class BatteryVehicle permits BatteryCar, BatteryBike
{
 public void run()
 {
  System.out.println("Running using Battery");
 }
}
```

```
non-sealed class BatteryCar extends BatteryVehicle
{
 public void run()
 {
  System.out.println("Running Car by using Battery");
 }
}

non-sealed class BatteryBike extends BatteryVehicle
{
 public void run()
 {
  System.out.println("Running Bike by using Battery");
 }
}

public class SelaedDemo2 {

 public static void main(String[] args)
 {
  BatteryVehicle b = null;
  b = new BatteryCar(); b.run();
  b = new BatteryBike(); b.run();
 }

}
```

---

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.

```
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() //error
 {
  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();
 }
}
```

Note : We can’t oevrride final method in the sub class.

```
class Alpha
{
 private final void accept()
 {
  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.

```
class A
{
 final int A = 10;
 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();
  a1.setData();
 }
}
```

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.

```
abstract class Shape
{
 public abstract void draw();
}

class Square extends Shape
{
 @Override
 public void draw()
 {
  System.out.println("Drawing Square");
 }

}
class Circle extends Shape
{
 @Override
 public void draw()
 {
  System.out.println("Drawing Circle");
 }

}

public class AbstractDemo1
{
 public static void main(String[] args)
 {
    Shape s1 = null;
    s1 = new Circle(); s1.draw();
    s1 = new Square(); s1.draw();

 }
}
```

09-12-2024

```
package com.ravi.abstract_demo;

abstract class Bike
{
 protected int speed = 60;

 public Bike()
 {
  System.out.println("Bike Constructor");
 }
```

```
 public void getBikeDeatils()
 {
  System.out.println("It has two wheels");
 }

 public abstract void run();

}

class KTM extends Bike
{
 @Override
 public void run()
 {
  System.out.println("KTM Bike is running");
 }
}

public class AbstractDemo2 {

 public static void main(String[] args)
 {
  Bike obj = new KTM();
  System.out.println("SPEED IS :"+obj.speed);
  obj.getBikeDeatils();
  obj.run();
 }

}
```

IQ :

What is the advantage of taking instance variable OR writing constructor inside abstract class ?

As we know we can't create an object for abstract class but still we can take object properties (Instance variable) and
constructor, To call the abstract class constructor for initialization of instance variable we should use sub class object (Using super keyword)

[Note : Even at the time of working with inheritance concept, to initialize the super class instance variable through super class constructor, super class object is not required, by creating the object of sub class, we can initialize super class properties(Instance variable)]

//Program that describes how to initialize super class properties :

```
package com.ravi.abstract_demo_ex;

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);
 }

}
public class AbstractDemo3 {

 public static void main(String[] args)
 {
  Shape s = null;
  s = new Rectangle("Rectangle"); s.draw();
  s = new Circle("Circle"); s.draw();
 }

}
```

```
//Program that describes we should compulsory override all
  the abstract methods of super class in sub classes.

package com.ravi.abstract_demo_ex;

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)
 {
  Gamma g = new Gamma();
  g.show(); g.demo();


 }

}
```

WAP to force the sub class developer to implement super class
abstract method by using Array concept.

```
package com.ravi.abstract_demo_ex;

abstract class Animal
{
 public abstract void checkup();
}
class Lion extends Animal
{
 protected String name;

 public Lion(String name)
 {
  super();
  this.name = name;
 }

 @Override
 public void checkup()
 {
  System.out.println(name+ " Lion is going for Checkup");
 }

}
```

```
class Elephant extends Animal
{
protected String name;

 public Elephant(String name)
 {
  super();
  this.name = name;
 }

 @Override
 public void checkup()
 {
  System.out.println(name+ " Elephant is going for Checkup");
 }
}

class Horse extends Animal
{
protected String name;

 public Horse(String name)
 {
  super();
  this.name = name;
 }

 @Override
 public void checkup()
 {
  System.out.println(name+ " Horse is going for Checkup");
 }
}

public class AbstractDemo5 {

 public static void main(String[] args)
 {
        Lion lions[] = {new Lion("Simba"),new Lion("myLion")};

        Elephant elephants[] = {new Elephant("Erawat"), new Elephant("jambo")};

        Horse horses[] = {new Horse("Chetak"), new Horse("MyHorse")};

        visitZooForCheckup(lions);
        System.out.println(".....");
        visitZooForCheckup(elephants);
        System.out.println(".....");
        visitZooForCheckup(horses);
 }

 public static void visitZooForCheckup(Animal ...animals)
 {
  for(Animal animal : animals)
  {
```

```
   animal.checkup();
  }
 }

}
```

10-12-2024

Anonymous inner class with abstract class and Concrete class.

What is Anonymous inner class ?

If we define a class inside a method body without any name then it is called Anonymous inner class.

All the inner class .class files are represented by $ symbol.

The main purpose of anonymous inner class to extend a class OR to implement an interface that means creating a sub type.

An anonymous inner class object will be created by using new keyword at the time of defining the anonymous inner class.

Program on Anonymous inner class using Concrete class :

```
package com.rvai.anonymous_inner_demo;

class Super
{
 public void show()
 {
  System.out.println("Super class show method");
 }
}
public class AnonymousInnerDemo1
{
 public static void main(String[] args)
 {
  //Anonymous inner class
  Super sub = new Super()
  {
   @Override
   public void show()
   {
    System.out.println("Sub class show method");
   }
  };

  sub.show();
 }

}
```

Program on Anonymous inner class using abstract class :

```
package com.rvai.anonymous_inner_demo;

abstract class Vehicle
{
 public abstract void run();
}

public class AnonymousInnerDemo2
{
 public static void main(String[] args)
 {
  //Anonymous inner class
  Vehicle car = new Vehicle()
  {
   @Override
   public void run()
   {
    System.out.println("Car is Running");
   }

  };

  //Anonymous inner class
  Vehicle bike = new Vehicle()
  {
   @Override
   public void run()
   {
    System.out.println("Bike is Running");
   }
  };
  car.run();  bike.run();
 }

}
```

---

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 methods 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.nit.interface_demo;

sealed interface Moveable permits Car
{
 int SPEED = 100; //public + static + final
 void move();  //public + abstract
}

non-sealed class Car implements Moveable
{
 @Override
 public void move()
 {
  //SPEED = 120;   [Invalid]
  System.out.println("Car is Moving with :"+SPEED+ "KM/Hr");

 }
}

public class InterfaceDemo1
{
 public static void main(String[] args)
 {
    Moveable m = new Car();
    System.out.println("SPEED of Car is :"+Moveable.SPEED);
    m.move();
 }

}
```

```
package com.nit.interface_demo;

interface Bank
{
 void deposit(double amount);
 void withdraw(double amount);
}
class Customer implements Bank
{
 double balance;
```

```
 public Customer(double balance)
 {
  super();
  this.balance = balance;
 }

 @Override
 public void deposit(double amount)
 {
  if(amount<=0)
  {
     System.err.println("deposit is not possible");
  }
  else
  {
   this.balance = this.balance + amount;
   System.out.println("After deposit amount is :"+this.balance);
  }
 }

 @Override
 public void withdraw(double amount)
 {
  if(amount > this.balance)
  {
   System.err.println("Insufficient Balance");
  }
  else
  {
   this.balance = this.balance - amount;
   System.out.println("Balance after withdraw is :"+this.balance);
  }

 }
}

public class InterfaceDemo2
{
 public static void main(String[] args)
 {
  Bank b = new Customer(10000);
  b.deposit(10000);
  b.withdraw(5000);
 }

}
```

11-12-2024

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 is reqd]

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.

//Program on loose coupling :
---
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
 public void prepare()
 {
   System.out.println("Preparing Tea");
 }

}
```

Coffee.java

```
package com.ravi.loose_coupling;

public class Coffee implements HotDrink {

 @Override
 public void prepare()
 {
  System.out.println("Preparing Coffee");

 }

}
```

Horlicks.java

```
package com.ravi.loose_coupling;

public class Horlicks implements HotDrink {

 @Override
 public void prepare()
 {
  System.out.println("Preparing Horlicks");

 }

}
```

Restaurant.java

```
package com.ravi.loose_coupling;

public class Restaurant
{
   public static void acceptObject(HotDrink hd)
   {
   hd.prepare();
   }
}
```

LooseCoupling.java

```
package com.ravi.loose_coupling;

public class LooseCoupling
{
 public static void main(String[] args)
 {
  Restaurant.acceptObject(new Tea());
  Restaurant.acceptObject(new Coffee());
        Restaurant.acceptObject(new Horlicks());
 }

}
```

Method retutn type as a interface :

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.....................
}
```

Compile time constant :

A compile time constant is a constant that is evaluated and replaced with its value at compile time rather than runtime.

It must be declared with static and final modifier as well as initialized with constant expression. (Must not be initialized by method call)

At compile time constant value will be converted by compiler at the time of compilation itself so, at runtime JVM can see the value but not the class name so class will not be loaded as shown in the program.

Example : public static final int A = 100; //Valid

public static final int A = m1(); //valid [Here class will be loaded by JVM]

CompileTimeConstant.java

```
class Alpha
{
 static
 {
    System.out.println("Static block of Alpha class");
 }

 public static final int A = 100;
}


public class CompileTimeConstant
{
 public static void main(String[] args)
 {
  System.out.println(Alpha.A);
 }
}
```

The following program explains that compiler will convert the final, static variable value at the time of compilation itself
(so compile time OR early binding)

2 files :

Beta.java

```
public class Beta
{
 public static final int D = 1200;
```

```
}
```

Main.java

```
public class Main
{
 public static void main(String[] args)
 {
  System.out.println(Beta.D); //1000
 }
}
```

Instruction :
---
1) Compile both the program and execute Main.java

2) Change the value of D variable, re-compile the code

3) Execute Main code without compilation.

compile time constant with interface :

```
package com.ravi.loose_coupling;

interface Hello
{
 public static final int X = 100;
}

public class MainDemo {

 public static void main(String[] args)
 {
  System.out.println(Hello.X); //Hello interface is not loaded

 }

}
```

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 sub class constructor's super keyword will directly move to Object class constructor.(11-DEC)

```
package com.ravi.inheritance;

interface Alpha
{
 void m1();
}
interface Beta
{
 void m1();
```

```
}
class Implementer implements Alpha, Beta
{
 @Override
 public void m1()
 {
  System.out.println("MI is possible");
 }
}



public class MultipleInheritance
{
 public static void main(String[] args)
 {
  Implementer i = new Implementer();
  i.m1();

 }

}
```

12-12-2024

Extending interface :

One interface can extends another interface, it cannot implement because interface cannot provide implementation for the abstract method.

```
package com.ravi.exetnding_interface;

interface Alpha
{
 void m1();
}
interface Beta extends Alpha
{
 void m2();
}
class MyClass implements Beta
{

 @Override
 public void m1()
 {
  System.out.println("M1 method Overridden");
 }

 @Override
 public void m2()
 {
  System.out.println("M2 method Overridden");
 }
}
```

```
public class ExtendingInterfaceDemo {

 public static void main(String[] args)
 {
  MyClass m = new MyClass();
  m.m1(); m.m2();

 }

}
```

---

java 8 features :

---

intreface from JDK 1.8V [Java 8 = March 2014]

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?

To avoid this maintenance problem java software people introduced default method inside an interface.

What is default method :

We can write default method (method with body) inside an interface with default keyword from Java 8v.

This default method provides "default implementation" so the implementer class can override to provide specific implementation in the class.

Unlike abstract method, default method does not provide any kind of boundation to override this default method in the sub class.

By default the access modifier of default method is public.

We can’t write default method inside a class, we can write
only inside an interface.

4 files :

Vehicle.java(I)

```
package com.ravi.java_new_features;

public interface Vehicle
{
    void run();
    void horn();
```

```
    default void digitalMeter()  //java 8
    {
     System.out.println("Default Implementation");
     System.out.println("Digital Meter Facility is coming soon");
    }
}
```

Car.java

```
---------
package com.ravi.java_new_features;

public class Car implements Vehicle
{
 @Override
 public void run()
 {
  System.out.println("Car is running");
 }

 @Override
 public void horn()
 {
  System.out.println("Car has horn");
 }

  @Override
  public void digitalMeter()  //java 8
  {
   System.out.println("Digital Meter Facility is Available in the Car");
  }
}
```

Bike.java

```
----------
package com.ravi.java_new_features;

public class Bike implements Vehicle {

 @Override
 public void run()
 {
  System.out.println("Bike is running");
 }

 @Override
 public void horn()
 {
  System.out.println("Bike has horn");
 }
}

package com.ravi.java_new_features;

public class DefaultMethod
```

```
{
 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(); v.digitalMeter();

 }

}
```

Note :- abstract method is a common method which is used to provide easiness to the programmer so, by looking the abstract method we will get confirmation that this is common behavior for all the sub classes and it must be implemnted in all the sub classes.

Common method [Behavior] -> abstract method
Uncommon method [Behavior] -> default method

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.

```
package com.ravi.java_new_features;

interface A
{
 default void m1()
 {
  System.out.println("Default Method of interface A");
 }
}
class B
{
 public void m1()
 {
  System.out.println("Concrete Method of Class B");
 }
}

class C extends B implements A
{
}

public class AmbiguityIssue {

 public static void main(String[] args)
 {
  C c1 = new C();
  c1.m1();

 }

}
```

Multiple Inheritance by 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(Alpha.super.m1()).

```
package com.ravi.java_new_features;

interface Alpha
{
 default void m1()
 {
  System.out.println("m1 method of Alpha interface");
 }
}
interface Beta
{
 default void m1()
 {
  System.out.println("m1 method of Beta interface");
 }
}

class MI implements Alpha, Beta
{

 @Override
 public void m1() //Overriding is compulsory, otherwise
 {                  we will get compilation error
  Alpha.super.m1();
  Beta.super.m1();
  System.out.println("MI is possible");
 }
}
public class MultipleInheritance
{
 public static void main(String[] args)
 {
  MI m = new MI();
  m.m1();
 }

}
```

Note : Here both the interfaces are having same method name m1() so, overridng is compulsory in the implementer class otherwise we will get compilation error due to ambiguity issue.

13-12-2024

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 :

Calculate.java(I)

```
package com.ravi.static_method;

public interface Calculate
{
   static double getSquare(int num)  //JDK 1.8 [by default public]
   {

    return num*num;
   }

   static double getCube(int num)
   {
    return num*num*num;
   }
}
```

Main.java

```
package com.ravi.static_method;

public class Main {

 public static void main(String[] args)
 {
  System.out.println("Square is :"+Calculate.getSquare(12));
  System.out.println("Cube is :"+Calculate.getCube(12));

 }

}
```

Program that describe that static method of an interface is only available to interface only that means we can access the static method of an interface by using only one way i.e interface name.

