

21-FEB-24

javaravishanker@gmail.com

---

What is a language?

A language is a communication media through which we can communicate with each other.

---

What is a programming language?

A Programming language is an intermediate between the user and computer System.

Every language contains two important aspects :

- 1) Syntax (Rules given by the language)
- 2) Semantics (Meaning OR Structure of the language)

In English language, if we want to make a translation then the syntax is :

Subject + verb + Object

She is a girl. [Valid]

She is a box. [Invalid but still I am following the syntax]

In our programming language also :

```
int x = 12;
```

```
int y = 0;
```

```
int z = x /y;
```

---

In Java programming language we have 2 types of Security :

1) At Compilation level : Here our java compiler will verify whether the code is valid or not according to the syntax.

2) At Runtime Level : Here our runtime environment will verify the semantics of the code that means the code is meaningful or not?

---

Statically(Strongly) 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(Looesly) typed language :-

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 a function :-

---

A function is a self defined block for any general purpose, calculation or printing some data.

The major benefits with function are :-

---

1) Modularity :- Dividing the bigger modules into number of smaller modules where each module will perform its independent task.

2) Easy understanding :- Once we divide the bigger task into number of smaller tasks then it is easy to understand the entire code.

3) Reusability :- We can reuse a particular module so many number of times so It enhances the reusability nature.

Note :- In java we always reuse our classes.

4) Easy Debugging :- Debugging means finding the errors, With function It is easy to find out the errors because each module is independent with another module.

---

Why we pass parameter to a function :-

---

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

Eg:-

userdefined function	predefined function
public void switchOn(int a)	switchOn(3); // The fan is running in mode 3
{	
//start the fan	
}	

---

Why member functions are called method in java ?

---

variable -> fields  
function -> methods

---

Why functions are called method in java?

---

In C++ there is a facility to write a function inside the class as well as outside of the class by using :: (Scope resolution Operator), But in java all the functions must be declared inside the class only.

That is the reason member functions are called method in java.

Variable --> Field  
function ---> Method

---

History of java :

---

First Name of Java : OAK (In the year 1991 which is a tree name)

Project Name :- Green Project

Inventor of Java : - James Gosling and his friends

Official Symbol :- Coffee CUP

Java :- Island (Indonesia)

---

Why java become so popular in the IT Industry ?

---

The role of compiler :

---

- a) Compiler are used to check the syntax.
- b) It also check the compatibility issues(LHS = RHS)
- c) It converts the source code into machine code.

Java code :

---

- a) Java programs must be saved having extension .java
- b) java compiler(javac) is used to compile our code.
- c) After successful compilation we are getting .class file (bytecode)
- d) This .class file we submit to JVM for execution prupose (for executing my java code)

JVM :- It stands for Java Virtual Machine. It is a software in the form of interpreter written in 'C' language.

Every browser contains JVM, Those browsers are known as JEB (Java Enabled Browsers) browsers.

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.

Where as on the other hand 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.

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

---

---

What is the difference between the 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 machine understandable format.

---

---

Comments in JAVA :-

---

Comments are used to increase the readability of the program. It is ignored by the compiler.  
In java we have 3 types of comments

- 1) Single line Comment (//)
- 2) Multiline Comment /\* ----- \*/
- 3) Documentation Comment /\*\* ----- \*/

```
/**  
Name of the Project : Online Shopping  
Date created :- 12-12-2021  
Last Modified - 16-01-2022  
Author :- Ravishankar  
Modules : - 10 Modules  
*/
```

---

---

WAP in java to display welcome message :

---

At initial level we can write a java program with the help class and methods.

Syntax for a class

---

```
[Access Modifier] class [Name of the class]  
{  
}
```

Example :-

```
public class Test  
{  
}
```

Syntax for a method :

---

```
[Access modifier] [return type] [Name of the method] ()
```

```
{
```

```
}
```

Example :-

```
public void accept()  
{  
}
```

---

Note :-

1) In java whenever we write a program we need at least a main method.

2) In java the execution of the program always starts and ends with main method.

---

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 to declare our methods as public.

---

static :-

-----  
In java our main method is static so we need not to create an object to call the main method.

If a method is declared as a static then we need not to create an object to call that method. We can directly call the static methods with the help of class name (Object is not required).

If we don't declare the main method as static method then our program will compile but it will not be executed by JVM.

---

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() :-

---

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.

We can write multiple main methods in our class but argument must be different.

---

---

---

Command Line Argument (Introduction only) :-

---

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

The argument inside the main method is String because String is a alpha-numeric collection of character so, It can accept numbers,decimals, characters, combination of number and character.

That is the reason java software people has provided String as a parameter inside the main method.(More Wider scope to accept the value)

---

System.out.println() :-

---

It is an output statement in java, By using System.out.println() statement we can print anything on the console.

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

In System.out, .(dot) is a member access operator. It is called as period. It is used to access the member of the class.

---

WAP in java to add two numbers :

---

Addition.java

---

public class Addition

```
{  
    public static void main(String[] args)  
    {  
        int val1 = 100;  
        int val2 = 200;  
        int result = val1 + val2;  
        System.out.println(result);  
    }  
}
```

In this program we are getting the result but it is not user-friendly, to provide user-friendly message we should re-write the program

```
public class AdditionWithMessage  
{  
    public static void main(String[] args)  
    {  
        int val1 = 100;  
        int val2 = 200;  
        int result = val1 + val2;  
        System.out.println("Sum is :" + result);  
    }  
}
```

---

//WAP to add two numbers without 3rd variable

```
public class AdditionWithout3rdVariable  
{  
    public static void main(String[] args)  
    {  
        int x = 12;  
        int y = 15;  
        System.out.println("Addition is :" + x+y); //Addition is :1215  
        System.out.println(+x+y); //27  
        System.out.println("Addition is :" +(x+y));  
    }  
}
```

---

18-12-2023

---

Command Line Argument :

---

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

By using command line argument we can pass some value at runtime.

The advantage of Command Line Argument is "Single time Compilation and number of times execution".

Note :- String is a collection of alpha-numeric character so Java software people decided to pass String as a parameter hence, it can accept all different types of values like int, float, double, character and so on.

Note :- If we modify .java file then we should re-compile our java file to generate a new .class file.

```
-----  
//Accepting value from Command Line  
public class Command  
{  
    public static void main(String[] x)  
    {  
        System.out.println(x[0]);  
    }  
}
```

```
javac Command.java  
java Command Virat Rohit Siraj
```

Output :- Virat

-----  
WAP to take different value at the time of running the program

```
//WAP to take different value at the time of running the program
```

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

```
javac DifferentValuesFromCommand.java  
java DifferentValuesFromCommand Rahul Rohit Dhoni  
Output :- Rahul  
java DifferentValuesFromCommand Smiriti Mithali Jhulan  
Output :- Smiriti  
java DifferentValuesFromCommand ABC DEF GHI  
Output :- ABC
```

=====

Eclipse IDE :

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

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

WAP in Eclipse IDE to print full name of user through command Line Argument

```
-----  
package com.ravi.command;  
  
public class FullNameByCommand  
{  
    public static void main(String[] args)  
    {  
        System.out.println(args[0]);  
    }  
}
```

```
}
```

```
java FullNameByCommand.java (Compilation)
java FullNameByCommand Virat Kohli
Output :- Virat
java FullNameByCommand "Virat Kohli"
Output :- Virat Kohli
```

---

WAP to accept integer value from command line Argument :

---

```
package com.ravi.command;

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

```
javac AcceptIntUsingCommand.java
java AcceptIntUsingCommand 12 89
Output : 12
java AcceptIntUsingCommand 12.45 89.78
Output : 12.45
java AcceptIntUsingCommand true false
Output : true
```

From the above it is clear that String parameter in the main method will accept different kinds of values.

---

WAP to add two numbers using Command line Argument :

---

```
package com.ravi.command;

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

```
javac CommandAddition.java
java CommandAddition 123 123
Output : 123123 [Here '+' operator behaves as a String concatenation Operator]
```

---

How to convert String into Integer :

---

If we want to convert any String value into corresponding integer then java software people has provided predefined class called Integer available in java.lang package.