```
interface Alpha
{
 static void m1()
 {
  System.out.println("Interface static method");
 }
}
class Beta implements Alpha
{
}
public class StaticMethodOfInterface
```

```
{
 public static void main(String[] args)
 {
  Alpha.m1();

  //Beta.m1();  [Invalid]

  Beta b1 = new Beta();
  //b1.m1(); //[Invalid]
 }
}
```

From the above program it is clear that interface static method
we can call through interface only.

Can we write a main method inside an interface ?

Yes, We can write main method inside an interafce from java 8 version and it will be executed becuase interface is internally an abstract class.

```
package com.ravi.execution;

public interface Printable
{
   public static void main(String[] args)
   {
   System.out.println("Main Method");
   }
}
```

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.

```
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
 }
}
```

Introdction to Functional Programming :

In OOP, We always concentrate on objects but in Function Programming which is introduced from JDK 1.8V, Here we will
concentrate on functions.

What is a Functional Interface ?

A Functional interface is an interface which contains exactly one abstract method.

It may contain ’n’ number of static and default methods but it must contain exactly one abstract method.

A functional interface can be annotated be @FunctionalInterface annotation.

Example :

```
@FunctionalInterface
public interface Printable
{

   void print();  //[SAM = Single Abstract Method]

   default void m1()
   {

   }

   static void m3()
   {

   }

}
```

Functional interface by using Anonymous inner class :

```
package com.ravi.interface_demo;

@FunctionalInterface
interface Payment
{
 void makePayment();
}
```

```
public class AnonymousWithFunctionalInterface {

 public static void main(String[] args)
 {
  Payment creditCard = new Payment()
  {
   @Override
   public void makePayment()
   {
    System.out.println("Making payment through Credit Card");
   }
  };

  Payment debitCard = new Payment()
  {
   @Override
   public void makePayment()
   {
    System.out.println("Making payment through Debit Card");
   }
  };

  creditCard.makePayment();  debitCard.makePayment();

 }

}
```

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.
It can be used with functional interface only.
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.

```
abstract class Drawable
{
 abstract void draw();
}
public class LambdaTarget
{
 public static void main(String[] args)
 {
```

```
  Drawable d = ()-> System.out.println("Drawing");
  d.draw();
 }
}
```

Note : The above program will generate compilation error, Lambda Target must be Functional interface.

Program on Lambda Expression :

```
package com.ravi.lambda;

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();
 }

}
```

```
package com.ravi.basic_concepts;

import java.util.Scanner;

@FunctionalInterface
interface Calculate
{
 double doSum(double x, double y);

}

public class LambdaDemo2 {

 public static void main(String[] args)
 {
  Calculate c1 =  (a, b)->  a + b;

  Scanner sc = new Scanner(System.in);
  System.out.println("Enter the value of a :");
  double p = sc.nextDouble();
  System.out.println("Enter the value of b :");
  double q = sc.nextDouble();
```

```
  System.out.println("Sum is :"+c1.doSum(p, q));
  sc.close();
 }

}
-------------------------------------------------------------
package com.ravi.lambda;

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"));

 }

}
-------------------------------------------------------------
package com.ravi.basic_concepts;

import java.util.Scanner;

@FunctionalInterface
interface Verifier
{
 boolean verify(Integer num);
}

public class LambdaDemo4
{
 public static void main(String[] args)
 {
  //Check whether a number is even or odd
  Scanner sc = new Scanner(System.in);
  System.out.print("Enter a Number :");
  int no = sc.nextInt();

        Verifier v1 =  num ->
        {
           return num % 2==0;
        };

        System.out.println("Is "+no+ " even Number :"+v1.verify(no));
        sc.close();
 }

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

```
/* If the input number is 0 or negative return -1
 * If the input number is even return square of the number
 * If the input number is even return cube of the number
 *  */

package com.ravi.basic_concepts;

import java.util.Scanner;

@FunctionalInterface
interface Calculator
{
 Double getSquareAndCube(Integer num);
}


public class LambdaDemo5 {

 public static void main(String[] args)
 {
  Calculator calc = num ->
  {
   if(num<=0)
   {
    return -1D;
   }
   else if(num % 2== 0)
   {
    Double y = Double.valueOf(num*num);
    return y;
   }
   else
   {
    Double z = Double.valueOf(num*num*num);
    return z;
   }
  };

  Scanner sc = new Scanner(System.in);
  System.out.print("Enter a number :");
  int no = sc.nextInt();

  System.out.println(calc.getSquareAndCube(no));

  sc.close();
 }

}
```

==========

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 (reference classe) to this type parameter.

We cannot pass any primitive type to this type parameter.

```
package com.ravi.basic_concepts;

import java.util.stream.DoubleStream;

class Accept<T>
{
 private T data;

 public Accept(T data) //Product data =
 {
  super();
  this.data = data;
 }

 public T getData()
 {
  return data;
 }
}

public class TypeParameter
{
 public static void main(String[] args)
 {
    Accept<Integer> acceptInt = new Accept<Integer>(12);
       System.out.println("Integer type is :"+acceptInt.getData());

       Accept<Double> acceptDouble = new Accept<Double>(90.78);
       System.out.println("Double type is :"+acceptDouble.getData());


       Accept<Boolean> acceptBoolean = new Accept<Boolean>(false);
       System.out.println("Boolean type is :"+acceptBoolean.getData());


       Accept<Product> acceptProduct = new Accept<Product>(new Product(111));
       System.out.println("Product type is :"+acceptProduct.getData());




 }

}


class Product
{
 private int productId;

 public Product(int productId)
```

```
 {
  super();
  this.productId = productId;
 }

 @Override
 public String toString()
 {
  return "Product [productId=" + productId + "]";
 }

}
```

17-12-2024

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.

Programs on Predicate :

```
-----------------------
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo1
{
 public static void main(String[] args)
 {
  //WAP to verify whether a number is even or odd

  Predicate<Integer> p1 = num -> num%2==0;

  Scanner sc = new Scanner(System.in);
  System.out.print("Enter a Number :");
  int num = sc.nextInt();

  System.out.println("Is "+num+" even number ?"+p1.test(num));
  sc.close();
 }

}
-----------------------------------------------------------
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo2
{
 public static void main(String[] args)
 {
    //Given name starts with character 'A' or not

    Predicate<String> p2 = str -> str.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 or not ?"+p2.test(name));
    sc.close();

 }

}
-----------------------------------------------------------
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo3
{
```

```
 public static void main(String[] args)
 {
  //Based on age person is eligible for voting or not

  Predicate<Integer> p3 = age -> age >=18;

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

  boolean isEligible = p3.test(age);

  if(isEligible)
  {
   System.out.println("You are eligible for Voting");
  }
  else
  {
   System.out.println("You are not eligible for Voting");
  }
  sc.close();
 }

}
```

```
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo4 {

 public static void main(String[] args)
 {
  //Verify my name is Ravi or not

  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>

Consumer<T> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It contains an abstract method accept() which takes T type parameter and returns nothing (void). It is used to accept the parameter value or consume the value.

```
@FunctionalInterface
public interface Consumer<T>
{
   void accept(T x);
}
```

```
package com.ravi.functional_interface;

import java.util.function.Consumer;

public class ConsumerDemo1 {

 public static void main(String[] args)
 {
  Consumer<Integer> c1 = num -> System.out.println("Integer type "+num);
     c1.accept(12);

     Consumer<Boolean> c2 = bool -> System.out.println("Boolean Type :"+bool);
     c2.accept(false);

     Consumer<Customer> c3 = cust -> System.out.println("Customer Type :"+cust);
     c3.accept(new Customer(111));

 }

}

class Customer
{
 private int customerId;

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

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

}
```

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.

```
@FunctionalInterface
public interface Function<T,R>
{
   public abstract R apply(T x);
}
```

```
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionDemo1 {

 public static void main(String[] args)
 {
  //Finding the cube of a number
  Function<Integer,Integer> fn1 = num -> num*num*num;

  Scanner sc = new Scanner(System.in);
  System.out.print("Enter a Number :");
  int num = sc.nextInt();

  System.out.println("Cube of "+num+" is :"+fn1.apply(num));
  sc.close();

 }

}
```

```
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionDemo2 {

 public static void main(String[] args)
 {
  //Finding the length of given city

  Function<String,Integer> fn2 = str -> str.length();

  Scanner sc = new Scanner(System.in);
  System.out.print("Enter your City Name :");
  String city = sc.next();

  System.out.println("Langth of "+city+" is :"+fn2.apply(city));
  sc.close();
```

```
 }

}
```

```
package com.ravi.functional_interface;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionDemo3 {
 public static void main(String[] args)
 {
  // Verify whether my name starts with character A or not

  Function<String, Boolean> fn3 = str -> str.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 "+fn3.apply(name));
       sc.close();
 }

}
```

18-12-2024

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
public interface Supplier<T>
{
   T get();
}
```

//Programs on Supplier :

```
package com.ravi.supplier;

import java.util.function.Supplier;

public class SupplierDemo1
{

 public static void main(String[] args)
 {
  Supplier<String>  s1 = () -> 100 + 200+"NIT"+ 80 + 80 ;
  System.out.println(s1.get());
```

```
 }

}
-----------------------------------------------------------
package com.ravi.supplier;

import java.util.function.Supplier;

class Employee
{
 private Integer employeeId;
 private String employeeName;
 private Double employeeSalary;

 public Employee(Integer employeeId, String employeeName, Double employeeSalary) {
  super();
  this.employeeId = employeeId;
  this.employeeName = employeeName;
  this.employeeSalary = employeeSalary;
 }

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



}

public class SupplierDemo2
{

 public static void main(String[] args)
 {
       Supplier<Employee>  s2 = () -> new Employee(1, "Scott", 34890.90);

       Employee obj = s2.get();

       System.out.println(obj);
 }

}
-----------------------------------------------------------
package com.ravi.supplier;

import java.util.Scanner;
import java.util.function.Supplier;

class Product
{
 private Integer productId;
```

```
 private String productName;
 private Double productPrice;

 public Product(Integer productId, String productName, Double productPrice) {
  super();
  this.productId = productId;
  this.productName = productName;
  this.productPrice = productPrice;
 }

 @Override
 public String toString() {
  return "Product [productId=" + productId + ", productName=" + productName + ", productPrice=" +
productPrice
    + "]";
 }
}

public class SupplierDemo3
{
   public static void main(String[] args)
   {
        Supplier<Product> s3 =  ()->
        {
         Scanner sc = new Scanner(System.in);
         System.out.println("Enter Product Id :");
         int id = sc.nextInt();
         System.out.println("Enter Product Name :");
         String name = sc.nextLine();
         name = sc.nextLine();
         System.out.println("Enter Product Price :");
         double price = sc.nextDouble();

         return new Product(id, name, price);
        };

        Product obj = s3.get();
        System.out.println(obj);


   }
}
```

Creating our own Functional interface with various Parameter

We can create our own userdefined functional interaface
with various parameters as shown below :

```
package com.ravi.custom_fun_interface;

@FunctionalInterface
interface TriFunction<T,U,V,R>
{
 public abstract R myApply(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(fn1.myApply(12, 24, 44));

 }

}
```

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.

```
@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.

```
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);
 }
}
```

```
-----------------------------------------------------------
import java.util.function.BiPredicate;

public class Lambda12
{
  public static void main(String[] args)
  {
    // 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.

```
@FunctionalInterface
public interface BiConsumer<T, U>
{
    void accept(T t, U u);
}
-----------------------------------------------------------
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.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,R>
{
  public abstract T apply(T x);
}
```

```
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.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,U,R>
{
  public abstract T apply(T x, T y);
}
```

```
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));
 }
}
```

19-12-2024

Can an interface extend a class ?

An interface can’t extend a class, It can extend only interface.

Every public method of Object class is implicitly re-declared inside every interface as an abstract method to support upcasting if interface does not extend any super interface.

We can’t override any public method of object class as a
default method inide interface.

```
package com.ravi.interface_member;
```

```
interface Drawable
{

}
public class InterfaceMemberDemo1 {

 public static void main(String[] args)
 {
  Drawable d = null;
  d.hashCode();
  d.toString();
  d.equals(null);

 }

}
```

```
package com.ravi.interface_member;

interface Printable
{

}
class Print implements Printable
{

 @Override
 public String toString() {
  return "Print []";
 }

}
public class InterfaceMemberDemo2
{
 public static void main(String[] args)
 {
  Printable p = new Print();
  System.out.println(p.hashCode());
  System.out.println(p.toString());
 }

}
```

```
package com.ravi.interface_member;

@FunctionalInterface
abstract interface Moveable
{
 void move();
 public String toString();
 public int hashCode();
    public boolean equals(Object obj);


}
```

```
public class InterfaceMemberDemo3 {

 public static void main(String[] args) {
  // TODO Auto-generated method stub

 }

}
```

```
package com.ravi.interface_member;

interface Alpha
{

 default String toString() //error
 {
  return null;
 }

 default int hashCode() //error
 {
  return 100;
 }


}

public class InterfaceMemberDemo4 {

 public static void main(String[] args) {
  // TODO Auto-generated method stub

 }

}
```

Note : From the above program, It is clear that we can't override Object class public method as a default method inside interface.

Interface from JAVA 9V

We can write private static and private non static (not public)
methods inside an interface from java 9 version.

The main purpose of providing these two methods inside an
interface are as follows :

1) Code Reusability

If two or more than two default methods want to share a common code (Helper Method code) then we can write these common code in private methods so it will enhance code reusability.

2) Hide the Logic from Outer World

By writing these code in private static and private non static methods the actual logic is not visible to the outer world so, It is way to make our interface as a fully abstract class so 100% abstraction is possible.

Note : By default interface is not Fully abstract but we can make it full abstract from java 9V by writing the logic inside private method.