In this class we have predefined static method, parseInt(String x) which is accepting a single parameter String as a parameter and convert this String into int value as shown in the below example.

```
public class Integer
{
    public static int parseInt(String x)
    {
        //Here logic is written to convert x which is String into corresponding int
    }
}
```

Note :- "123" -----> parseInt(String x) -----> 123 [Accepting String and converting into integer]

---

WAP to add two numbers using command line Argument with the help of Integer class

```
package com.ravi.command;
```

```
public class CommandAddition
{
    public static void main(String[] num)
    {

        int a = Integer.parseInt(num[0]);
        int b = Integer.parseInt(num[1]);
        System.out.println("Sum is :" +(a+b));
    }
}
```

```
javac CommandAddition.java
java CommandAddition 123 123
Output 246
java CommandAddition 500 1000
Output 1500
```

---

19-12-2023

---

What is a Package ?

---

A package is nothing but folder in windows or directory in DOS.  
A package is of two types :

1) Predefined Package :- Created by java software people.  
Example :- java.lang, java.util, java.io, java.text, java.awt and so on

2) Userdefined Package :- Created by user.  
Example :- com.ravi.basic, com.ravi.introduction, com.ravi.oop and so on

The program which contains package statement is different style of compiling :

```
javac -d . Test.java  
(javac space -d space . space FileName.java)
```

Program :

```
-----  
package com.tcs.shopping;  
  
public class Ravi  
{  
  
}
```

```
javac -d . Ravi.java  
3 folders will be created and then the .class file will be placed  
inside the folder.
```

-----  
Program which describes how Integer.parseInt(String str) is working :

File Name :  
Main.java

```
-----  
package com.ravi.basic;  
  
public class Main  
{  
    public static void main(String[] args)  
    {  
        int result = Calculate.doSum(12, 36);  
        System.out.println("Addition of two number :" + result);  
  
        result = Calculate.getSquare(12);  
        System.out.println("Square is :" + result);  
  
        result = Calculate.getCube(2);  
        System.out.println("Cube is :" + result);  
    }  
}  
  
class Calculate  
{  
    public static int doSum(int x, int y)  
    {  
        int z = x + y;  
        return z;  
    }  
  
    public static int getSquare(int a)  
    {  
        int b = a * a;  
        return b;  
    }  
}
```

```
public static int getCube(int p)
{
    return (p*p*p);
}
```

---

\*Naming convention in Java :

---

#### 1) How to write a class in java

---

While writing a class in java we should follow pascal naming convention. A java class represents noun.

ThisIsExampleOfClass (Each word first letter is capital)

Example :

---

```
String
System
Integer
BufferedReader
DataInputStream
ClassNotFoundException
ArithmaticException
```

#### 2) How to write a method in java :

---

In order to write methods in java we need to follow camel case naming convention. A java method represents verb.

thisIsExampleOfMethod()

Example:

---

```
read()
readLine()
toUpperCase()
charAt()
```

#### 3) How to write variable(Fields) in java

---

In order to write variables in java we need to follow camel case naming convention.

```
rollNumber;
employeeName;
customerNumber;
customerBill;
```

#### 4) How to write final variable(Field)

---

```
final double PI = 3.14;
final int A = 90;
```

#### 5) How to write final and static variable

```
MAX_VALUE;  
MIN_VALUE;
```

Each character must be capital and in between every word \_ symbol should be there.

## 6) How to write package

Package name must be in small character. It is reverse of company name.  
com.ravi.basic;

---

Token :

A token is the smallest unit of the program that is identified by the compiler.

Every Java statements and expressions are created using tokens.

A token can be divided into 5 types

- 1) Keywords
- 2) Identifiers
- 3) Literals
- 4) Punctuators
- 5) Operators

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.

Identifiers :

A name in java program by default considered as identifiers.

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

We can't use keyword as an identifier.

Ex:-

```
class Fan  
{  
    int coil ;  
  
    void start()  
    {  
    }  
}
```

Here Fan(Name of the class), coil (Name of the variable) and start(Name of the function) 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 :-

---

Assigning some constant value to variable is called Literal.

Java supports 5 types of Literals :

- 1) Integral Literal Ex:- int x = 15;
- 2) Floating Point Literal Ex:- float x = 3.5f;
- 3) Character Literal Ex:- char ch = 'A';
- 4) Boolean Literal Ex:- boolean b = true;
- 5) String Literal Ex:- String x = "Naresh i Technology";

Note :- null is also a literal.

---

20-12-2023

---

Integral Literal :

---

If any numeric literal does not contain decimal or fraction then it is called Integral Literal.  
Example :- 67, 89, 234.

In Integral literal we have 4 data types

byte (8 bits)  
short (16 bits)  
int (32 bits)  
long (64 bits)

An integral literal we can specify or represent in different ways

- a) Decimal literal (Base 10)
- b) Octal literal (Base 8)
- c) Hexadecimal literal (Base 16)
- d) Binary Literal (Base 2) (Available from JDK 1.7 onwards)

Note :- As a developer we can represent an integral literal in different forms(decimal, octal, hexadecimal and binary) but JVM always produces the output in decimal form only.

## Decimal Literal :-

---

The base of decimal literal is 10. we can accept any digit from 0-9

## Octal Literal :-

---

The base is 8. Here we can accept digits from 0-7 only. In java if any integral literal prefix with '0' (Zero) then it becomes octal literal.

### Example:-

```
int x = 015; //Valid  
int y = 018;//Invalid [Digit '8' is out of the range]
```

## Hexadecimal Literal :-

---

The base is 16. Here we can accept digits from 0-15 (0-9 and A-F). In java if any integral literal prefix with 0X or 0x (zero with capital X OR zero with small x) then it becomes hexadecimal literal.

### Example :-

```
int x = 0X15; //Valid  
int y = 0x14;//Valid  
int z = 0Xadd; //Valid  
int a = 0Xage; //Invalid ['g' is out of range]
```

## Binary Literal :-

---

It is introduced from jdk 1.7 onwards. The base or radix is 2. Here we can accept digits 0 and 1 only. In java if any integral Literal prefix with 0B or 0b (zero capital B or 0 small b) then it becomes binary literal.

### Example :-

```
int x = 0B111; //Valid  
int y = 0b101010; //Valid  
int z = 0B12; //Invalid [digit 2 is out of range]
```

```
//Octal literal  
public class Test1  
{  
    public static void main(String[] args)  
    {  
        int one = 01;  
        int six = 06;  
        int seven = 07;  
        int eight = 010;  
        int nine = 011;  
        System.out.println("Octal 01 = "+one);  
        System.out.println("Octal 06 = "+six);  
        System.out.println("Octal 07 = "+seven);  
        System.out.println("Octal 010 = "+eight);  
        System.out.println("Octal 011 = "+nine);  
    }  
}
```

```
}
```

---

```
//Hexadecimal
public class Test2
{
    public static void main(String[] args)
    {
        int i = 0x10; //16
        int j = 0Xadd; //2781
        System.out.println(i);
        System.out.println(j);
    }
}
```

---

```
//Binary Literal
public class Test3
{
    public static void main(String[] args)
    {
        int i = 0b101;
        int j = 0B111;
        System.out.println(i); //5
        System.out.println(j); //7
    }
}
```

---

By default every integral literal is of type int only but we can specify explicitly as long type by suffixing with L (small L) OR L (Capital L).