Note : from default method we can call private static as well as private non static methods but from public static method of interface we can call only private static method.

```
package com.ravi.interface_member;

interface Acceptable
{
 int MAX_VALUE = 500;  //JDK 1.0

 void m1(); //JDK 1.0

 default void m2() //JDK 1.8
 {
  m4();
  m5();
 }

 static void m3()  //JDK 1.8
 {
  m4();

 }

 private static void m4()  //Java 9
 {
  System.out.println("Private static method");
 }

 private void m5()  //Java 9
 {
  System.out.println("Private non static method");
 }
}
class Accept implements Acceptable
{
 @Override
 public void m1()
 {
  System.out.println("M1 overridden Method");

 }

}

public class InterfaceMemberFromJava9
{
 public static void main(String[] args)
 {
     Acceptable a = new Accept();
```

```
      a.m1();
      a.m2();
      Acceptable.m3();


 }

}
```

The following Program explains how to use Helper method (private Method) to validate different data

```
package com.ravi.method_overloading;

import java.util.Scanner;

class Payment
{
// Payment using cash
    public void makePayment(double amount)
    {
        if (validateAmount(amount))
        {
            System.out.println("Processing payment via Cash...");
            System.out.println("Amount Paid RS :" + amount);
            System.out.println("Payment Successful!");
        }
    }

    //Payment through creditCard
    public void makePayment(String cardHolderName, String creditCardNumber,double amount)
    {
        if (validateCardNumber(creditCardNumber) && validateAmount(amount))
        {
            System.out.println("Processing payment via Credit Card...");
            System.out.println("Card Holder: " + cardHolderName);
            System.out.println("Card Number: " + maskCardNumber(creditCardNumber));
            System.out.println("Amount Paid RS :" + amount);
            System.out.println("Payment Successful!");
        }
    }

    //Payment through debitCard
    public void makePayment(String debitCardNumber, double amount)
    {
        if (validateCardNumber(debitCardNumber) && validateAmount(amount))
        {
            System.out.println("Processing payment via Debit Card...");
            System.out.println("Card Number: " + maskCardNumber(debitCardNumber));
            System.out.println("Amount Paid RS :" + amount);
            System.out.println("Payment Successful!");
        }
    }

    // Helper method to validate amount
```

```
    private boolean validateAmount(double amount)
    {
        if (amount <= 0)
        {
            System.err.println("Error: Amount must be greater than zero.");
            return false;
        }
        return true;
    }

    // Helper method to validate credit OR debit card
    private boolean validateCardNumber(String cardNumber)
    {
        if (cardNumber.length() != 16 )
        {
            System.err.println("Error: Invalid card number. It must be 16 digits.");
            return false;
        }
        return true;
    }

    //Helper Method
    private String maskCardNumber(String cardNumber)
    {
        return "****-****-****-" + cardNumber.substring(12);
    }


}

public class PaymentProcess
{
    public static void main(String[] args)
    {
        System.out.println("Payment Menu");
        System.out.println("Please select any one Payment Method from the Menu :");
        System.out.println("\t\t 1) Payment by using Cash ");
        System.out.println("\t\t 2) Payment by using Credit Card ");
        System.out.println("\t\t 3) Payment by using Debit Card ");

        Payment payment = new Payment();
        Scanner sc = new Scanner(System.in);

        System.out.println("Please enter your Payment choice [1/2/3]");
        int choice = sc.nextInt();

        switch(choice)
        {
        case 1:
        System.out.println("Enter the amount you want to pay through cash :");
        double amount = sc.nextDouble();
        payment.makePayment(amount);
        break;

        case 2:
```

```
        System.out.println("Enter your name :");
        String name = sc.nextLine();
        name = sc.nextLine();
        System.out.println("Enter your 16 digit Credit Card Number :");
        String creditCard = sc.nextLine();

        System.out.println("Enter your Payment Amount :");
        amount = sc.nextDouble();
        payment.makePayment(name, creditCard, amount);
        break;

        case 3 :
        System.out.println("Enter your 16 digit Debit Card Number :");
        String debitCard = sc.nextLine();
        debitCard = sc.nextLine();
        System.out.println("Enter your Payment Amount :");
        amount = sc.nextDouble();
        payment.makePayment(debitCard, amount);
        break;
        }

        sc.close();


    }
}
```

What is a Marker interface ?

If an interface does not contain any field or method, Basically It is an empty interface then it is called Marker interface OR Tag interface.

Example :

```
interface Callable  //Marker interface
{

}
```

In java we have following marker interfaces are available :

a) java.io.Serializable
b) java.lang.Cloneable
c) java.util.RandomAccess

**The main purpose of marker interface to provide additional information to the JVM reagarding the Object like Object is Serializable, Cloneable OR Randomly Accessible.

****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 (no instance variable).

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 Completed.....

1) Exception Handling
2) Multithreading
3) Collections Framework (22 Session)

a) enum b) inner classes c) Object class and its method
d) file Handling and Input and Output
==========
20-12-2024

Exception Handling :

An exception is an abnormal situation OR un-expected situation in a normal execution flow.

Due to an exception, the execution of the program will be disturbed first and then terminated permanently.

Exception always encounter at runtime only.

Exception encounter due to the following reasons :

1) The Wrong input given by the user.
2) Due to dependency, When one part of the program is dependent to another part to complete the task then there might be a chance of getting an exception.

Different Crieteria of Exception :

The following are the different criteria for exception :

1) java.lang.ArithmeticException

Whenever we divide a number by zero(an int value) then we will get a RuntimeException i.e java.lang.ArithmeticException.

```
int x = 10;
int y = 0;
int z = x/y;  //java.lang.ArithmeticException.
System.out.println(z);
```

2) java.lang.ArrayIndexOutOfBoundsException
If we try to access the index of the array where element is not available then we will get java.lang.ArrayIndexOutOfBoundsException

```
int []arr = {10,20,30};
System.out.println(arr[3]); //No value available for 3rd index
```

3) java.lang.StringIndexOutOfBoundsException
While retrieving the character from the String, if we pass any negative index then we will get java.lang.StringIndexOutOfBoundsException

```
String str = "Hyderabad";
System.out.println(str.substring(-2,7));
```

Note : IndexOutOfBoundsException is the super for both the folowing classes
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException

4) java.lang.NegativeSizeArrayException
While defining an array, the size of the Array must be positive integer value otherwise we will get java.lang.NegativeArraySizeException

5) java.lang.NullPointerException
If any refrence variable is pointing to null literal then we can’t invoke any non static method on this reference variable which is pointing to null otherwise we will get NullPointerException.

```
String str = null;
System.out.println(str.length()); //NullPointerException
```

OR
--

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter your Name :");
String name = sc.nextLine();

System.out.println(name.length());  //No Exception
System.out.println(name.toUpperCase()); //No Exception
```

Here we will not get any exception even we provide null

6) java.lang.NumberFormatException

If we try to convert a String value into primitive OR Wrapper type but if the number is not available in numeric format then we will get java.lang.NumberFormatException

```
String str = "NIT";
//int p = Integer.parseInt(str);
Integer w = Integer.valueOf(str);
```

7) java.util.InputMismatchException
While reading the data through Scanner class if the input

is not is a proper format then we will get java.util.InputMismatchException

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter your Roll :");
int roll = sc.nextInt();

System.out.println(roll);
```

Exception Hierarchy :

Diagram (20-DEC-24)

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.

23-12-2024

Exception format :

In java, If we want to print any exception object by using print() statement then the java software people has provided
the following format :

Fully Qualified Name (Package Name + class Name) : errorMessage

```
package com.ravi.exception;

public class ExceptionFormat {

 public static void main(String[] args)
 {
  ArithmeticException e1 = new ArithmeticException("Divided the number by zero");
  System.out.println(e1.toString());

  //java.lang.ArithmeticException : Divided the number by zero

 }

}
```

WAP to show that java.lang.Exception is the super class for all the exceptions (Checked + Unchecked)

```
package com.ravi.exception;

import java.io.IOException;

public class ExceptionSuper
{
 public static void main(String[] args)
 {
  Exception e1 = new ArithmeticException();
  System.out.println(e1);

  e1 = new ArrayIndexOutOfBoundsException();
```

```
  System.out.println(e1);

  e1 = new NullPointerException();
  System.out.println(e1);

  e1 = new NumberFormatException();
  System.out.println(e1);

  e1 = new IOException();
  System.out.println(e1);
 }

}
```

WAP that describes that whenever an exception is encounter in the program then program will be terminated in the middle.

```
package com.ravi.exception;

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();
 }
}
```

Note : In the above program, If we enter the value of y as 0 then our program will be terminated abnormally, JVM has
default exception handler, which will terminate the program in the middle (abnormal termination) and provide the execption message with line number.

In order to work with exception, java software people has provided the following keywords :

1) try block
2) catch block
3) finally block [Java 7 try with resourses]
4) throw
5) throws

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 encounter 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.

```
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.

24-12-2024

```
public class Main
{
 public static void main(String[] args)
 {
  try
  {
    //System.out.println(10/0);

              //OR

    throw new ArithmeticException("I am dividing no by zero");
  }
  catch(Exception e)
  {
   System.err.println("Inside Catch Block");
   System.err.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.

The main purpose of Exception handling to provide user-friendly message to our end user as shown in the program.

```
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, Please don’t put zero here");
    }

    System.out.println("Thank you client, Have a nice Day!!!");
    sc.close();
 }

}
```

Exception handlinag = No Abnormal Termination + User-friendly message on wrong input given by the client.

==========

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 Object 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();
   System.out.println("Your Roll is :"+roll);

  }
  catch(InputMismatchException e)
  {
   e.printStackTrace();
  }
  sc.close();
  System.out.println("Main ended");
 }
}
```

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

```
   throw new Error();
  }
  catch (Exception e)
  {
   System.err.println("Catch");
   System.err.println(e);
  }
 }
}
```

Note : Here the catch block is unable to handle the exception
because there is no relation between Error and Exception, both are sub class of Throwable class.

In the corresponding catch block, If we write Error OR Throwable
then we catch block would be executed.

26-12-2024

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.

```
package com.ravi.basic;

public class InfinityFloatingPoint
{
 public static void main(String[] args)
 {
    System.out.println("Main Started");
    System.out.println(10/0.0);
    System.out.println(-10/0.0);
    System.out.println(0/0.0);
    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.

```
package com.ravi.basic;
public class MultipleTryCatch
{
 public static void main(String[] args)
 {
   System.out.println("Main method started!!!!");

   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;
```

```
public class MultyCatch
{
 public static void main(String[] args)
 {
  System.out.println("Main Started...");
  try
  {
   int c = 10/2;
   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...");
 }
}
-----------------------------------------------------------------
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

     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!!");
 }

}
```

27-12-2024

---

finally block [100% Guaranteed 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 :- In the above program finally block will be executed, even we have an exception in the try block but here only the resourses will be handled but not the exception.

```
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 resourses :

To avoid all the limitation of finally block, Java software people introduced a separate concept i.e try with resources from java 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 DatabaseResource class and FileResourse class.

3 files :

DatabaseResourse.java

```
package com.ravi.try_with_resourses;

public class DatabaseResourse implements AutoCloseable
{
 @Override
 public void close() throws Exception
 {
  System.out.println("Database Resourse Closed!!!");

 }

}
```

FileResourse.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!!!");

 }

}
```

Main.java

```
package com.ravi.try_with_resourses;

public class Main
{
 public static void main(String[] args) throws Exception
 {
  DatabaseResourse dr = new DatabaseResourse();
  FileResourse fr = new FileResourse();

  try(dr;fr)
  {
    System.out.println(10/0);
  }
  catch(ArithmeticException e)
  {
   System.err.println("Divide by zero problem");
  }
  System.out.println("Main method Completed!!");
 }

}
```

```
//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 TryWithResourses {

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

```
  try(sc)
  {
   System.out.println("Enter your Salary :");
   double sal = sc.nextDouble();
   System.out.println("Salary is :"+sal);
  }
  catch(InputMismatchException e)
  {
   System.err.println("Input is Inavlid");
  }


 }

}
```

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.

28-12-2024

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");
      }
 }
}
```

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
   {
    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.

```
package com.ravi.advanced;
public class ReturnExample
{
    public static void main(String[] args)
    {
       System.out.println(methodReturningValue());
    }


 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");
```

```
    }
}
```

```
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!!!");
 }

}
```

30-12-2024

**Difference between Checked Exception and Unchecked Exception :

Checked Exception :

A checked exception is a common exception that must be throws 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 throw by throws keyword 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

RuntimeException is also Unchecked Exception.

All the Errors comes under 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

---
*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)

When to provide try-catch or declare the method as throws :-
---
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 (not recommended) or caller method.

Note :- It is always better to use try catch so we can provide appropriate user defined messages to our client.

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 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.[30-DEC]

----------Exception Propagation program :

```
package com.ravi.custom_exception;

public class ExceptionPropagationWithUnchecked {

 public static void main(String[] args)
 {
    System.out.println("Main Method started..");
   try
   {
    m1();
   }
   catch(ArithmeticException e)
   {
    System.out.println("Handled in main");
   }
    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()
 {
  throw new ArithmeticException();
 }
}
```

```
package com.ravi.exception;

public class ExceptionPropagationWithChecked {

 public static void main(String[] args)
 {
    System.out.println("Main Method started..");
    try
    {
     m1();
    }
    catch(ClassNotFoundException e)
    {
     System.out.println("Handled by main method");
    }
    System.out.println("Main Method ended..");
 }
 public static void m1() throws ClassNotFoundException
 {
    System.out.println("M1 Method started..");
    m2();
    System.out.println("M1 Method ended..");
 }
 public static void m2() throws ClassNotFoundException
```

```
 {
  System.out.println("M2 method Body");

  Class.forName("Sample");
 }
}
```

** 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.

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.

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.

WAP to develop user-defined 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)
 {
  Scanner sc = new Scanner(System.in);
  try(sc)
  {
   System.out.print("Enter your Age :");
   int age = sc.nextInt();
   validateAge(age);
  }
  catch (InvalidAgeException e)
  {
   System.err.println("You are not allowed for Movie "+e);
  }
```

```
 }

 public static void validateAge(int age) throws InvalidAgeException
 {
  if(age<18)
  {
   throw new InvalidAgeException("Age is Invalid");

  }
  else
  {
   System.out.println("You are allowed for Movie");
  }


 }

}
```

01-01-2025

WAP to develop user-defined un-checked Exception :

```
package com.ravi.exception;

import java.util.Scanner;

@SuppressWarnings("serial")
class GreaterMarksException extends RuntimeException
{
 public GreaterMarksException()
 {
 }

 public GreaterMarksException(String errorMessage)
 {
    super(errorMessage);
 }
}


public class UserDefinedUnchecked {

 public static void main(String[] args)
 {
  Scanner sc = new Scanner(System.in);
  try(sc)
  {
   System.out.print("Enter your Marks :");
   int marks = sc.nextInt();
   validateMarks(marks);
  }
  catch(GreaterMarksException e)
  {
   System.out.println(e);
```

```
  }

 }

 public static void validateMarks(int marks)
 {
  if(marks > 100)
  {
   throw new GreaterMarksException("Invalid Marks");
  }
  else
  {
   System.out.println("Your Marks is :"+marks);
  }
 }
}
```

Some Basic Rules related to 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.FileNotFoundException;
import java.io.IOException;

import com.ravi.basic.ThrowException;

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.

```
package com.ravi.method_related_rule;

import java.io.FileNotFoundException;
import java.io.IOException;

class Super
{
 public void show()
 {
  System.out.println("Super class method not throwing checked Exception");
 }
}
class Sub extends Super
{
 @Override
 public void show() throws ClassNotFoundException //error
 {
  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.

```
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  //error
 {
  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.

```
public class Main
{

 {
  try
  {
```

```
   throw new ArithmeticException();
  }
  catch (ArithmeticException e)
  {
    System.out.println("Normal Termination");
  }
 }

 public static void main(String[] args)
 {
   new Main();
   System.out.println("Main");
 }

}
```

```
public class ArrayStoreException {

 public static void main(String[] args)
 {
  Object []obj = new String[3]; //valid with Array
  obj[0] = "Ravi";
  obj[1] = "hyd";
  obj[2] =  67.90; //java.lang.ArrayStoreException

  System.out.println(Arrays.toString(obj));

 }

}
```

---

public boolean equals(Object obj) :

There is predefined non static method called equals(Object obj) which is available in java.lang.Object class.

It is used to compare two objects based on the memory reference or memory address so we can say the object class equals() method behavior is similar to == operator because internally, It uses == operator only as shown in the program.

```
package com.ravi.equals;

class Employee extends Object
{
 private int employeeId;
 private String employeeName;

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

```
}

public class EqualsDemo
{
 public static void main(String[] args)
 {
    Employee e1 = new Employee(111, "Scott"); //1000x
    Employee e2 = new Employee(111, "Scott"); //2000x

    System.out.println(e1==e2); //false
    System.out.println(e1.equals(e2)); //false [== operator]

 }

}
```

In the above program equals(Object obj) methdo will return false because internally object class equals(Object obj) method uses == operator.

We can override this equals(Object obj) method in the Employee class for content comparison. Eclipse IDE provides auto generate facility to override hashCode() and equals(Object obj).

```
package com.ravi.equals;

import java.util.Objects;

class Employee extends Object
{
 private int employeeId;
 private String employeeName;

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

 //Auto generated by Eclipse IDE
 @Override
 public int hashCode() {
  return Objects.hash(employeeId, employeeName);
 }

       //Overriding equals(Object obj) method for content
 comparison
 @Override
 public boolean equals(Object obj) {
  if (this == obj)
   return true;
  if (obj == null)
   return false;
  if (getClass() != obj.getClass())
   return false;
  Employee other = (Employee) obj;
```

```
  return employeeId == other.employeeId && Objects.equals(employeeName, other.employeeName);
 }



}

public class EqualsDemo
{
 public static void main(String[] args)
 {
    Employee e1 = new Employee(111, "Scott");
    Employee e2 = new Employee(111, "Scott");


    System.out.println(e1.equals(e2));  //true

 }

}
```

Note : In the above program we will get the output as true because the overridden equals(Object obj) method will compare the content of the both the objects. It is auto generated method i.e equals() and hashCode() method

There is contract between equals() and hashCode() method is, we should always overridde both the methods together.

Record class in java [java 17 features]

---

public abstract class Record extends Object.

record Student(){} //final 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 (final 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.

3 files :

ProductClass.java(C)

```
package com.ravi.record;

import java.util.Objects;

public class ProductClass
{
 private Integer productId;
 private String productName;

 public ProductClass(Integer productId, String productName)
 {
  super();
  this.productId = productId;
  this.productName = productName;
 }

 public Integer getProductId() {
  return productId;
 }

 public void setProductId(Integer productId) {
  this.productId = productId;
 }

 public String getProductName() {
  return productName;
 }

 public void setProductName(String productName) {
  this.productName = productName;
 }

 @Override
 public String toString() {
  return "ProductClass [productId=" + productId + ", productName=" + productName + "]";
 }

 @Override
 public int hashCode() {
  return Objects.hash(productId, productName);
```

```
 }

 @Override
 public boolean equals(Object obj)
 {
  if (this == obj)
   return true;
  if (obj == null)
   return false;
  if (getClass() != obj.getClass())
   return false;
  ProductClass other = (ProductClass) obj;
  return Objects.equals(productId, other.productId) && Objects.equals(productName,
other.productName);
 }
}
```

ProductRecord.java(R)

```
----------------------
package com.ravi.record;


public record ProductRecord(Integer productId, String productName)
{

 //to validate Outer world data (Compact constructor)
 public ProductRecord
 {
    if(productId <= 0)
    {
     System.err.println("Invalid Id");
    }

 }

}
```

MainClass.java

```
---------------
package com.ravi.record;

public class MainClass {

 public static void main(String[] args)
 {
  ProductClass p1 = new ProductClass(111, "Camera");
  System.out.println(p1);
  ProductClass p2 = new ProductClass(111, "Camera");
  System.out.println(p1.equals(p2));
  System.out.println(p1.getProductName());



  System.out.println(".....................................");
  ProductRecord r1 = new ProductRecord(999, "Laptop");
```

```
        System.out.println(r1);
        ProductRecord r2 = new ProductRecord(999, "Laptop");
        System.out.println(r1.equals(r2));
        System.out.println(r1.productName());
 }

}
```

==========

03-01-2025

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
b) Thread based Multitasking

[Diagram : 03-JAN]

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.

```
  public native static Thread currentThread()
```

```
Thread t = Thread.currentThread(); //static Factory Method
```

Thread class has provided predefined method getName() to get the name of the Thread.

```
   public String getName();
```

Program to show main is a thread without method chaning :

```
public class Main
{
 public static void main(String[] args)
 {
      Thread t1 = Thread.currentThread();
   System.out.println("Current thread Name is :"+t1.getName());
 }

}
```

Program to show main is a thread with method chaning :

```
package com.ravi.mt;

public class MainThread
{
 public static void main(String[] args)
 {
  String name = Thread.currentThread().getName();
  System.out.println("Current Thread Name is :"+name);

 }

}
```

Note : The main pupose of main thread to execute the entire main method.

How to create user-defined thread ?

In order to create user-defined thread we can use 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 Thread by using java.lang package :

In order to create user-defined thread by using java.lang package we can use the following two techniques :

a) By extending java.lang.Thread class
b) By implementing java.lang.Runnable interface (Functional interface)

Note : java.lang.Thread class and java.lang.Runnable interface
both are available in java.lang package from JDK 1.0V.

04-01-2025

Creating a user thread by extends Thread class approach :
---
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.

```
package com.ravi.mi;

class Test extends Thread
{
 @Override
 public void run()
 {

  System.out.println("Child Thread is running!!!");
 }
}

public class UserThread
{
 public static void main(String[] args)
 {
  System.out.println("Main thread started!!!");

  Test t1 = new Test();
  t1.start();

  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. [04-JAN-25]

In entire Multithreading start() is the only method which is responsible to create a new thread.

---
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

```
package com.ravi.is_alive;

class Demo extends Thread
{
 @Override
 public void run()
 {
  System.out.println("Child Thread is running in a separate Stack");
 }
}

public class IsAlive {

 public static void main(String[] args)
 {
  Demo d1 = new Demo();
  System.out.println("Is child thread started before start() :"+d1.isAlive());

  d1.start();
  System.out.println("Is child thread started after start():"+d1.isAlive());

  d1.start();  //java.lang.IllegalThreadStateException

 }
}
```

```
package com.ravi.basic;

class Stuff extends Thread
{
 @Override
 public void run()
 {
  String name = Thread.currentThread().getName();
  System.out.println("Child Thread is Running, name is :"+name);
 }
}
public class ExceptionDemo
{
 public static void main(String[] args)
 {
```

```
  String name = Thread.currentThread().getName();
     System.out.println(name+" 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

---

05-01-2025

WAP to show that when we work with multiple threads then processor will frequently move from one thread to another thread.

```
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 value is :"+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 value is :"+i+ " by "+name+ " thread");
  }
```

```
  int x = 1;
  do
  {
   System.out.println("India by :"+name);
   x++;
  }
  while(x<=10);


 }
}
```

In the above program, Processor is frequently moving from main thread to child 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("Running Thread name is :"+name);
 }
}
public class ThreadName
{
 public static void main(String[] args) throws InterruptedException
 {
  DoStuff t1 = new DoStuff();
  DoStuff t2 = new DoStuff();



  t1.start();
  t2.start();



 System.out.println(Thread.currentThread().getName()+" thread is running.....");
```

```
 }
}
```

We are not providing the user-defined names so by default the name of thread would be Thread-0, Thread-1.

```
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");

    Demo d1 = new Demo();
    Demo d2 = new Demo();

    d1.setName("Child1");
    d2.setName("Child2");

    d1.start();
    d2.start();

    String name = Thread.currentThread().getName();
    System.out.println(name + " Thread is running Here..");


 }
}
```

Note : Here we are providing the user-defined name i.e child1 and child2 for both the user-defined thread.

```
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
{
 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
{
   @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.out.println("Thread is interrupted "+e);
  }
  }
   }

}
public class SleepDemo
{
 public static void main(String[] args)
 {
    Sleep s1 = new Sleep();
    s1.start();

 }
}
```

Note : Here child thread is not interrupted, so catch block will not be executed.

```
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   55
   try
   {
    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();
 }
}
```

Assignment :

Thread.sleep(long mills, int nanos);

06-01-2025

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 state 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.

Anonymous inner class by using Thread class :

Case 1:

Creating Anonymous inner class object using Thread class with ref.

```
package com.ravi.basic;

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(name+" thread is running Here.");
   }
  };
  t1.start();
  String name = Thread.currentThread().getName();
  System.out.println("Currently Executing Thread name is :"+name);


 }

}
```

Case 2 :

Creating Anonymous inner class object using Thread class without ref.

```
package com.ravi.basic;

public class AnonymousThreadWithoutReference {

 public static void main(String[] args)
 {
  new Thread()
  {
   @Override
   public void run()
   {
    String name = Thread.currentThread().getName();
    System.out.println(name+" thread is running Here.");
   }
  }.start();

 }

}
```

---07-01-2025

---

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.

```
package com.ravi.basic;

class Join extends Thread
{
 @Override
 public void run()
 {
  String name = Thread.currentThread().getName();
  for(int i=1; i<=5; i++)
  {
   System.out.println("i value is :"+i+" by "+name);
   try
   {
    Thread.sleep(1000);
   }
   catch(InterruptedException e)
   {
    e.printStackTrace();
   }
  }
  System.out.println(name+ "Thread is dead now");
 }
}

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.setName("J1");
  j2.setName("J2");
  j3.setName("J3");


  j1.start();
  j1.join();
  System.out.println("Main Thread wake up");
  j2.start();
  j3.start();

  System.out.println("Main Thread Completed");

 }

}
```

```
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
{
 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");
 }
}
```

```
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();

       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");
 }
}
```

Note : Here main thread is waiting for main thread to
complete so, It is deadlock situation.

Assignment :

join(long millis)
join(long millis, long nanos)

Assigning target by Runnable interface :[Loose Coupling]

By using Runnable interface we can assign different targets to our thread but thread will be created by using start() method
of Thread class.

```
package com.ravi.basic;

class Demo1 implements Runnable
{
 @Override
 public void run()
 {
  System.out.println("Thread is Running");
 }
}

public class RunnableDemo
```

```
{
   public static void main(String [] args)
   {
    System.out.println("Main");
   Demo1 d = new Demo1();
   Thread t1 = new Thread(d);
   t1.start();
   }
}
```

Assigning different Target by multiple thread :

```
package com.ravi.basic;

class Tatkal implements Runnable
{
 @Override
 public void run()
 {
  String name = Thread.currentThread().getName();
  System.out.println(name+" is booking ticket under Tatkal Scheme");
 }
}
class PremimumTatkal implements Runnable
{
 @Override
 public void run()
 {
  String name = Thread.currentThread().getName();
  System.out.println(name+" is booking ticket under PremimumTatkal Scheme");
 }
}

public class RunnableDemo1
{
 public static void main(String[] args) throws InterruptedException
 {
  Thread t2 = new Thread(new PremimumTatkal(),"Smith");
  t2.start();

  Thread t1 = new Thread(new Tatkal(),"Scott");
  t1.start();

  Thread t3 = new Thread(new Tatkal(), "Martin");
  t3.start();


 }

}
```

08-01-2025

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);

Anonymous inner class by using Runnable interface :

Case 1 :

Implementing run() method by using Anonymous inner class :

```
package com.ravi.mi;

public class AnonymousRunnable {

 public static void main(String[] args)
 {
  Runnable r1 = new Runnable()
  {
   @Override
   public void run()
   {
    String name = Thread.currentThread().getName();
    System.out.println(name);
   }
  };
  Thread t1 = new Thread(r1);
  t1.start();

 }

}
```

Case 2 :

By using Lambda :

```
package com.ravi.runnable_ex;

public class RunnableLambdaDemo {

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

```
  Runnable r1 = ()->
  {
   String name = Thread.currentThread().getName();
   System.out.println("Current Thread Name is :"+name);

  };

  Thread t1 = new Thread(r1,"Child1");
  t1.start();

 }

}
```

Case 3 :

```
package com.ravi.runnable_ex;

public class RunnableImplementationUsingConstructor {

 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 4 :

```
package com.ravi.runnable_ex;

public class RunnableImplementationUsingConstructorWithoutRef {

 public static void main(String[] args)
 {
  new Thread(new Runnable()
  {
     @Override
     public void run()
     {
      String name = Thread.currentThread().getName();
      System.out.println("Current Thread Name is :"+name);
     }
```

```
    }).start();
 }
}
```

Case 5 :

```
package com.ravi.runnable_ex;

public class LambdaImplementationInConstructor {

 public static void main(String[] args)
 {
  new Thread(()-> System.out.println(Thread.currentThread().getName())).start();

  new Thread(()-> System.out.println(Thread.currentThread().getName()),"Child1").start();
 }

}
```

Limitation 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 then multithreading is good but if the data is updatable data (modifiable data) then multithreading may produce some wrong result or wrong data as shown in the diagram. [08-JAN-25]

```
package com.ravi.runnable_ex;

class Customer implements Runnable
{
 private int availableSeat = 1;
 private int wantedSeat;

 public Customer(int wantedSeat)
 {
  super();
  this.wantedSeat = wantedSeat;
 }

 @Override
 public  void run()
 {
   String name = null;

   if(availableSeat >= wantedSeat)
   {
    name = Thread.currentThread().getName();
    System.out.println(wantedSeat+ " berth is reserved for :"+name);
    availableSeat = availableSeat - wantedSeat;
   }
```

```
   else
   {
    System.out.println("Available Seat is :"+availableSeat);
    name = Thread.currentThread().getName();
    System.err.println("Sorry!!"+name+ " berth is not available");
   }

 }
}

public class RailwayReservation {

 public static void main(String[] args)
 {
  Customer c1 = new Customer(1);

  Thread t1 = new Thread(c1,"Scott");
  Thread t2 = new Thread(c1,"Smith");

  t1.start();


  t2.start();
 }
}
```

Most of the time, both the Threads will get the ticket.