According to industry standard L is more preferable because l (small l) looks like 1(digit 1).

There is no direct way to specify byte and short literals explicitly. If we assign any integral literal to byte variable and if the value is within the range (-128 to 127) then it is automatically treated as byte literals because compiler internally converts from int to byte.

If we assign integral literals to short and if the value is within the range (-32768 to 32767) then automatically it is treated as short literals because compiler internally converts from int to short.

---

21-12-2023

---

```
/* 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;
        System.out.println(s);
    }
}

-----
```

```

//Converting bigger type to smaller type
public class Test6
{
    public static void main(String[] args)
    {
        short s = 128;
        byte b = (byte) s; //Explicit type casting [Chance of data loss]
        System.out.println(b);
    }
}

-----
```

```

public class Test7
{
    public static void main(String[] args)
    {
        byte x = (byte) 127L;
        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 language. In fact any language which accepts the primary data type like int, float, char is not a pure object oriented language hence java is also not a pure object oriented language.

Example of pure object oriented language : Ruby, smalltalk and so on

If we remove all 8 primitive data types from java then Java will become pure object oriented language.

In java we have a concept called Wrapper classes through which we can convert the primary data types into corresponding Wrapper Object.(Autoboxing 1.5V)

Primary data types	Corresponding Wrapper Object
byte	- Byte
short	- Short

int	-	Integer
long	-	Long
float	-	Float
double	-	Double
char	-	Character
boolean	-	Boolean

All these wrapper classes are available in java.lang package.

---

```
//Wrapper classes
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);
    }
}
```

---

How to know the minimum and maximum value as well as size of integral literal data types:

---

These classes (Wrapper classes) are providing the static and final variables through which we can find out the minimum, maximum value as well as size of the data types

Ex:- I want to find out the range and size of Byte class

Byte.MIN\_VALUE = -128

Byte.MAX\_VALUE = 127

Byte.SIZE = 8 (in bits format)

Here MIN\_VALUE, MAX\_VALUE and SIZE these are static and final variables available in these classes(Byte, Short, Integer and Long).

---

```
//Program to find out the range and size of Integral 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);

    }

}

-----
```

```

//We can provide _ in integral literal
public class Test10
{
    public static void main(String[] args)
    {
        long mobile = 9812_3456_78L;
        System.out.println("Mobile Number is :" + mobile);
    }
}
```

Note :- From java 1.7V now we can provide \_ symbol in numeric literal just to increase the readability of the code.

```

public class Test11
{
    public static void main(String[] args)
    {
        final int x = 127;
        byte b = x;
        System.out.println(b);
    }
}
```

The above program will generate the output.

---

22-12-2023

---

```

// Converting from decimal to another number system
public class Test12
{
    public static void main(String[] argv)
    {
        //decimal to Binary
        System.out.println(Integer.toBinaryString(5)); //101

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

        //decimal to Hexadecimal
    }
}
```

```
        System.out.println(Integer.toHexString(2781)); //add
    }
}

//var keyword [Introduced from java 10]
public class Test13
{
    public static void main(String[] args)
    {
        var x = 12;
        System.out.println(x);
    }
}
```

floating point literal :

1) Any numeric literal, if contains decimal or fraction then it is called floating point literal.

Example :- 67.89, 78.23, 90.90

2) In floating point literal we have 2 data types :

- a) float (32 bits)
- b) double (64 bits)

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

```
float f = 2.3; //Invalid
```

Now, we have 3 solutions

```
float f1 = 2.3f;
float f2 = 2.3F;
float f3 = (float) 7.8;
```

4) Even though, every floating point literal is of type double only but in order to enhance the readability of the code, java software people has provided two flavors to represent double literal explicitly.

```
double d1 = 2.3D;
double d2 = 2.5d;
```

\*5) While working with integral literal, we can represent integral literal in 4 different forms i.e. decimal, octal, hexadecimal and binary but in floating point literal we have only one form i.e decimal.

\*6) We can assign integral literal to floating point literal but floating point literal we cannot assign to integral literal.

7) We can represent floating point literal in exponent form.

Example :- 15e2  
15 X 10 X 10 (10 to the power 2)

---

```
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 a = 1.0;  
        float b = 15.29F;  
        float c = 15.25f;  
        float d = (float) 15.25;  
  
        System.out.println(b+" : "+c+" : "+d);  
    }  
}  
-----  
public class Test2  
{  
    public static void main(String[] args)  
    {  
        double d = 15.15;  
        double e = 15.15d;  
        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; //error  
  
        System.out.println(x+","+y+","+z);  
    }  
}  
-----  
class Test4  
{  
    public static void main(String[] args)  
    {  
        double x = 0X29;  
  
        double y = 0X9.15; //error
```

```
        System.out.println(x+"," +y);
    }
}

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

        double d2 = 15e3;
        System.out.println("d2 value is :" +d2);

    }
}

public class Test6
{
    public static void main(String[] args)
    {
        double a = 0791;

        double b = 0791.0;

        double c = 0777;

        double d = 0Xdead;

        double e = 0Xdead.0;
    }
}

public class Test7
{
    public static void main(String[] args)
    {
        double a = 1.5e3;
        float b = 1.5e3; //E
        float c = 1.5e3F;
        double d = 10;
        int e = 10.0; //E
        long f = 10D; //E
        int g = 10F; //E
        long l = 12.78F; //E
    }
}

//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);
    }
}

```

---

**Boolean Literal :**

---

It is used to represent two states either true or 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 also depends on JVM implementation.

Unlike C and C++, we cannot assign 0 and 1 to boolean data type because in java any number without decimal is treated as integral literal

```
boolean isEmpty = 0; [Invalid in Java but valid in C and C++]
```

We cannot assign String to boolean type as shown below.

```

boolean isValid = "true"; //Invalid

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

        System.out.println(isValid);
        System.out.println(isEmpty);
    }
}
```

---

```

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

---

```

public class Test3
{
    public static void main(String[] args)
```

```
{  
    boolean x = "true"; //error  
    boolean y = "false"; //error  
    System.out.println(x);  
    System.out.println(y);  
}  
}
```

---

Character Literal :

---

It is also known as char literal.

Here we have only one data type i.e char data type which accepts 2 bytes of memory.

We can represent char literal in different ways which are as follows :

a) Single character enclosed with single quotes.

Example :- char ch = 'A';

b) We can assign integral literal to char data type to represent the UNICODE value for that character.

In older language like C and C++, the range is 0 - 255 which is representing the ASCII value of the character.

But Java supports UNICODE where the range is 0 - 65535.

```
char ch = 65535; //Valid  
char ch = 65536; //InValid
```

c) The char literal we can assign to integeral data type to know the UNICODE number value of that character.

```
int ch = 'A'; -> 65
```

d) All our escape sequnces we can represent as char literal.

```
char ch = '\n';
```

e) We can represent char literal in 4 digit hexadecimal number to represent UNICODE value where the format is :

```
'\uXXXX'
```

Here \u stands for UNICODE where as X represents Digits.