```
package com.ravi.runnable_ex;

class Customer
{
 private double availableBalance = 20000;
 private double withdrawAmount;

 public Customer(double withdrawAmount)
 {
  super();
  this.withdrawAmount = withdrawAmount;
 }

 public void withdraw()
 {
  String name = null;

  if(withdrawAmount <= availableBalance)
  {
   name = Thread.currentThread().getName();
   System.out.println(name+" has successfully withdraw :"+withdrawAmount+" amount");
   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) throws InterruptedException
 {
  Customer c1 = new Customer(20000);

  Runnable r1 = () -> c1.withdraw();


  Thread t1 = new Thread(r1,"Scott");
  Thread t2 = new Thread(r1,"Smith");

  t1.start();
  t2.start();

 }

}
```

09-01-2025

***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 a time for 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. [09-JAN]

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 (09-JAN-25)

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.

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.mt;

class Table
{
   public synchronized void printTable(int num)
   {
     for(int i=1; i<=10; i++)
     {
     try
     {
      Thread.sleep(1000);
     }
     catch(InterruptedException e)
     {
      e.printStackTrace();
     }
       System.out.println(num+" X "+i+" = "+(num*i));
     }
     String name = Thread.currentThread().getName();
     System.out.println(name+" thread is completed!!");
   }
}

public class MethodLevelSynchronization
{
```

```
 public static void main(String[] args)
 {
  Table obj = new Table(); //lock is created

  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.mt;

class ThreadName
{
 public void printThreadName()
 {
  String name = Thread.currentThread().getName();
  System.out.println("Running Thread name is :"+name);

  synchronized(this)
  {
   System.out.println("Synchronized block started by thread :"+name);
   for(int i = 1; i<=10; i++)
   {
    System.out.println("i value is :"+i+" by "+name);
   }
   System.out.println("Synchronized block ended by thread :"+name);
  }
 }


}


public class BlockLevelSynchronization {
```

```
 public static void main(String[] args)
 {
  ThreadName tn = new ThreadName();

  Runnable r1 = ()-> tn.printThreadName();

  Thread t1 = new Thread(r1,"Child1");
  Thread t2 = new Thread(r1,"Child2");

  t1.start();  t2.start();

 }

}
```

10-01-2025

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.
[09-JAN-25]

```
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);
        }
        catch(Exception e)
        {
        }
       }
       System.out.println(".....................");
    }
}

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

```
  PrintTable pt1 = new PrintTable(); //lock1 [2 , 3]
  PrintTable pt2 = new PrintTable(); //lock2 [8 , 9]

  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();
 }
}
```

Here we are getting exprected output because two locks are available from two differenet object. It is clear that synchronization logic will not work with multiple objects.

To avoid this problem, We introduced static synchronization.

**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)
{

}
```

```
package com.ravi.advanced;
class MyTable
{
  public static synchronized void printTable(int n)  //static synchronization
     {
        for(int i=1; i<=10; i++)
        {
         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();

  }
}
```

Thread Priority :

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.

```
package com.ravi.priority;

public class PriorityDemo1 {

 public static void main(String[] args)
 {
  Thread t1 = new Thread();
  System.out.println("Priority is :"+t1.getPriority());

 }

}

package com.ravi.priority;

class UserThread extends Thread
{

}

public class PriorityDemo2 {

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

```
  int priority = Thread.currentThread().getPriority();
        System.out.println("Main Thread priority is :"+priority);

        UserThread ut = new UserThread();
        System.out.println("User Thread priority is :"+ut.getPriority());


 }

}
```

ote : By default every thread even main thread is having default priority i.e 5.

```
package com.ravi.priority;

class Foo extends Thread
{

}

public class PriorityDemo3
{
 public static void main(String[] args)
 {
       Thread t = Thread.currentThread();
       t.setPriority(Thread.MAX_PRIORITY);


       Foo f1 = new Foo();
       System.out.println("Child Thread priority is :"+f1.getPriority());

       //f1.setPriority(11); //Invalid [java.lang.IllegalArgumentException]




 }

}
```

Note : Always child thread will acquire main thread priority.

```
package com.ravi.priority;

class Priority extends Thread
{
 @Override
 public void run()
 {
  int count = 0;
  for(int i = 1; i<=1000000; i++)
  {
   count++;
  }
```

```
  int priority = Thread.currentThread().getPriority();
  String name = Thread.currentThread().getName();

  System.out.println("The priority is :"+priority+" and thread name is :"+name);

 }
}

public class PriorityDemo4
{
 public static void main(String[] args)
 {
  Priority p1 = new Priority();
  Priority p2 = new Priority();

  p1.setPriority(Thread.MIN_PRIORITY);
  p2.setPriority(Thread.MAX_PRIORITY);

  p1.setName("Last");
  p2.setName("First");

  p1.start();  p2.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.

Lab Task :

Problem Statement:

You are tasked with creating an education institute course enrollment system using Java. The system should provide courses and offers to students, allowing them to view available courses, ongoing offers, and enroll in their preferred courses.

Classes:

class Course:

Attributes:

-> id (int): Unique identifier for the course.

-> name (String): Name of the course.

-> fee (double): Fee for the course.

Methods:

-> Course(int id, String name, double fee): Constructor to initialize the course attributes.

-> getId(): Returns the course ID.

-> getName(): Returns the course name.

-> getFee(): Returns the course fee.

class Offer:

Attributes:

-> offerText (String): Description of the special offer provided by the education institute.

Methods:

-> Offer(String offerText): Constructor to initialize the offer description.

-> getOfferText(): Returns the offer description.

class EducationInstitute:

Attributes:

-> courses (Course[]): An array to store the available courses.

-> offers (Offer[]): An array to store ongoing offers.

Methods:

-> EducationInstitute(): Constructor to initialize courses and offers.

-> getCourses(): Returns the array of available courses.

-> getOffers(): Returns the array of ongoing offers.

-> enrollStudentInCourse(int courseId, String studentName): Simulates the enrollment process and prints a message when a student -> enrolls in a course.

class Student:

Attributes:

-> name (String): Name of the student.

-> institute (EducationInstitute): Reference to the education institute where the student interacts.

Methods:

-> Student(String name, EducationInstitute institute): Constructor to initialize the student with their name and the education institute reference.

-> viewCoursesAndFees(): Displays the available courses and their fees.

-> viewOffers(): Displays the ongoing offers.

-> enrollInCourse(int courseId): Enrolls the student in the specified course using the education institute's enrollment process.

class Main :

The EducationInstituteApp class is the main program that simulates concurrent student interactions using threads. It creates an education institute, initializes students, and allows them to view course details, ongoing offers, and enroll in courses concurrently without disturbing the execution flow of each thread.

Instructions for Students:

-> Implement the above classes and their methods following the given specifications.

-> Create an instance of EducationInstitute, and initialize courses and offers with hardcoded data for simplicity.

-> Create two students: "John" and "Alice". Allow them to view available courses, check ongoing offers, and enroll in their preferred courses concurrently using threads.

-> Use the Thread class to simulate concurrent student interactions. Ensure that the system provides a responsive user experience for multiple students.

-> Test your program with multiple executions and verify that students can view course details, offers, and enroll without conflicts.

-> Feel free to enhance the program with additional features or error handling to further improve its functionality.

[Note : Include appropriate comments and use meaningful variable names to make your code more

readable and understandable.]

Sample Output :

Available Courses:

1. Mathematics - Fee: Rs.1000.0

2. Science - Fee: Rs.1200.0

3. English - Fee: Rs.900.0

Ongoing Offers:

Special discount: Get 20% off on all courses!

Limited time offer: Enroll in any two courses and get one course free!

Virat has enrolled in the course: Mathematics

Available Courses:

1. Mathematics - Fee: Rs.1000.0

2. Science - Fee: Rs.1200.0

3. English - Fee: Rs.900.0

Ongoing Offers:

Special discount: Get 20% off on all courses!

Limited time offer: Enroll in any two courses and get one course free!

Dhoni has enrolled in the course: Science

```
package com.ravi.educational_institute;

public class Course {
 private int courseId;
 private String courseName;
 private double courseFee;

 public Course(int courseId, String courseName, double courseFee) {
  super();
  this.courseId = courseId;
  this.courseName = courseName;
  this.courseFee = courseFee;
 }
```

```
 public int getCourseId() {
  return courseId;
 }

 public String getCourseName() {
  return courseName;
 }

 public double getCourseFee() {
  return courseFee;
 }

 @Override
 public String toString() {
  return "Course [courseId=" + courseId + ", courseName=" + courseName + ", courseFee=" + courseFee
+ "]";
 }

}


package com.ravi.educational_institute;

public class Offer
{
  private String offerText;

 public Offer(String offerText) {
  super();
  this.offerText = offerText;
 }

 public String getOfferText() {
  return offerText;
 }



}


package com.ravi.educational_institute;

public class EducationInstitute
{
  private Course  courses[];
  private Offer offers[];
  private int count;

 public EducationInstitute(Course[] courses, Offer[] offers) {
  super();
  this.courses = courses;
  this.offers = offers;
 }
```

```
 public Course[] getCourses() {
  return courses;
 }



 public Offer[] getOffers() {
  return offers;
 }



 public int getCount() {
  return count;
 }


 public void enrollStudentInCourse(int courseId, String studentName)
 {
  for(int i=0; i<courses.length; i++)
  {
   if(courseId == courses[i].getCourseId())
   {
    System.out.println(studentName +" enrolled in : "+courses[i].getCourseName());
   }
  }
 }

}

  package com.ravi.educational_institute;

public class Student
{
  private String name;
  private EducationInstitute educationInstitute;


  public Student(String name, EducationInstitute educationInstitute) {
 super();
 this.name = name;
 this.educationInstitute = educationInstitute;
}

  public void viewCoursesAndFees()
  {
  Course[] courses = educationInstitute.getCourses();

  for(Course course : courses)
  {
```

```
   System.out.println(course.getCourseId()+" : "+course.getCourseName()+" : "+course.getCourseFee());
  }
  }

  public void viewOffers()
  {
   Offer[] offers = educationInstitute.getOffers();
   for(Offer offer : offers)
   {
    System.out.println(offer.getOfferText());
   }
  }

  public void enrollInCourse(int courseId)
  {
   educationInstitute.enrollStudentInCourse(courseId, name);
  }
}

package com.ravi.educational_institute;

public class Main {

 public static void main(String[] args) throws InterruptedException
 {
  Course [] course = {new Course(1, "Java", 29000),new Course(2, ".Net", 25000), new Course(3,
"Python", 27000)};
  Offer [] offer = {new Offer("Special discount: Get 20% off on all courses!"), new Offer("Limited time offer:
Enroll in any two courses and get one course free!")};
  EducationInstitute ei = new EducationInstitute(course, offer);

  Student john = new Student("John",ei);
  Student alice = new Student("Alice",ei);

  Thread t1 = new Thread()
   {
       @Override
       public void run()
       {
        System.out.println("Available Courses and Fees :");
        john.viewCoursesAndFees();
        System.out.println("Available Offers :");
        john.viewOffers();
        john.enrollInCourse(1);


       }
   };




  Thread t2 = new Thread()
  {
      @Override
```

```
        public void run()
        {
        System.out.println("Available Courses and Fees :");
        alice.viewCoursesAndFees();
        System.out.println("Available Offers :");
        alice.viewOffers();
        alice.enrollInCourse(3);


        }
   };
   t1.start();
   t1.join();
   System.out.println(".....................");
   System.out.println();
         t2.start();
 }
}
```

Thread.yield() :[Prevent from over-utilisation a CPU]

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.

```
package com.nit.testing;

class Test implements Runnable
{
 @Override
 public void run()
 {
  String name = Thread.currentThread().getName();

  for(int i=1; i<=10; i++)
  {
   System.out.println(i+" by "+name+ " thread");

   if(name.equals("Child1"))
```

```
   {
    Thread.yield();  //give a chance to child2
   }

  }
 }
}

public class YieldDemo {

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

  Thread child1 = new Thread(t1, "Child1");
  Thread child2 = new Thread(t1, "Child2");

  child1.start();  child2.start();

 }

}
```

Note : In the above program, Generally when child1 thread is
in running state then it will give a chance to child2
so most of the time child1 will execute for single cycle inside the loop.

20-01-2025

** 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 java.lang.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(wait()) will get the notification from the another thraed using notify()/notifyAll() method then the waiting thread will move from Blocked state to Runnable state(Ready to run state) but it will continue its execution after getting the lock.

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.

*** Difference between sleep() and wait() [20-JAN-25]

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();

 }



}
```

```
package com.ravi.itc;

class Test extends Thread
{
 private int val = 0;  // 1  3  6  10  15  21

 @Override
 public void run()
 {
  for(int i=1; i<=100; i++)
  {
   val = val + i;
  }
```

```
 }

 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(1);

       System.out.println(t1.getVal());

       System.out.println("Main Thread ended!!!");

 }

}
```

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()
 {
  //child thread will wait for Object lock
  synchronized(this)
  {
   System.out.println("Loop Started");
   for(int i=1; i<=10; i++)
   {
    val = val + i;
   }
   System.out.println("Sending notification to main thread");
```

```
   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)
  {
   //Suspended
   System.out.println("Waiting for child thread to complete");
   System.out.println("Lock is released");
   d1.wait();
   System.out.println("Main Thread wake up");
   System.out.println(d1.getVal());
  }


 }

}
```

Note : Here we have co-ordination between main thread and child thread so we will get predicatable output.

21-01-2025

```
//Program on ITC where son can withdraw the amount and father
  can deposit the amount.

package com.ravi.itc;

class Customer
{
 private double balance = 10000;

 public synchronized void withdraw(double amount)
```

```
 {
  System.out.println("Son is going to withdraw...");

  if(amount > balance)
  {
   System.out.println("Less amount, Waiting for deposit..");
   try
   {
    wait();  //lock is released
    System.out.println("Got Notification, Going for withdraw");
   }
   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("Going to deposit the amount...");
    this.balance = this.balance + amount;
    System.out.println("Deposit Completed, Balance after deposit is :"+this.balance);
    notify();
 }

}

public class ITCDemo4
{
 public static void main(String[] args)
 {
  Customer cust = new Customer();

  Thread son = new Thread()
  {
     @Override
     public void run()
     {
      cust.withdraw(15000);
     }
  };

  son.start();

  Thread father = new Thread()
  {
     @Override
     public void run()
     {
      cust.deposit(10000);
```

```
    }
  };

  father.start();
 }

}
```

```
//Program to show how to cancel and book the ticket on the same TicketSystem  object

package com.ravi.itc;

class TicketSystem
{
   private int availableTickets = 5;   //availableTickets = 5

   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;  //5 - 4

      System.out.println("Booked " + numberOfTickets + " ticket(s). Remaining tickets: " +
availableTickets);
      notify();

   }


  public synchronized void cancelTicket(int numberOfTickets)//numberOfTickets = 2
   {
      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 = 4
                {
                    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
            public void run()
         {
                int[] ticketsToCancel = {1, 3, 2};

                for (int ticket : ticketsToCancel) //ticket = 2
                {
                    ticketSystem.cancelTicket(ticket);
                    try
                    {
                        Thread.sleep(1500);  // give some time b/w cancellation
                    }
                    catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        };
        cancellationThread.start();




    }
}
```

Note : From the above program it is clear that we can use
notify() and wait(), both the methods in the a userdefined method together.

```
//Program on notifyAll() method :
--------------------------------
package com.ravi.itc;

class Resource
{
    private boolean flag = false;

    public synchronized void waitMethod() //child1  child2  child3
 {
  System.out.println("Wait");

        while (!flag)
  {
          try
    {
             System.out.println(Thread.currentThread().getName() + " is waiting...");
             System.out.println(Thread.currentThread().getName()+" is Waiting for Notification");
             wait();
          }
    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();
    }
}
```

22-01-2025

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();
     for(int i=1; i<=3; i++)
     {
      System.out.println("i value is :"+i+" by "+name+ " thread");
      try
      {
       Thread.sleep(500);
```

```
      }
      catch(InterruptedException e)
      {
       e.printStackTrace();
      }
     }

 }
}

public class ThreadGroupDemo1
{
  public static void main(String[] args) throws InterruptedException
  {

  ThreadGroup tg = new ThreadGroup("Batch 39");

  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 39 group :"+tg.activeCount());

 System.out.println("Name of the the Group is :"+tg.getName());



  }
}
```

```
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 tatkal = new ThreadGroup("Tatkal");
      ThreadGroup premimumTatkal = new ThreadGroup("Premimum_Tatkal");


      Thread t1 = new Thread(tatkal, new TatkalTicket(), "t1");
      Thread t2 = new Thread(tatkal, new TatkalTicket(), "t2");
      Thread t3 = new Thread(tatkal, new TatkalTicket(), "t3");
      t1.start();  t2.start();  t3.start();


      Thread t4 = new Thread(premimumTatkal,new PremiumTatkal(), "t4");
      Thread t5 = new Thread(premimumTatkal,new PremiumTatkal(), "t5");
      Thread t6 = new Thread(premimumTatkal,new PremiumTatkal(), "t6");
      t4.start(); t5.start(); t6.start();


 }

}
```

```
package com.ravi.group;

public class ThreadGroupDemo2 {

 public static void main(String[] args)
 {
  Thread t = Thread.currentThread();
  System.out.println(t.toString());
 }

}
```

Note : Here we will get the output as Thread[#1,main,5,main]

1 : Id of the main thread
main : Name of the main thread
5 : default priority of main thread
main : Group name, under which main thread is running

==========

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<=19; i++)
  {
                System.out.println("User Thread: " + i);
                try
   {
                    Thread.sleep(2000);
                }
    catch (InterruptedException e)
    {
                    e.printStackTrace();
                }
            }
        });

        userThread.start();
```

```
        System.out.println("Main Thread Ended...");
    }
}
```
---
public void 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.

```
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
 }
}
```

23-01-2025

```
class Interrupt extends Thread
{
   @Override
   public void run()
 {
    try
    {
     Thread.currentThread().interrupt(); //self interruption

    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 hence for loop
       will be executed only one time.

```
public class InterruptThread2
{
    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.");
        }
    }
}
```

Note : If main thread will not interrupt the child thread then child thread will not come out from infinite while loop hance
the lock will not be released.

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. (23-JAN-25)

```
public class DeadlockExample
 {
  public static void main(String[] args)
  {
    String resource1 = new String("Ameerpet");  //(L1)
```

```
   String resource2 = new String("S R Nagar"); //(L2)

  // t1 tries to lock resource1(L1) then resource2(L2)

  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)
     {}

  //t1 thread is waiting here for Lock2
    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)
   {}
 //t2 thread will wait for L1  (Resourse1)
        synchronized (resource1) //Nested synchronized block
  {
          System.out.println("Thread 2: locked resource 1");
        }
      }
    }
  };
  t1.start();
  t2.start();
 }
```

}

Note : Here this situation is known as Deadlock situation because both the threads are waiting for infinite state.

New Thread life cycle :

New thread life cycle which is available from java 5V. Java software people has provided an enum called State (State is an enum which is defined inside Thread class)

A thread is well known for independent execution, During the life cycle of a thread it passes through different states which are as follows :

1) NEW (Thread Object is created)
2) RUNNABLE (Already Started but waiting for Processor time)
3) BLOCKED (Waiting for lock/Monitor)
4) WAITING (Waiting for another thread without timeout time)
5) TIMED_WAITING (Waiting with timeout time)
6) TERMINATED (Executed run())

NEW :

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 :

Whenever we call start() method on thread object, A thread moves to Runnable state i.e Ready to run state. Here the thread is considered "alive," but it doesn t immediately start execution unless the CPU scheduler assigns it time.

BLOCKED :

If a thread is waiting for object lock OR monitor to enter inside synchronized area OR re-enter inside synchronized area then it is in blocked state.

WAITING :

A thread in the waiting state is waiting for another thread to perform a particular action but WITHOUT ANY TIMEOUT time. A thread that has called wait() method on an object is waiting for another thread to call notify() or notifyAll() on the same object OR A thread that has called join() method is waiting for a specified thread to terminate.

TIMED_WAITING :

A thread in the timed_waiting state, if we call any method which put the thread into temporarly timed_waiting state but WITH POSITIVE TIMEOUT period like sleep(lons ms), join(long ms), wait(long ms) then the Thread is considered as Timed_Waiting state.

TERMINATED :

The thread has successuflly completed it’s execution in the separate stack memory.

24-01-2025

Volatile Keyword in java :

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.

```
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();
    }
}
```

Note : In the above program, remove the volatile keyword and verify the output.

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]

```
package com.ravi.clone_method;

class Employee implements Cloneable
{
 private Integer employeeId;
 private String employeeName;

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

```
 @Override
 public Object clone() throws CloneNotSupportedException
 {
  return super.clone();
 }

 public void setEmployeeId(Integer employeeId) {
  this.employeeId = employeeId;
 }

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


 @Override
 public String toString()
 {

  return "Employee [employeeId=" + employeeId + ", employeeName=" + employeeName + "]";
 }

}
public class CloneMethodDemo
{
 public static void main(String[] args) throws CloneNotSupportedException
 {
  Employee e1 = new Employee(111,"Scott");

  Employee e2 = (Employee) e1.clone();  //Down-casting
  e2.setEmployeeId(999);
  e2.setEmployeeName("Raj");

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

  System.out.println(e1.hashCode());
  System.out.println(e2.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.

```
package com.ravi.finalize;

record Product(Integer productId)
{
 @Override
 public void finalize()
 {
  System.out.println("Product Object is eligible for GC");
 }
}
public class FinalizeDemo
{
 public static void main(String[] args) throws InterruptedException
 {
        Product p1 = new Product(111);
        System.out.println(p1);

        p1 = null;

        System.gc(); //Explicitly calling GC

        Thread.sleep(3000);

        System.out.println(p1);

 }

}
```

25-01-2025

** What is the difference final, finally and finalize() method :

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 is automatically called by JVM 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().

Collections Framework : (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 utility class which is available from JDK 1.2V which contains only static methods (Constructor is private)

27-01-2025

Collection Hierarchy :

Hierarchy is available in Paint Diagram (27-JAN-25)

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 Hierarchy :

Available in Paint Digram [28-JAN-25]

List interface :

List interface is a sub interface of Collection in java.util package available from JDK 1.2V

List interface, Internally uses array concept so all the elements will be stored based on the index.

Duplicates are allowed.

null, homogeneoues, hetrogeneous values are allowed.

We can perform automatic sorting by using Collections.sort(List list) because sort() method accept List as a parameter.

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. [28-JAN]

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.

29-01-2025

How many ways we can fetch the 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 forEach() loop.[JDK 1.5]
4) By uisng Enumeration interface.[JDK 1.0]
5) By using Iterator interface.[JDK 1.2]
6) By using ListIterator interface.[JDK 1.2]
7) By using SplIterator interface.[JDK 1.8]
8) By using forEach() method.[JDK 1.8]
9) By using Method Reference.[JDK 1.8]

Note : Among all these 9 ways Enumeration, Iterator, ListIterator and SplIterator are the cursors so It 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<E> 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 extends from Itr 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 :

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> cons) method for fetching the elements.

forEach(Consumer<T> cons)

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 :

```
package com.ravi.for_each_method_internals;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternalDemo1 {

 public static void main(String[] args)
 {
  Vector<String> fruits = new Vector<>();
  fruits.add("Orange");
  fruits.add("Apple");
  fruits.add("Mango");
  fruits.add("Kiwi");
  fruits.add("Grapes");

  //Anonymous inner class
  Consumer<String> cons = new Consumer<String>()
  {
   @Override
   public void accept(String fruit)
   {
      System.out.println(fruit.toUpperCase());
   }
  };

  fruits.forEach(cons);

 }

}
```

Case 2 :

```
package com.ravi.for_each_method_internals;

import java.util.Vector;
import java.util.function.Consumer;
```

```
public class ForEachMethodInternalDemo2
{

 public static void main(String[] args)
 {
  Vector<String> fruits = new Vector<>();
  fruits.add("Orange");
  fruits.add("Apple");
  fruits.add("Mango");
  fruits.add("Kiwi");
  fruits.add("Grapes");

  //Lambda
  Consumer<String> cons = fruit -> System.out.println(fruit.toUpperCase());
  fruits.forEach(cons);

 }

}
```

Case 3 :

```
package com.ravi.for_each_method_internals;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternalDemo3 {

 public static void main(String[] args)
 {
  Vector<String> fruits = new Vector<>();
  fruits.add("Orange");
  fruits.add("Apple");
  fruits.add("Mango");
  fruits.add("Kiwi");
  fruits.add("Grapes");

 fruits.forEach(fruit -> System.out.println(fruit.toUpperCase()));

 }

}
```

WAP that describes how to retrieve the Objects by using above 9 ways :

```
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
{
 public static void main(String[] args)
 {
  Vector<String> fruits = new Vector<>();
  fruits.add("Orange");
  fruits.add("Apple");
  fruits.add("Mango");
  fruits.add("Kiwi");
  fruits.add("Grapes");

  System.out.println("1) BY USING TOSTRING METHOD :");
  System.out.println(fruits.toString());

  System.out.println("2) BY USING ORDINARY FOR LOOP :");

  for(int i=0; i<fruits.size(); i++)
  {
   System.out.println(fruits.get(i));
  }


  System.out.println("3) BY USING FOR EACH LOOP :");

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

  System.out.println("4) BY USING ENUMARATION INTERFACE :");

     Enumeration<String> ele = fruits.elements();

     while(ele.hasMoreElements())
     {
      System.out.println(ele.nextElement());
     }

     System.out.println("5) BY USING ITERATOR INTERFACE :");

     Iterator<String> itr = fruits.iterator();
     itr.forEachRemaining(fruit -> System.out.println(fruit));


     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("7) BY USING SPLITERATOR INTERFACE");

    Spliterator<String> splItr = fruits.spliterator();
    splItr.forEachRemaining(fruit -> System.out.println(fruit));

    System.out.println("8) BY USING FOREACH METHOD :");
    fruits.forEach(fruit -> System.out.println(fruit));

    System.out.println("9) BY USING METHOD REFERENCE :");
    fruits.forEach(System.out::println);

 }
}
```

30-01-2025

Working with List interface Specific 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>

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.

Here Iterator is Fail Fast Iterator.

It 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

```
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");

  System.out.println("Before Sorting :"+listOfCity);

        Collections.sort(listOfCity);

        System.out.println("After Sorting :"+listOfCity);

        //Remove the element based on the index position
```

```
        listOfCity.remove(2);
        System.out.println(listOfCity);

        //Remove based on the Object
        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<>(100,9);

  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());

  for(int i=0; i<v.size(); i++)
  {
   if(i%5==0)
   {
    System.out.println();
   }
   System.out.print(v.get(i)+"\t");
  }


 }
}
```

```
package com.ravi.vector;

//Array To Collection
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));

  //Vector to Array
   Object[] array = v.toArray();
  System.out.println("Vector to array");
  System.out.println(Arrays.toString(array));

 }
}
```

```
package com.ravi.vector;

import java.util.Vector;

record Prod(Integer id, String name)
{

}

public class VectorDemo3
{
 public static void main(String[] args)
 {
    Vector<Prod> listOfProduct = new Vector<>();
    listOfProduct.add(new Prod(333, "Mobile"));
    listOfProduct.add(new Prod(111, "Camera"));
    listOfProduct.add(new Prod(222, "Laptop"));
    listOfProduct.add(new Prod(444, "Tablet"));

    listOfProduct.forEach(System.out::println);
 }
}
```

31-01-2025

What is Fail Fast Iterator in Collection ?

While retrieving the object from the collection by using Itearor interface or for each loop, if at any point of time the original structure is going to modify after the creation of Itearator then we will get java.util.ConcurrentModificationExacption.

```
package com.nit.collection;

import java.util.Iterator;
import java.util.Vector;

class Concurrent extends Thread
{
 private Vector<String> cities = null;

 public Concurrent(Vector<String> cities)
 {
  super();
  this.cities = cities;
 }

 @Override
 public void run()
 {
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }

  cities.add("Ameerpet");

 }
}


public class FailFastIterator {

 public static void main(String[] args) throws InterruptedException
 {
  Vector<String> cityName = new Vector<>();
  cityName.add("Indore");
  cityName.add("Bhubneswar");
  cityName.add("Hyderabad");
  cityName.add("Mumbai");
  cityName.add("Pune");

  Concurrent c1 = new Concurrent(cityName);
  c1.start();

  Iterator<String> itr = cityName.iterator();

  while(itr.hasNext())
  {
```

```
   System.out.println(itr.next());
   Thread.sleep(500);
  }


 }

}
```

In the above program we will get java.util.ConcurrentModificationException because Iterator is Fail Fast iterator hence while iterating the element if the structure will be modified then exception will be generated.

In order to resolve the issue, Java software people has provided
a new concept in a new package i.e java.util.concurrent sub package which is introduced from JDK 1.5V.

Here we have basically two advantages :

 1) Here classes are immutable classes (Unchanged classes)
 2) Here Iteartor is Fail safe iterator but not fail fast.

```
package com.nit.collection;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Spliterator;
import java.util.concurrent.CopyOnWriteArrayList;

class Concurrent extends Thread
{
 private CopyOnWriteArrayList<String> cities = null;

 public Concurrent(CopyOnWriteArrayList<String> cities)
 {
  super();
  this.cities = cities;
 }

 @Override
 public void run()
 {
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }

  cities.add("Ameerpet");

 }
}
```

```
public class FailFastIterator {

 public static void main(String[] args) throws InterruptedException
 {
  CopyOnWriteArrayList<String> cityName = new CopyOnWriteArrayList<>();
  cityName.add("Indore");
  cityName.add("Bhubneswar");
  cityName.add("Hyderabad");
  cityName.add("Mumbai");
  cityName.add("Pune");

  Concurrent c1 = new Concurrent(cityName);
  c1.start();

  Iterator<String> itr = cityName.iterator();

  while(itr.hasNext())
  {
   System.out.println(itr.next());
   Thread.sleep(500);
  }

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

     Spliterator<String> spliterator = cityName.spliterator();
     spliterator.forEachRemaining(System.out::println);

 }

}
```

03-02-2025

Program that describes ArrayList is better than Vector in performance wise :

As we know ArrayList methods are not synchronized so multiple threads can access the method of ArrayList, on the other hand most of the methods are synchronized in Vector class.

java.lang.System class has provided a predefined static method called currentTimeMillis() through which we can get the current system time in millisecond.