---

05-03-2024

---

```
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);

    }

}

class Test2
{
    public static void main(String[] args)
    {
        int ch = 'A';
        System.out.println("ch value is :" + ch);
    }
}

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

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

        char ch3 = 65;
        System.out.println("ch3 value is :" + ch3);
    }
}

}

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

        char ch2 = 0Xadd;
        System.out.println("ch2 value is :" + ch2);
    }
}

```

Here we will get ? as an output because the language translator is not available.

---

```

//Addition of two character in the form of Integer
public class Test5
{
    public static void main(String txt[])
    {
        int x = 'A';
        int y = 'B';
        System.out.println(x+y); //131
        System.out.println('A' + 'A'); //130
    }
}

```

```
        }
    }

-----
```

```
//Range of UNICODE Value (65535)
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
class Test7
{
    public static void main(String[] args)
    {
        char ch1 = '\u0002';
        System.out.println(ch1);

        char ch2 = '\uffff';
        System.out.println(ch2);

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

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

-----
```

```
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(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
    }
}
```

Java software people has not provided the support for MIN\_VALUE  
and MAX\_VALUE for Character class

---

```
//Java Unicodes
public class Test12
{
    public static void main(String[] args)
    {
        System.out.println(" Java Unicodes\n");

        for (int i = 31; i < 126; i++)
        {
            char ch = (char)i; // Convert unicode to character
            String str = i + " " + ch;

            System.out.print(str + "\t\t");
            if ((i % 5) == 0) // Set 5 numbers per row
                System.out.println();
        }
    }
}
```

---

String Literal :-

---

A string literal in Java is basically a sequence of characters. These characters can be anything like alphabets, numbers or symbols which are enclosed with double quotes. So we can say String is alpha-numeric collection of character.

---

How we can create String in Java :-

---

In java String can be created by using 3 ways :-

1) By using String Literal

```
String x = "Ravi";
```

2) By using new keyword

```
String y = new String("Hyderabad");
```

3) By using character array

```
char z[] = {'H','E','L','L','O'};
```

---

```
//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) Separators :

---

It is used to inform the compiler that how the things will be group together.

We have following separators :

( ), {}, [], ; , ...

---

#### 5) Operator :

---

It is symbol through which we will understand that how the calculation will be performed on the operands.

Java supports the following operators :

- 
- 1) Arithmetic Operator OR Binary Operator
  - 2) Unary Operator
  - 3) Assignment Operator
  - 4) Relational Operator
  - 5) Logical Operator (&& || !)
  - 6) Boolean Operatot (& |)
  - 7) Bitwise Operator (~)
  - 8) Ternary Operator
  - \*9) new Operator
  - \*10) Dot Operator (. [Member Access Operator])
  - \*11) instanceof operator
- 

How to read the data from Client with user-friendly approach :

---

By using command line Argument, we can't provide user-friendly message to our user to accept the data.

Java software people introduced a predefined class called Scanner from java 1.5v available in java.util pacakge.

Static variable of System class :

---

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

- 1) System.out :- It is used to print the normal message on the screen.
- 2) System.err :- It is used to print error message on the screen in red color.
- 3) System.in : It is used to accept the input from the source.

Creating Object for Scanner class :

```
Scanner sc = new Scanner(System.in);
```

06-03-2024

-----  
Methods Of Scanner class :

- 1) public byte nextByte() :- Used to read byte data.
- 2) public short nextShort() :- Used to read short data
- 3) public int nextInt() :- Used to read int data.
- 4) public long nextLong() :- Used to read long data.
- 5) public float nextFloat() :- Used to read float data.
- 6) public double nextDouble() :- Used to read double data.
- 7) public boolean nextBoolean() :- Used to read boolean data.
- 8) public String nextLine() :- Used to read complete line, the  
return type of this method is String
- 9) public char next().charAt(0) :- Used to read character data.

Note :- next() method is used to read a word from the client in the  
String format.

-----  
//Program to read the name from the keyboard

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

-----  
//Program to read a character (gender [M/F]) from the client

```
import java.util.Scanner;  
  
public class ReadGender  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter your Gender [M/F] :");  
    }  
}
```

```
char gen = sc.next().charAt(0); //Method Chaining
System.out.println("Your Gender is :" + gen);
}
-----
```

//WAP to read the employee id and name from Scanner class :

```
-----
```

```
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 eid = sc.nextInt();

        System.out.print("Enter Employee Name:");
        String ename = sc.nextLine(); //buffer problem
        ename = sc.nextLine();

        System.out.println("Employee id is :" + eid);
        System.out.println("Employee name is :" + ename);
    }
}
```

---

Some operator related task :

```
-----
```

Whenever we work with Arithmetic operator OR unary minus then after the execution of expression the result will be promoted to int type.  
Here minimum int data type is required.

```
public class Test
{
    public static void main(String[] args)
    {
        short s = 2;
        short t = 1;
        short u = s + t; //error
        System.out.println(u);
    }
}
```

---

The following program will compile and executed.

```
public class Test
{
    public static void main(String[] args)
    {
        short s = 2;
        short t = 1;
        s += t; //s = s + t [Short hand operator]
        System.out.println(s);
    }
}
```

```
-----  
public class Test  
{  
    public static void main(String[] args)  
    {  
        byte b = 2;  
        b = -b; //error  
        System.out.println(b);  
    }  
}  
-----
```

```
-----  
public class Test  
{  
    public static void main(String[] args)  
    {  
        int x = 40;  
        int y = 40++; //error  
        System.out.println(y);  
    }  
}
```

What is the difference between logical operator and Boolean Operator

Logical Operator :

&&

||

!

Boolean Operator :

& : Boolean AND -> Even first expression is false but still it will evaluate all the right side expression.

| : Boolean OR -> Even first expression is true but still it will evaluate all the right side expression.

```
-----  
public class Test  
{  
    public static void main(String[] args)  
    {  
        int x = 5;  
  
        if(++x > 6 & ++x > 7) //Boolean AND  
        {  
            x++;  
        }  
        System.out.println(x);  
    }  
}
```

```
-----  
public class Test  
{
```

```
public static void main(String[] args)
{
    System.out.println(6&7); //6
    System.out.println(6|7); //7
    System.out.println(6^7); //1    //Exclusive OR
}
}
```

From the above program Diagram is available for output (06-Mar)

---

07-03-2024

---

What is a local variable in java ?

---

If we declare a variable inside the method body but not as a method parameter then it is called local/temporary/Automatic/Stack variable.

```
public class Test
{
    public static void main(String[] args)
    {
        int x = 10; //Local Variable
        System.out.println(x);
    }
}
```

A local variable must be initialized before use.

A local variable will not accept any access modifier (public, private, protected and so on) except final.

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

As far as its scope is concerned, It is accessible within the same method body only that means we can't access local variable outside of the method.

---

Why we can't access local variable outside of the method ?

---

In java all the methods are executed as a part of Stack memory on the basis of LIFO (Last In First out).

Whenever we call a method then for the execution of the method, one stack frame will be created and once the method execution is over then Stack frame will be deleted.

```
package com.ravi.method_execution;

public class StackExecution
{
    public static void main(String[] args)
    {
        System.out.println("Main method Started");
        m1();
        System.out.println("Main method Ended");

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

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

---

Control Statements in Java :

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 :-

In switch case depending upon the parameter the appropriate case would be executed otherwise default would be executed.

In this approach we need not to check each and every case, if the appropriate case is available then directly it would be executed.

break keyword is optional here but we can use as per requirement. It will move the control outside of the body of the switch.

---

```
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.nextLine().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.nextLine().toLowerCase();

switch(season)
{
    case "summer" :
        System.out.println("It is summer Season!!!");
        break;

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

```

Note :- Strings are allowed in the switch case from java 1.7 and enums are allowed from java 1.5v

---

```

public class Test
{
    public static void main(String[] args)
    {
        float choice = 100;
        switch(choice)
        {
            case 100:
                System.out.println("Police Services");
                break;
        }
    }
}

```

In switch expression we can't pass long, float and double.

---

In the label of switch case we can pass final variable as shown below.

```

public class Test
{
    public static void main(String[] args)
    {
        final int x = 100;
        switch(x)
        {
            case x: //valid
                System.out.println("Police Services");
                break;
        }
    }
}

```

```
}
```

---

```
public class Test
{
    public static void main(String[] args)
    {
        byte b = 100;
        switch(b)
        {
            case 100:
                System.out.println("Police Services");
                break;

            case 128: //error 128 is out of the range
                System.out.println("Training Services");
                break;
        }
    }
}
```

---

Loops in Java :

---

Java supports 4 types of loop :

---

- 1) do while loop
- 2) while loop
- 3) for loop
- 4) for each loop (java 1.5V)

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

        do
        {
            int x = 1; // x is a block level variable
            System.out.println(x);
            x++;
        }
        while (x<=10); //error

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

---

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

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

---

08-03-2024

---

For Each loop :

---

It was introduced from JDK 1.5 onwards.

The main purpose of for each loop to fetch OR retrieve the data from Collection.

Each value of collection will assign to ordinary variable to satisfy the loop.

```
package com.ravi.loop;

public class ForEachDemo {

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

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

---

In Java, Array can hold heterogeneous types of data as shown in the program

```
package com.ravi.loop;

public class ForEachDemo1
{
    public static void main(String[] args)
    {
        Object []arr = {12,"Ravi",true,23.90};

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

---

Program that describes how to sort an array of integers in ascending order.

How to sort the array in ascending order :

There is a predefined class called Arrays available in java.util package, This class contains a predefined static method sort(int []x) through which we can sort the Integer array in ascending order.

```
public class Arrays
{
    public static void sort(int []arr)
    {
        //Convert this array into ascending order
    }
}
```

ForEachDemo2.java

```
package com.ravi.loop;

import java.util.Arrays;

public class ForEachDemo2
{
    public static void main(String[] args)
    {
        int []values = {92,78,45,12,68};

        Arrays.sort(values);

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

```
String []fruits = {"Mango","Orange","Apple","Kiwi"};
```

```
Arrays.sort(fruits);

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

Working with Method return type and Parameter :

What is BLC and ELC class :

BLC : It stands for Business Logic class. In this class we should write all the business logic, this class will never contain main method.

**ELC** : It stands for Executable Logic class. In this class we should not write any logic and this class must contain main method because the execution of the program will start from main method i.e ELC class.

Note :- IN A JAVA FILE WE CAN'T WRITE MULTIPLE PUBLIC CLASSES THAT MEANS IN A SINGLE JAVA FILE WE MUST HAVE ONLY ONE PUBLIC CLASS.

---

Write a program to find out the square of the number.

2 files :

---

FindSquare.java(BLC)

---

```
package com.ravi.method_parameter;

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

---

Main.java (ELC)

---

```
package com.ravi.method_parameter;

import java.util.Scanner;

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

---

Write a program to find out the Area of the Circle.

2 files :

---

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

---

Circle.java(BLC)

---

```
package com.ravi.pack7;
public class Circle
```

```

{
    public static String getAreaOfCircle(double rad)
    {
        if(rad <0 || rad ==0)
        {
            return ""+(-1);
        }
        else
        {
            final double PI = 3.14;
            double areaOfCircle = PI * rad * rad;
            return ""+areaOfCircle;
        }
    }
}

```

Test7.java(ELC)

---

```

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 rad = sc.nextDouble();

        String circle = Circle.getAreaOfCircle(rad);

        //Converting String into double
        double areaOfCircle = Double.parseDouble(circle);

        DecimalFormat df = new DecimalFormat("00.000");
        System.out.println("Area of circle is :" +df.format(areaOfCircle));
        sc.close();
    }
}

```

How to format the floating point literal :

---

If we want to provide our own formatting in floating point literal then we should use a predefined class called DecimalFormat available in java.text package.

```

DecimalFormat df = new DecimalFormat("00.00"); //00.00 is the format
df.format(double d);

```

Now, this DecimalFormat contains a predefined method called format(double d) which is accepting double as a parameter which will print the floating point literal in specified format.

---

09-03-2024

---

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 || 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;  
  
//ELC  
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 val = Calculate.getSquareAndCube(num);
```

```
        System.out.println("Value is :" +val);
        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 Rectangle :");  
        double length = sc.nextDouble();  
        System.out.print("Enter the breadth of the Rectangle :");  
        double breadth = sc.nextDouble();  
  
        double areaOfRectangle = Rectangle.getAreaOfRectangle(length, breadth);  
        System.out.println("Area of rectangle is :" +areaOfRectangle);  
        sc.close();  
    }  
}  
-----  
2 Files :  
-----  
EvenOrOdd.java  
-----  
package com.ravi.pack6;  
  
//BLC  
public class EvenOrOdd  
{  
    public static boolean isEven(int num)  
    {
```

```
        return (num % 2 == 0);
    }
}

Test6.java
-----
package com.ravi.pack6;

//ELC
public class Test6
{
    public static void main(String[] args)
    {
        boolean val = EvenOrOdd.isEven(4);
        System.out.println("4 is even :" +val);

        val = EvenOrOdd.isEven(5);
        System.out.println("5 is even :" +val);
    }
}
```

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+"]";
    }
}
```

Test.java

```
-----  
package com.ravi.pack8;

public class Test8
{
    public static void main(String[] args)
    {
        String details = Student.getStudentDetails(111, "Raj", 18000.90);
        System.out.println("Student Details are :" +details);
    }
}
```

-----  
2 Files :

-----  
Table.java  
-----

```
package com.ravi.pack9;

//BLC
public class Table
{
    public static void printTable(int num) //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<=5; i++)
        {
            Table.printTable(i);
        }
    }
}
```

-----  
Types of variable in java :

-----  
According to data type the variables are divided into 2 types :

1) Primitive variables :

In java we have 8 primitive data types, by using those data types if we define a variable then it is called Primitive variable.

Example :- int x = 10;  
          boolean b = false;

2) Non primitive OR reference variable :

If any variable is declared using predefined/user-defined class name then it is called reference variable.

Example :- Test t = new Test(); // t is a reference variable

Based on the declaration position, now the variables are further divided into 4 categories :

- 1) Class Variable OR Static Field (Object is not required)
- 2) Instance Variable OR Non static Field (Object is required)
- 3) Local Variable
- 4) Parameter variable

Class Variable OR Static Field :

---

If a variable is declared with static keyword inside a class but outside of a method then it is called static field.

In order to access static field object is not required.

2) Instance Variable OR Non static Field

---

If a variable is declared without static keyword inside a class but outside of a method then it is called Instance variable OR non static field.

In order to access the non static field, Object is required.