```
public static native long currentTimeMillis();
```

```
//Program to describe that ArrayList is better than Vector in performance

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();

  System.out.println("Total time taken by ArrayList class :"+(endTime - startTime)+" ms ");

        Vector<Integer> v1 = new Vector<>();

     startTime = System.currentTimeMillis();

  for(int i=0; i<1000000; i++)
  {
   v1.add(i);
  }

   endTime = System.currentTimeMillis();

  System.out.println("Total time taken by Vector class :"+(endTime - startTime)+" ms ");


 }
}
```

Note : Performance wise ArrayList is better than Vector becoz ArrayList methods are not synchronized.

```
package com.ravi.vector;

import java.util.Arrays;
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");


  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...");
  //sort(List list, Comparator<T> comp);
  Collections.sort(listOfNumbers, Collections.reverseOrder());
  System.out.println(listOfNumbers);

  //Converting Our Vector(Collection Object) into Array
  Vector<String> listOfFruits = new Vector<>();
  listOfFruits.add("Orange");
  listOfFruits.add("Apple");
  listOfFruits.add("Mango");

  Object[] fruits = listOfFruits.toArray();
  System.out.println(Arrays.toString(fruits));



 }

}
```

Methods used in the above program :

---

1) public void sort(List<E> list, Comparator<T> cmp) : It is sort the
   list based on the specified Comparator.

2) public Object[] toArray() : It is used to convert the Collection object into Object array.

```
   package com.ravi.vector;

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:
                // 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();
    }
}
```

---

public Iterator asIterator() : It is a default method provided inside Enumeration interface from java 9V. It will return Iteartor interface Object so we can apply Iterator interface method.

```
package com.ravi.vector;

import java.util.Enumeration;
import java.util.Iterator;
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> ele = listOfProduct.elements();


  Iterator<Product> itr = ele.asIterator();
  itr.forEachRemaining(System.out::println);


 }

}
```

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

```
//Program to insert and fetch the elements from stack
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());
             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
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));

            Stack<Double> st4 = new Stack<>();
            st4.add(10.5);
            st4.add(20.5);
            st4.forEach(x -> System.out.println(x));
      }
}
```

```
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);
  }
}
```

```
//Searching an element in the Stack
package com.ravi.stack;
import java.util.Stack;  //
public class Stack4
{
 public static void main(String[] args)   //1  -1  false  2
  {

   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
  }
}
```

04-02-2025

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 so performance wise it is good.

*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)

//Program on loose coupling :

```
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.Vector;

public class LooseCoupling {

 public static void main(String[] args)
 {
  Vector<Integer> v1 = new Vector<Integer>();
  v1.add(12);
  v1.add(29);
  v1.add(59);
  v1.add(24);

  ArrayList<Integer> list = new ArrayList<>(v1);
  System.out.println(list);


 }

}
```

```
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);
    }
}
```

```
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.Collections;

record Customer(Integer custId, String custName, Double custSal)
{
}

public class ArrayListDemo1
{
 public static void main(String[] args)
 {
  ArrayList<Customer> listOfCustomer = new ArrayList<>();
  listOfCustomer.add(new Customer(111, "Scott", 800D));
  listOfCustomer.add(new Customer(222, "Smith", 1200D));
  listOfCustomer.add(new Customer(333, "Alen", 1800D));
  listOfCustomer.add(new Customer(444, "Martin", 1500D));
  listOfCustomer.add(new Customer(555, "John", 1300D));


  listOfCustomer.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/remove an element then we will get an Excption java.lang.UnsupportedOperationException.

List<E> list = Arrays.asList(T ...x);

FixedLengthArray.java

```
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);
  //numbers.add(8); java.lang.UnsupportedOperationException
  //numbers.add(0, 100); java.lang.UnsupportedOperationException
  //numbers.remove(0);  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 remove or replace operation otherwise we will get java.lang.UnsupportedOperationException.

List<E> list = List.of(E ...x)

```
package com.ravi.arraylist;

import java.util.List;

public class ImmutableList {

 public static void main(String[] args)
 {
  List<Integer> list = List.of(1,2,3,4,5,6,7,8,9,10,11,12);
        //list.add(13);  java.lang.UnsupportedOperationException
  //list.remove(0); java.lang.UnsupportedOperationException
  //list.set(0, 100); java.lang.UnsupportedOperationException
  System.out.println(list);

 }

}
```

//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

  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());
  }

  }
}
```

Serialization and Deserialization on ArrayList object :

```
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.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
 var fout = new FileOutputStream("D:\\new\\IceCreamFlavors.txt");
 var oos = new ObjectOutputStream(fout);

 try(oos; fout)
 {
  oos.writeObject(listOfIceCream);
  System.out.println("Object Data stored Successfully!!!");
 }
 catch(Exception e)
 {
  e.printStackTrace();
 }

 //De-Serialization

 var fin = new FileInputStream("D:\\new\\IceCreamFlavors.txt");
 var ois = new ObjectInputStream(fin);

 try(ois; fin)
```

```
 {

  @SuppressWarnings("unchecked")
  ArrayList<String> list = (ArrayList<String>) ois.readObject();
     list.forEach(System.out::println);
 }
 catch(Exception e)
 {
  e.printStackTrace();
 }

  }
}
```

Note : In the above program, String and ArrayList both the classes implements from java.io.Serializable.

```
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
{
}


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)
  {

   @SuppressWarnings("unchecked")
   ArrayList<Employee> empList = (ArrayList<Employee>) ois.readObject();
   empList.forEach(System.out::println);

  }
  catch(Exception e)
  {
   e.printStackTrace();
  }


 }

}
```

Collections.sort(List<E> list, Comparator<T> comp):

Collections class has provided static method called reverseOrder()
to reverse the Collection data, the return type of this method is Comparator<T> interface.

```
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);

    }
```

```
}
-----------------------------------------------------------------
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
{
 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
 }
}
-----------------------------------------------------------------
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 addProfessor(Professor prof)
 {
  this.professors.add(prof);
 }


 public String getDepartmentName()
 {
  return departmentName;
```

```
 }


 public List<Professor> getProfessors()
 {
  return professors;
 }

}

public class ArrayListDemo7
{
    public static void main(String[] args)
    {

     Department dept = new Department("Computer Science");
     dept.addProfessor(new Professor("Mr James","Java"));
     dept.addProfessor(new Professor("Mr Scott","Python"));
     dept.addProfessor(new Professor("Mr Smith","Adv Java"));
     dept.addProfessor(new Professor("Mr Martin",".NET"));

     System.out.println("Department Name is :"+dept.getDepartmentName());

     System.out.println("Professor in :"+dept.getDepartmentName()+" department :");

     List<Professor> professors = dept.getProfessors();
     professors.forEach(System.out::println);


    }
}
```

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)

```
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> duplicate =(ArrayList<String>) original.clone();
         System.out.println(duplicate);
```

```
        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 as a Predicate.

```
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.List;

public class ArrayListDemo9 {

 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)
        boolean contains = list.contains(9);
        System.out.println(contains);

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

        //public int indexOf(Object obj)
```

```
        System.out.println(list.indexOf(1));


         //public void removeIf(Predicate<T> p)
           list.removeIf(num -> num%2==1);
           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 :"+listOfNames);

  //Remove all the name which starts from character 'R'

  listOfNames.removeIf(str -> str.startsWith("R"));
  System.out.println("After removing :"+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.

```
package com.ravi.arraylist;

import java.util.ArrayList;
```

```
import java.util.RandomAccess;

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());



 }

}
```

Time Complexity of ArrayList :

The time complexity of ArrayList to insert OR 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 OR begning 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.

[05-FEB]

In order to insert and delete the element in middle of the list frequently, we introduced LinkedList class.

LinkedList<E>
---
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.

It stores the elements in non-contiguous memory location.

*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 Head node OR tail node OR closer one based on the index.

**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()

The time complexcity for insertion and deletion is O(1) The time complexcity for seraching O(n) because it serach the elemnts using node reference.

==========

```
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("1st Position Element is :"+list.get(1));

   //Iterator interface

    Iterator<Object> itr = list.iterator();
    itr.forEachRemaining(System.out::println); //JDK 1.8


  }
}
```

```
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
         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 : From the above program, It is clear that insertion and deletion in the LinkedList is very efficient due to doubly LinkedList data structure.

```
package com.ravi.linked_list;

//Methods of LinkedList class
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]
    }
}
```

```
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: ");

        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.");
        }
     }
  }
}
```

```
package com.ravi.linked_list;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

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);




 }

}
```

```
package com.ravi.linked_list;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

record Product(Integer productId, String productName)
{

}

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);
```

```
 }

}
```

```
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 :
(07-FEB-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).

It is more efficient than our classical array approach which works on the basis of index.

08-02-2025

HashSet<E> [UNORDERED, UNSORTED, 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 hs4 = new HashSet(Collection c);
   Interconversion of Collection.

```
------------------------------------------------------------
//Unsorted, Unordered and no duplicates
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(45);
  hs.add(12);
  hs.add(35);
  hs.add(35);
  hs.add(null);

  hs.forEach(num-> System.out.println(num));
 }
}
------------------------------------------------------------
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));

      }
}

Note : While working with HashSet order is un-predictable.
------------------------------------------------------------
package com.nit.collection;

import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo2 {

 public static void main(String[] args)
 {
  Boolean []arr = new Boolean[5];

  Set<Object> set = new HashSet<>();
```

```
  arr[0] =  set.add(12);
  arr[1] =  set.add(12);
  arr[2] =  set.add(14);
  arr[3] =  set.add(14);
  arr[4] =  set.add(15);

  System.out.println(Arrays.toString(arr));

  System.out.println("Fetching the Data from Set ");

  set.forEach(obj -> System.out.println(obj));

  //Verify whether 15 is available or not ?

  if(set.contains(15))
  {
   System.out.println("15 is available");
  }
  else
  {
   System.out.println("It is not available");
  }

 }

}
```

```
//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();
      }
   }
}
```

```
package com.nit.collection;

import java.util.HashSet;

public class HashSetDemo4 {

 public static void main(String[] args)
 {
  HashSet<String> hs1 = new HashSet<>();
  hs1.add(new String("india"));
  hs1.add(new String("india"));
  System.out.println(hs1.size());
  System.out.println(hs1);

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

  HashSet<StringBuffer> hs2 = new HashSet<>();
  hs2.add(new StringBuffer("india"));
```

```
  hs2.add(new StringBuffer("india"));
  System.out.println(hs2.size());
  System.out.println(hs2);



 }

}
```

String size is 1 but StringBuffer size will be 2 because hashCode() and equals(Object obj) methods are not overridden
in StringBuffer class.

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.

```
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));
 }
}
```

```
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());

       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.

10-02-2025

Comparable<T> and Comparator<T> interfaces :

---

1) Comparable<T> and Comparator<T> both are functional interfaces.

2) Both are available from JDK 1.2V.

Note : All the Wrapper classes as well as String class implements
from Comparable<T> so, all these classes provides the support of Object comparison using compreTo(T x) method.

*** Difference between Comparable<T> and Comparator<T> ----------
Available in paint Digram [10-FEB-25]

//Program on Comparable :

```
package com.ravi.comparable;

public record Customer(Integer id, String name) implements Comparable<Customer>
{
 @Override
 public int compareTo( Customer c2)
 {
   return this.id - c2.id;
 }
}
```

```
package com.ravi.comparable;

import java.util.ArrayList;
import java.util.Collections;

public class CustomerComparable {

 public static void main(String[] args)
 {
  ArrayList<Customer> listOfCustomers = new ArrayList<>();
  listOfCustomers.add(new Customer(333, "Scott"));
  listOfCustomers.add(new Customer(111, "Zuber"));
  listOfCustomers.add(new Customer(222, "Aryan"));

  System.out.println("Original Data :");
  listOfCustomers.forEach(System.out::println);

  Collections.sort(listOfCustomers);
  System.out.println("Data After sorting based on the ID :");
  listOfCustomers.forEach(System.out::println);


 }

}
```

```
package com.ravi.updated_array_25;

import java.util.Arrays;
```

```
record Employee(Integer id, String name) implements Comparable<Employee>
{

 @Override
 public int compareTo(Employee e2)
 {
  return this.id - e2.id;
 }

}

public class ArrayDemo15
{
 public static void main(String[] args)
 {
        Employee []employees = new Employee[3];

        employees[0] = new Employee(333, "Zuber");
        employees[1] = new Employee(222, "Aryan");
        employees[2] = new Employee(111, "Raj");

        System.out.println("Original Data :");

        for(Employee emp : employees)
        {
         System.out.println(emp);
        }

        System.out.println("Sorted Data Based On the ID:");

        Arrays.sort(employees);

        for(Employee emp : employees)
        {
         System.out.println(emp);
        }

 }
}
```

Limitation of Comparable interface :

We have 3 limitations with Comparable<T>

---

1) We need to modify the BLC class OR Original source code to provide current object support(this keyword), If the BLC class OR
source code is provided by any 3rd party developer and we are unable to modify the source code then Comparable will not work.

2) We can’t write multiple sorting logic, It provides the facility
to write only one sorting logic using compareTo(T x) method.

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.

In order to avoid the above said limitations, Java software people has introduced Comparator<T> interface.

//Program on Comparator<T>

```
package com.ravi.comparator;

public record Product(Integer id, String name)
{

}
```

```
package com.ravi.comparator;

import java.util.ArrayList;
import java.util.Collections;

public class ProductComparator {

 public static void main(String[] args)
 {
  ArrayList<Product> listOfProduct = new ArrayList<Product>();
  listOfProduct.add(new Product(333, "Laptop"));
  listOfProduct.add(new Product(111, "Mobile"));
  listOfProduct.add(new Product(222, "Camera"));

  System.out.println("Original Data :");
  listOfProduct.forEach(System.out::println);

  System.out.println("Sorted Data based on the Id :");
  Collections.sort(listOfProduct, (p1,p2)-> p1.id().compareTo(p2.id()));
  listOfProduct.forEach(System.out::println);


  System.out.println("Sorted Data based on the Name :");
  Collections.sort(listOfProduct, (p1,p2)-> p1.name().compareTo(p2.name()));
  listOfProduct.forEach(System.out::println);

 }

}
```

How to sort the Integer object in descending order by using Comparator :

```
package com.ravi.comparator;

import java.util.ArrayList;
import java.util.Collections;
```

```
public class IntegerDesc {

 public static void main(String[] args)
 {
  ArrayList<Integer> al = new ArrayList<>();
  al.add(300);
  al.add(400);
  al.add(100);
  al.add(200);

  Collections.sort(al,(i1,i2)-> i2.compareTo(i1));

  System.out.println(al);


 }

}
```

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);

 }
}
```

11-02-2025

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.

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 Objects 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.

```
//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);
 }
}
```

```
import java.util.*;
public class TreeSetDemo1
{
 public static void main(String[] args)
 {
  TreeSet<String> t1 = new TreeSet<>();
  t1.add("Orange");
  t1.add("Mango");
  t1.add("Pear");
  t1.add("Banana");
  t1.add("Apple");
  System.out.println("In Ascending order");
  t1.forEach(i -> System.out.println(i));

  TreeSet<String> t2 = new TreeSet<>();
  t2.add("Orange");
  t2.add("Mango");
  t2.add("Pear");
  t2.add("Banana");
  t2.add("Apple");

        System.out.println("In Descending order");
  Iterator<String> itr2 = t2.descendingIterator();  //for descending order

         itr2.forEachRemaining(x -> System.out.println(x));
 }
}
```

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()

//How to sort TreeSet by using Comparable :

```
package com.ravi.treeset;

import java.util.Iterator;
import java.util.TreeSet;

record Employee(Integer id, String name) implements Comparable<Employee>
{
 @Override
 public int compareTo(Employee e2)
 {
  return Integer.compare(this.id(), e2.id());
 }
```

```
}

public class TreeSetDemo3
{
 public static void main(String[] args)
 {
      TreeSet<Employee> ts1 = new TreeSet<Employee>();
      ts1.add(new Employee(333, "Raj"));
      ts1.add(new Employee(111, "Zuber"));
      ts1.add(new Employee(222, "Aryan"));
      System.out.println(ts1);

     Iterator<Employee> descItr = ts1.descendingIterator();
     descItr.forEachRemaining(System.out::println);

 }

}
```

How to sort TreeSet by using Comparator :

```
package com.ravi.treeset;

import java.util.TreeSet;

record Product(Integer id, String name)
{


}

public class TreeSetDemo4 {

 public static void main(String[] args)
 {
  TreeSet<Product> ts1 = new TreeSet<Product>((p1, p2)->p1.name().compareTo(p2.name()) );
        ts1.add(new Product(333, "Laptop"));
        ts1.add(new Product(111, "Mobile"));
        ts1.add(new Product(222, "Camera"));

        System.out.println(ts1);
 }

}
```

```
import java.util.*;
public class TreeSetDemo5
{
 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));


 }
}
```

```
import java.util.*;

public class TreeSetDemo6
 {
 public static void main(String[] args)
 {
  Set<Character> t = new TreeSet<>((c1,c2) -> c2.compareTo(c1));
  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));
 }
}
```

Sorting the Student Data based on the different Criteria :

```
package com.ravi.treeset;

import java.util.TreeSet;

record Student(Integer id, Double fees)
{

}

public class TreeSetDemo7
{
 public static void main(String[] args)
 {
    TreeSet<Student> ts1 = new TreeSet<>((s1,s2) -> Integer.compare(s1.id(), s2.id()));
       ts1.add(new Student(333, 25000D));
       ts1.add(new Student(222, 2200D));
       ts1.add(new Student(111, 20000D));

       System.out.println("Sorting student Data based on the ID :");

       for(Student st : ts1)
       {
        System.out.println(st);
       }
```

```
      TreeSet<Student> ts2 = new TreeSet<>((s1,s2) -> Double.compare(s1.fees(), s2.fees()));
      ts2.add(new Student(333, 25000D));
      ts2.add(new Student(222, 2200D));
      ts2.add(new Student(111, 20000D));

      System.out.println("Sorting student Data based on the Fees :");

      for(Student st : ts2)
      {
       System.out.println(st);
      }

    }
}
```

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.

```
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<E>

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 values. Here we can navigate among the values as shown below.

```
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(6));//Just greater than specified element or null

      System.out.println("ceiling(3): " + ns.ceiling(3));//Equal or greater or null


   }
}
```

12-02-2025

Map<K,V> 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)

```
public interface Map<K,V>
{
    public 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.

Map<K,V> interface Hierarchy :

Available in paint diagram [12-FEB-25]

Methods of Map<K,V> 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 the 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 :

Map interface has provided the following methods to convert the
Map to Collection, these methods are called as Collection views
methods.

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()

**** 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.

14-02-2025

** 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 RETURNS TRUE THEN BOTH OBJECTS MUST RETURN SAME HASHCODE.

```
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 Object class hashCode()) 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 by using Singly LinkedList.

So, the conclusion is, compulsory we need to override both the methods for removing duplicate elements.

```
package com.ravi.map;

import java.util.HashMap;
import java.util.Objects;

class Customer
{
 private Integer customerId;
 private String customerName;


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

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

```
 @Override
 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 Objects.equals(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());
    System.out.println(c1.equals(c2));

    System.out.println(".....................");
    HashMap<Customer,String> map = new HashMap<>();
    map.put(c1, "A");
    map.put(c2, "B");

    System.out.println(map.size()); //1
    System.out.println(map); //{c1 = B}

 }

}
```

Customer class we are using as a HashMap key so it must override
hashCode() and equals(Object obj) as well as at 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.

```
package com.ravi.map;

import java.util.HashMap;
```

```
record Manager(Integer id, String managerName)
{

}


public class HashMapInternalDemo2 {

 public static void main(String[] args)
 {
  Manager m1 = new Manager(111,"Alen");
  Manager m2 = new Manager(111,"Alen");

  HashMap<Manager,String> map = new HashMap<>();
  map.put(m1, "Ameerpet");
  map.put(m2, "S R Nagar");

  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.

```
package com.ravi.map;

import java.util.Scanner;

public class CustomStringHashCode
{
    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();
    }
}
```

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 complexity 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) OR Use record

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

```
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())
        {
            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 = 103;
        String studentName = studentRecords.get(searchId);

        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);


    }
}
-------------------------------------------------------------
package com.ravi.map;

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");

  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);


  }
}
```

17-02-2025

```
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));
 }
}
```

```
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.");
      }
      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>

---

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);

```
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.*;

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>, Cloneable, 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

```
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());
   }
       }
}
```

```
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.nit.weak;

import java.util.WeakHashMap;

record Product(Integer id, String name)
{
 @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> map = new WeakHashMap<>();
  map.put(p1, "Hyderabad");

  System.out.println(map);

  p1 = null;

  System.gc();  //Calling GC explicitly

  Thread.sleep(3000);

  System.out.println(map); //{}


 }

}
```

18-02-2025

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 static native int identityHashCode(Object obj)
```

```
package com.ravi.map;

class Foo
{

}
public class SystemHashCode {

 public static void main(String[] args)
 {
  String str1 = "india";
  String str2 = new String("india");

  System.out.println(System.identityHashCode(str1));
  System.out.println(System.identityHashCode(str2));

  System.out.println(".....................");
  Foo f1 = new Foo();
  Foo f2 = new Foo();
```

```
  System.out.println(System.identityHashCode(f1));
  System.out.println(System.identityHashCode(f2));

 }

}
```

Note : Hashcode of both the objects will be different.

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 compare the keys by using 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)

```
package com.ravi.map;

import java.util.HashMap;
import java.util.IdentityHashMap;

public class IdentityHashMapDemo {

 public static void main(String[] args)
 {
  HashMap<String,String> hm = new HashMap<>();
  hm.put("NIT", "Ameerpet");
  hm.put(new String("NIT"), "Hyderabad");
  System.out.println(hm.size());
```

```
  System.out.println(hm);

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

  IdentityHashMap<String,String> ihm = new IdentityHashMap<>();
  ihm.put("NIT", "Ameerpet");
  ihm.put(new String("NIT"), "Hyderabad");
  System.out.println(ihm.size());
  System.out.println(ihm);


 }

}
```

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.tree_map;
```

```
import java.util.TreeMap;

public class TreeMapDemo1 {

 public static void main(String[] args)
 {
  TreeMap<Integer, String> tm1 = new TreeMap<>();
  tm1.put(4, "A");
  tm1.put(9, "B");
  tm1.put(1, "D");
  tm1.put(2, "Z");
  tm1.put(3, "X");

  tm1.forEach((k,v) -> System.out.println("Key is :"+k+" value is :"+v));



 }

}
```

```
import java.util.*;
public class TreeMapDemo
{
 public static void main(String[] args)
 {
  TreeMap<Object,String> t = new TreeMap<>();
  t.put(4,"Ravi");
  t.put(7,"Aswin");
  t.put(2,"Ananya");
  t.put(1,"Dinesh");
  t.put(9,"Ravi");
  t.put(3,"Ankita");
  t.put(5,null);
  //t.put("six", "Xyz");
  //t.put(null, "abc");

  System.out.println(t);
 }
}
```

Note : put() method, internally uses compareTo() method of Integer class to sort the key object in ascending order.

```
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);
```

```
          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));
      }
}
-------------------------------------------------------------
//firstKey()  lastKey()  headMap()  tailMap()  subMap()      SortedMap
// first()     last()     headSet()  tailSet()  subSet()     SortedSet

import java.util.*;
public class TreeMapDemo2
{
    public static void main(String[] argv)
    {
        Map<String,String> map = new TreeMap<String,String>();
        map.put("key2", "value2");
        map.put("key3", "value3");
        map.put("key1", "value1");

        System.out.println(map); //

        SortedMap x = (SortedMap) map;
        System.out.println("First key is :"+x.firstKey());
        System.out.println("Last Key is :"+x.lastKey());
     }
}
-------------------------------------------------------------
package com.ravi.tree_map;

import java.util.TreeMap;

record Product(Integer pid, String pname)
{

}
public class TreeMapDemo3
{
 public static void main(String[] args)
 {
  TreeMap<Product,String> tm1 = new TreeMap<>((p1, p2)-> p1.pid().compareTo(p2.pid()));
  tm1.put(new Product(333, "Laptop"), "Hyderabad");
  tm1.put(new Product(444, "Mobile"), "Pune");
  tm1.put(new Product(111, "HeadPhone"), "Indore");
  tm1.put(new Product(222, "Camera"), "Mumbai");

  System.out.println("Sorting based on the Product Id in Ascending Order");
  tm1.forEach((k,v)-> System.out.println(k+" : "+v));
```

```
  TreeMap<Product,String> tm2 = new TreeMap<>((p1, p2)-> p2.pid().compareTo(p1.pid()));
  tm2.put(new Product(333, "Laptop"), "Hyderabad");
  tm2.put(new Product(444, "Mobile"), "Pune");
  tm2.put(new Product(111, "HeadPhone"), "Indore");
  tm2.put(new Product(222, "Camera"), "Mumbai");

  System.out.println("Sorting based on the Product Id in Descending Order");
  tm2.forEach((k,v)-> System.out.println(k+" : "+v));
 }

}
```

```
package com.ravi.tree_map;

import java.util.HashMap;
import java.util.SortedMap;
import java.util.TreeMap;

public class TreeMapDemo4 {

 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);

 }

}
```

```
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[])
  {
  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)

19-02-2025

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 :

Create a db.properties file as shown below :
---
db.properties
---
driver = oracle.jdbc.driver.OracleDriver
userName = scott
password = tiger

PropertiesDemo1.java

```
import java.util.*;
import java.io.*;

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("driver");
   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.properties;

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";
  var writer = new FileWriter(filePath);

  var properties = new Properties();

  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();
  }

 }
}
```

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;

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);

      }
}
```

```
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);
   }
}
```

```
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");   //       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());
  }
}
```

```
package com.nit.collection;

import java.util.Comparator;
import java.util.PriorityQueue;

record Task(String name, Integer priority)
{
}


public class PriorityQueueDemo3 {
   public static void main(String[] args)
   {
      PriorityQueue<Task> taskQueue = new PriorityQueue<>((t1,
t2)->t1.priority().compareTo(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.

==========

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
{
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]

package com.nit.collection;

import java.util.ArrayList;
import java.util.List;

record Dog(String name)
{

}

public class Test4 {

 public static void main(String[] args)
 {
   Dog dog = getListOfDogObject().get(1);
   System.out.println(dog.name());
 }

 public static List<Dog> getListOfDogObject()
 {
  List<Dog> listOfDogs = new ArrayList<>();
  listOfDogs.add(new Dog("A"));
  listOfDogs.add(new Dog("B"));
  listOfDogs.add(new Dog("C"));

  return listOfDogs;

 }
}
```

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 :

```
import java.util.*;

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.*;
public class Test6
{
 public static void main(String[] args)
 {
  List<Integer> myList = new ArrayList<>();
  myList.add(4);
  myList.add(6);
  myList.add(5);

  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 OR
 {
 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.

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

 public static void m1(List<Integer> list)
 {
 }

 public static void m1(List<String> str)
 {
 }
}
```

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

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");
 }
}
```

```
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.

```
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");
 }
}
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
```

21-02-2025

```
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 <?>

Different Cases :

Case 1 :

<Dog>   : Only we can take <Dog> object

<Animal> : Only we can take <Animal> object

<?>   : It is known as unknown type that means we can
            allow everything.[All posibillities]

Now to provide some kind of restriction, java software people has provided two more concepts
1) Upper Bound
2) Lower Bound

Upper Bound :

<? extends Animal> : This is upper bound here we can replace
                      wild-card (?) with any class which

extends Animal but here there is a chance
of wrong collection in the future because
in future Animal can have more sub classes
so, we can't add any element in the Collection.

Lower Bound :

<? super Dog> : This is called lower bound, Here we can replace wild card (?) with with any class which is super of Dog i.e Animal, Object. Here compiler knows that the only classes which are super of Dog are allowed so adding element in the collection is allowed.

```
package com.ravi.wild_card;

import java.util.ArrayList;

class Animal
{

}

class Dog extends Animal
{
}

class Horse extends Animal
{
}

public class WildCardDemo1 {

 public static void main(String[] args)
 {
    ArrayList<? super Dog> list = new ArrayList<Object>();
    list.add(new Dog());


    ArrayList<? extends Animal> list1 = new ArrayList<Horse>();
       list1.add(new Horse()); //error [Not allowed]
 }

}
```

```
import java.util.*;
class Parent
{
}
class Child extends Parent
{
}

public class Test11
{
```

```
public static void main(String [] args)
 {
  //ArrayList<Parent> lp = new ArrayList<Child>(); //error

  ArrayList<?> lp = new ArrayList<Child>();

  ArrayList<Parent> lp1 = new ArrayList<Parent>();

  ArrayList<Child> lp2 = new ArrayList<>();

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

-----------------------------------------------------------
Working in some places :
------------------------
var<?> lp = new ArrayList<Parent>();
lp.add(new Parent());
System.out.println("Wild card....");
-----------------------------------------------------------
//program on wild-card chracter
import java.util.*;
class Parent
{

}
class Child extends Parent
{
}
public class Test12
{
public static void main(String [] args)
 {
  ArrayList<?> lp = new ArrayList<Parent>();
  System.out.println("Wild card....");
 }
}
-----------------------------------------------------------------import java.util.*;
public class Test13
{
 public static void main(String[] args)
 {
  List<? extends Number> list1 = new ArrayList<Long>();

  List<? super String> list2 = new ArrayList<Object>();

  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>
{
 private E element;

 public Basket(E element) // Fruit element = new Apple();
 {
  super();
  this.element = element;
 }

 public E getElement()
 {
  return 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 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);

 }
}
```

//Generic Method

```
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);
  }
 }

}
```

==========