Programs :

---

1) Program on Primitive variable :

```
package com.ravi.variable_type;

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

    static int y = 200; //static field

    public static void main(String[] args)
    {
        PrimitiveTypes p = new PrimitiveTypes();
        System.out.println("Non Static Field :" + p.x);

        System.out.println("Static Field :" + y);
        m1(300);
        int d = 400; //local variable
        System.out.println("Local Variable :" + d);
    }

    public static void m1(int c) //c is parameter variable
    {
        System.out.println("Parameter Variable :" + c);
    }
}
```

---

Program on Reference variable :

---

```
package com.ravi.variable_type;

import java.util.Scanner;

public class ReferenceVariable
{
    Scanner sc = new Scanner(System.in); //Non static field

    static ReferenceVariable r = new ReferenceVariable(); //Static field

    public static void main(String[] args)
    {
        Test t1 = new Test(); //t1 is local variable
        getData(t1);
    }

    public static void getData(Test t) //t is parameter variable
    {
        t.m1();
    }
}

class Test
{
    public void m1()
    {
        System.out.println("m1 non static method");
    }
}
```

---

11-03-2024

---

Object Oriented Programming (OOPs)

---

Object Oriented Programming (OOPs) :

---

What is an Object?

---

An object is a physical entity which exists in the real world.

Example :- Pen, Car, Laptop, Mouse, Fan and so on

An Object has 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 OOPs :**

---

- 1) Modularity (Dividing the bigger task into number of smaller task)
- 2) Reusability (We can reuse the classes as per requirement)
- 3) Flexibility (Easy to maintain)

**Features Of OOPs :**

---

There are 6 features of OOPs

- 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.**

---

**Write an Object oriented Program in Java to define student data and Student behavior.**

**2 Files :**

---

**Student.java**

---

```
package com.ravi.oop;

//BLC
public class Student
{
    //Object Properties (Instance Variable OR Non Static field)
    String name;
    String regNo;
    int age;

    public void talk()
    {
```

```
        System.out.println("Hello Friends, My Name is :" + name + " my registration number is  
        :" + regNo + " and my age is :" + age);  
    }  
  
    public void writeExam()  
    {  
        System.out.println("Every week we have an Exam on saturday");  
    }  
  
}
```

### StudentDemo.java

---

```
-----  
package com.ravi.oop;  
  
//ELC  
public class StudentDemo  
{  
    public static void main(String[] args)  
    {  
        Student raj = new Student();  
        //Initialize the properties  
        raj.name = "Raj Gourav";  
        raj.regNo = "NIT024001";  
        raj.age = 21;  
        raj.talk();  
        raj.writeExam();  
  
        System.out.println(".....");  
  
        Student priya = new Student();  
        priya.name = "Priya Jain";  
        priya.regNo = "NIT024002";  
        priya.age = 21;  
        priya.talk();  
        priya.writeExam();  
    }  
}
```

This Program contains one diagram (11-Mar-24)

---

Write a Program to initialize the object properties through  
Method :

2 files :

---

Employee.java

---

```
-----  
package com.ravi.oop;
```

```
//BLC
```

```
public class Employee
{
    int eid;
    String ename;
    double salary;

    public void setEmployeeData()
    {
        eid = 1;
        ename = "Scott";
        salary = 50000;
    }

    public void getEmployeeInfo()
    {
        System.out.println("Employee Id is :" + eid);
        System.out.println("Employee name is :" + ename);
        System.out.println("Employee salary is :" + salary);
    }
}
```

EmployeeDemo.java

---

```
package com.ravi.oop;

//ELC
public class EmployeeDemo
{
    public static void main(String[] args)
    {
        Employee scott = new Employee();
        scott.setEmployeeData();
        scott.getEmployeeInfo();
    }
}
```

---

From the last two programs, It is clear that we can initialize the object properties through reference variable and by using methods also.

---

12-03-2024

---

Default constructor added by the compiler

---

In java whenever we write a class and if we don't write any type of constructor then automatically compiler will add default constructor to the class.

---

```
class Test
{
    //Here in this class we don't have constructor
}
```

javac Test.java (At the time of compilation automatically compiler will add default constructor)

```
class Test
```

```
{  
    Test() //default constructor added by the compiler  
}  
}
```

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

The access modifier of default constructor added by the compiler depends upon class access modifier.

---

Why compiler adds default constructor to our class :

---

If the compiler does not add default constructor to our class then object creation is not possible in java. At the time of object creation by using new keyword we depend upon the constructor.

The main purpose of defualt constructor(added by the compiler) to initialize the instance variables of the class with some default values.

The default values are:

```
byte - 0  
short - 0  
int - 0  
long - 0  
float - 0.0  
double - 0.0  
char - (Space)  
boolean - false  
String - null  
Object - null
```

Note :- For any reference variable the default value will be null.

---

WAP to show that default values are provided by the constructor

---

2 Files :

---

Employee.java

---

```
package com.ravi.oop_demo;  
  
//BLC  
public class Employee  
{  
    int employeeNumber;  
    String employeeName;  
  
    public void showEmployeeData()  
    {  
        System.out.println("Employee Number is :" + employeeNumber);  
        System.out.println("Employee Name is :" + employeeName);  
    }  
}
```

EmployeeDemo.java

---

```
package com.ravi.oop_demo;  
  
//ELC  
public class EmployeeDemo  
{  
    public static void main(String[] args)  
    {  
        Employee raj = new Employee();  
        raj.showEmployeeData();  
    }  
}
```

---

How to provide our user-defined values for the instance variable :

---

The default values provided by compiler are not useful for the user, hence user will take a separate method (`setEmployeeData()`) to re-initialize (already initialized by default constructor at the time of object creation) the instance variable value so the user will get its own userdefined values.

---

2 files :

---

Employee.java

---

```
package com.ravi.oop_demo;  
  
//BLC  
public class Employee  
{  
    int employeeNumber;  
    String employeeName;  
  
    public void setEmployeeData()  
    {  
        employeeNumber = 111;  
        employeeName = "Ravi";  
    }  
  
    public void showEmployeeData()  
    {  
        System.out.println("Employee Number is :" + employeeNumber);  
        System.out.println("Employee Name is :" + employeeName);  
    }  
}
```

EmployeeDemo.java

---

```
package com.ravi.oop_demo;  
  
//ELC  
public class EmployeeDemo  
{  
    public static void main(String[] args)
```

```
{  
    Employee raj = new Employee();  
    raj.showEmployeeData();  
    System.out.println(".....");  
    raj.setEmployeeData(); //0 - 111 null - Ravi  
    raj.showEmployeeData();  
  
}  
}
```

13-03-2024

-----  
How to re-initialize the variable by accepting the value from the user?

2 files :

-----  
Product .java

```
package com.nit.oop;  
  
import java.util.Scanner;  
  
//BLC  
public class Product  
{  
    int productId;  
    String productName;  
  
    public void setProductData()  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter Product Id :");  
        int productId = sc.nextInt();  
  
        System.out.print("Enter Product Name :");  
        String productName = sc.nextLine();  
        productName = sc.nextLine();  
        sc.close();  
    }  
  
    public void getProductInformation()  
    {  
        System.out.println("Product Id is :" + productId);  
        System.out.println("Product Name is :" + productName);  
    }  
}
```

-----  
ProductDemo.java

```
package com.nit.oop;
```

```
public class ProductDemo {  
    public static void main(String[] args)  
    {  
        Product p1 = new Product();  
        p1.setProductData();  
        p1.getProductInformation();  
    }  
}
```

In the above program we are initializing the local variable through user input but those local variables we can't use outside of the method hence, Still we will get default value i.e. 0 and null.

---

2 files :

---

Product .java

---

```
package com.nit.oop;  
  
import java.util.Scanner;  
  
//BLC  
public class Product  
{  
    int productId;  
    String productName;  
  
    public void setProductData()  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter Product Id :");  
        productId = sc.nextInt();  
  
        System.out.print("Enter Product Name :");  
        productName = sc.nextLine();  
        productName = sc.nextLine();  
        sc.close();  
    }  
  
    public void getProductInformation()  
    {  
        System.out.println("Product Id is :" + productId);  
        System.out.println("Product Name is :" + productName);  
    }  
}  
  
ProductDemo.java
```

---

```
package com.nit.oop;
```

```
public class ProductDemo {  
    public static void main(String[] args)  
    {  
        Product p1 = new Product();  
        p1.setProductData();  
        p1.getProductInformation();  
    }  
}
```

In the setProductData() method we are initializing the instance variable through user input.

---

What is an Instance variable ?

---

A non static variable which is defined inside the class but outside of the method is called instance variable Or non static field.

An instance variable will get the memory as well as initialized at the time of object creation.

**WITHOUT AN OBJECT WE CAN'T THINK ABOUT INSTANCE VARIABLE.**

As far as it's accessibility is concerned, it can be accessible anywhere within the class as well as depends upon the access modifier applicable on the instance variable.

What is local variable ?

---

A local variable is declared inside a method body.

It must be initialized by the user (no default value) before use.

We can access local variable within the same method only but not outside of the method.

---

What is parameter variable ?

---

The variable which we are defining in the method parameter are called Parameter variables.

These variables are mainly used to receive the value from the outer world (End client).

---

2 files :

---

Manager.java

---

```
package com.ravi.parameter;  
//BLC  
public class Manager  
{  
    int managerId;  
    String managerName;  
  
    public void setManagerData(int id, String name)  
    {
```

```
        managerId = id;
        managerName = name;
    }

    public String getManagerData()
    {
        return "[Manager id is :" +managerId+", Manager Name is :" +managerName+"]";
    }

}
```

### ManagerDemo.java

---

```
package com.ravi.parameter;

//ELC
public class ManagerDemo
{
    public static void main(String[] args)
    {
        Manager m = new Manager();
        m.setManagerData(111, "Scott");
        System.out.println(m.getManagerData());

    }

}
```

---

//WAP to initialize the employee grade based on the employee salary.

2 files :

---

### Employee.java

---

```
package com.ravi.oop_ex;

//BLC
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 getEmployeeData()
    {
        System.out.println("Id is :" +employeeId);
```

```

        System.out.println("Name is :" + employeeName);
        System.out.println("Salary is :" + employeeSalary);
        System.out.println("Grade is :" + employeeGrade);
    }

    public void calculateEmployeeGrade()
    {
        if(employeeSalary > 100000)
        {
            employeeGrade = 'A';
        }
        else if(employeeSalary > 90000)
        {
            employeeGrade = 'B';
        }
        else if(employeeSalary > 70000)
        {
            employeeGrade = 'C';
        }
        else if(employeeSalary > 50000)
        {
            employeeGrade = 'D';
        }
        else
        {
            employeeGrade = 'E';
        }
    }
}

```

EmployeeDemo.java

---

```

package com.ravi.oop_ex;

//ELC
public class EmployeeDemo
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        e1.setEmployeeData(1, "Scott", 120000);
        e1.calculateEmployeeGrade();
        e1.getEmployeeData();

    }
}

```

---

What is variable shadow ?

---

If instance variable and parameter variable/local variable both names are same then inside the method/constructor the local variable or parameter variable hides insatnce variable, This concept is known as Variable Shadow.

2 files :

-----  
Student.java

```
-----  
package com.ravi.oop_ex;  
  
public class Student  
{  
    int id = 111;  
    String name = "Scott";  
  
    public void accept()  
    {  
        int id = 222;  
        String name = "Raj";  
  
        System.out.println(id + " : " + name);  
    }  
}
```

-----  
StudentDemo.java

```
-----  
package com.ravi.oop_ex;  
  
public class StudentDemo  
{  
    public static void main(String[] args)  
    {  
        Student s1 = new Student();  
        s1.accept();  
    }  
}
```

Here the output is 222 : Raj because local variable hides instance variable within the method.

-----  
this keyword :

Whenever our instance variable name and parameter variable name both are same then at the time of variable initialization our runtime environment gets confused that which one is an instance variable which one is parameter variable.

To avoid this problem we should use this keyword. this keyword always refers to the current object and we know that instance variables are the part of object.

this keyword we can't use from a static context.

-----  
Product.java

```
-----  
package com.ravi.this_keyword;
```

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

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

    public void getProductData()
    {
        System.out.println("Product Id is :" + productId);
        System.out.println("Product Name is :" + productName);
    }
}
```

ThisKeywordDemo.java

---

```
package com.ravi.this_keyword;

public class ThisKeywordDemo
{
    public static void main(String[] args)
    {
        Product p1 = new Product();
        p1.setProductData(1, "Mobile");
        p1.getProductData();

    }
}
```

---

15-03-2024

---

Program that describes how compiler is searching the variables and helping the JVM to execute the Variables.

```
package com.nit.oop;

public class Demo
{
    static int x = 100; //Class Variable OR static Field

    int y = 200; //Instance Variable OR non static Field

    public void access()
    {
        int z = 300; //Local Variables
        System.out.println(Demo.x);
```

```
        System.out.println(this.y);
        System.out.println(z);
    }

    public static void main(String[] args)
    {
        Demo d1 = new Demo();
        d1.access();
    }
}
```

---

Role of instance variable while creating the Object :

---

In Java whenever we create the object, a separate copy of all the instance variables will be created with each and every object.

```
package com.nit.oop;

public class Test
{
    int x = 10;

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

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

        System.out.println(t1.x); //11
        System.out.println(t2.x); //9
    }
}
```

The above program contains diagram (15-MAR)

---

The role of static variable while creating the object :

---

Whenever we create an object, a single copy of static variable will be created and it is shared by all the objects at the same time so, if any modification is done by any of the object reference then it will be applicable to ALL THE OBJECTS.

```
package com.nit.oop;

public class Demo
{
    static int x = 10; //Class Variable OR static Field

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

        --d1.x; //9
    }
}
```

```
--d2.x; //8  
System.out.println(d1.x); //8  
System.out.println(d2.x); //8  
}  
}
```

instance variable = Multiple Copies  
static variable = Single Copy

---

When we should declare a variable as an instance variable and when we should declare a variable as a static variable ?

Instance Variable :

---

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

Static Variable :

---

If the value of the variable is common with respect to all the objects then we should use static variable.

Example :

```
class Student  
{  
    int roll;  
    String name;  
    String address;  
    static String collegeName = "NIT";  
    static String courseName = "Java";  
}
```

---

Program :

---

2 files :

---

Student.java

---

```
package com.ravi.oop_demo;
```

```
//BLC  
public class Student  
{  
    int rollNumber;  
    String studentName;  
    String studentAddress;  
    static String collegeName = "NIT";  
    static String courseName = "Java";  
  
    public void setStudentData(int rollNumber, String studentName, String studentAddress)  
    {  
        this.rollNumber = rollNumber;  
        this.studentName = studentName;  
        this.studentAddress = studentAddress;
```

```
}

public void displayStudentData()
{
    System.out.println("Roll Number is :" +this.rollNumber);
    System.out.println("Name is :" +this.studentName);
    System.out.println("Address is :" +this.studentAddress);
    System.out.println("College Name is :" +Student.collegeName);
    System.out.println("Course Name is :" +Student.courseName);
}

}
```

## StudentDemo.java

---

```
package com.ravi.oop_demo;

//ELC
public class StudentDemo
{
    public static void main(String[] args)
    {
        Student raj = new Student();
        raj.setStudentData(1, "Raj", "S R Nagar");
        raj.displayStudentData();

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

        Student priya = new Student();
        priya.setStudentData(2, "Priya", "Ameerpet");
        priya.displayStudentData();
    }
}
```

Note :- The main purpose of static variable to save the memory because single copy is sharable by all the objects.

---

16-03-2024

---

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

---

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

Now with the help of `toString()` method we need not 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 :

---

Player.java

---

```
package com.ravi.oop_demo;

//BLC
public class Player {
    int playerId;
    String playerName;

    public void setPlayerData(int playerId, String playerName)
    {
        this.playerId = playerId;
        this.playerName = playerName;
    }

    public String toString()
    {
        return "Player [playerId=" + this.playerId + ", playerName=" + this.playerName + "]";
    }
}
```

PlayerDemo.java

---

```
package com.ravi.oop_demo;
//ELC
public class PlayerDemo {

    public static void main(String[] args)
    {
        Player p1 = new Player();
        p1.setPlayerData(18, "Virat");
        System.out.println(p1); //toString()

        Player p2 = new Player();
        p2.setPlayerData(45, "Rohit");
        System.out.println(p2); //toString()

    }
}
```

---

Data Hiding : [Declaring the instance variable with private AM]

---

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.oop_ex;

//BLC
public class Customer
{
    private double balance = 1000; //Data Hiding

    public void deposit(int depositAmount)
    {
        if(depositAmount <=0)
        {
            System.out.println("Amount can't be deposited");
        }
        else
        {
            balance = balance + depositAmount;
            System.out.println("Amount after deposit :" +balance);
        }
    }

    public void withdraw(int withdrawAmount)
    {
        balance = balance - withdrawAmount;
        System.out.println("Amount after Withdraw :" +balance);
    }
}
```

BankingApplication.java

---

```
package com.ravi.oop_ex;

//ELC
public class BankingApplication
{
    public static void main(String[] args)
    {
        Customer raj = new Customer();
        raj.deposit(10000);
        raj.withdraw(6000);

    }
}
```

---

Abstraction :

---

Showing the essential details without showing the background details is called abstraction.

In our real world a user always interacts with functionality of the product but not the data so as a developer we should hide the data from end user by declaring them private.

On the other hand the function must be declared as public so our end user will interact with the function/method.

In Java we can achieve abstraction by using two ways :

1) Abstract class and abstract methods :- By using abstract class and abstract method we can achieve abstraction 0 to 100%

2) By using interface :- By using interface we can achieve 100% abstraction.

Abstract class example to achieve abstraction :

```
-----  
package com.ravi.oop_ex;  
  
public abstract class ATMMachine  
{  
    public abstract void deposit();  
  
    public abstract void withdraw();  
  
    public abstract void pinChange();  
  
    public void bankName()  
    {  
        System.out.println("ICICI");  
    }  
}
```

---

interface to achieve the abstraction :

```
-----  
package com.ravi.oop_ex;  
  
interface Lift  
{  
    void keyOne();  
    void keyTwo();  
    void keyThree();  
    void keyFour();  
    void keyFive();  
}  
  
class IronLift implements Lift  
{  
  
    @Override  
    public void keyOne() {  
        // TODO Auto-generated method stub  
    }  
}
```

```
@Override  
public void keyTwo() {  
    // TODO Auto-generated method stub  
  
}  
  
@Override  
public void keyThree() {  
    // TODO Auto-generated method stub  
  
}  
  
@Override  
public void keyFour() {  
    // TODO Auto-generated method stub  
  
}  
  
@Override  
public void keyFive() {  
    // TODO Auto-generated method stub  
  
}  
}
```

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

---

18-03-2024

---

Constructor in Java :

---

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.

[Diagram 18-MAR-24]

---

Constructor (In Details) :

---

If the name of the class and name of the method both are exactly same and there is no return type then it is called constructor.

\*The main purpose of constructor to initialize the instance variable (Properties) of an object.

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

By default constructor never contain any return type but implicitly it returns current class object as shown in the program.

```
public class Employee
{
    public Employee()
    {
        System.out.println("Hello");
    }

    void m1()
    {
        System.out.println("m1 method");

    }
}

package com.nit.oop;

public class EmployeeDemo
{
    public static void main(String[] args)
    {
        //Nameless OR Anonymous object
        new Employee().m1();

    }
}
```

If we put any kind of return type in the constructor then it will behave like a method.

We can write return keyword inside a constructor without value.

A constructor is automatically called and executed at the time of object creation. (We can't call it manually like methods)

A Constructor is called once per object that means if an object is created for first time then constructor will be called and executed, in the same way if we create object for 2nd time again constructor will be invoked.

19-03-2024

---

Types of Constructor :

---

Java supports 3 types of constructor :

---

1) Default constructor.

2) No Argument OR Zero Argument OR Parameterless OR Non parameterized Constructor.

3) Parameterized Constructor.

Default Constructor :

If a constructor is added by compiler to initialize the instance variable with default values then it is called Default constructor.

Example :

```
public class Hello  
{  
}
```

```
javac Hello.java
```

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

2) No Argument OR Zero Argument OR Parameterless OR Non parameterized Constructor.

If a constructor is written by user without parameter then it is called No Argument constructor.

Example :

```
public class Student  
{  
    private int sno;  
    private String sname;  
  
    public Student() //No Argument Constructor  
    {  
        sno = 101;  
        sname = "Raj";  
    }  
}
```

By using this no argument constructor, all the objects will be initialized with same value so it is not recommended because we can't initialize the instance variable of two objects with two different values as shown in the program :

2 files :

```
Person.java
```

```
package com.ravi.constructor_demo;  
  
//BLC  
public class Person {
```

```
private int personId;
private String personName;

public Person() {
    this.personId = 101;
    this.personName = "Raj";
}

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

}
```

### NoArgument.java

---

```
package com.ravi.constructor_demo;

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

        Person scott = new Person();
        System.out.println(scott);
    }
}
```

---

### 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 Test
{
    private int x,y;

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

---

### 2 Files :

---

```
Car.java
```

---

```

package com.ravi.constructor_demo;

public class Car {
    private int carModel;
    private String carName;
    private String carColor;
    private int horsePower;

    public Car(int carModel, String carName, String carColor, int horsePower) {
        super();
        this.carModel = carModel;
        this.carName = carName;
        this.carColor = carColor;
        this.horsePower = horsePower;
    }

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

```

#### ParameterizedConstructor.java

---

```

-----
package com.ravi.constructor_demo;

public class ParameterizedConstructor
{
    public static void main(String[] args)
    {
        Car c1 = new Car(2024, "Audi", "White", 1500);
        System.out.println(c1);

        Car c2 = new Car(2023, "Fortuner", "Grey", 1600);
        System.out.println(c2);
    }
}

```

---

#### How to modify my existing object data :

---

In order to modify the existing object data we should use setter method.

#### How to read private data value outside of a class :

---

In order to read my private data value outside of the BLC class then we should use getter.

#### How to write setter and getter :

---

```

public class Student
{
    private int rollNumber;

```

```
public void setRollNumber(int rollNumber) //setter
{
    this.rollNumber = rollNumber;
}

public int getRollNumber()
{
    return this.rollNumber;
}

}
```

----- Program on setter to modify the exiting book price from 900 to 1200.

2 files :

```
package com.ravi.constructor_demo;
```

```
//BLC
public class Book {
    private String bookTitle;
    private double bookPrice;

    public Book(String bookTitle, double bookPrice) {
        super();
        this.bookTitle = bookTitle;
        this.bookPrice = bookPrice;
    }
```

```
//setter to modify the book Title
public void setBookTitle(String bookTitle)
{
    this.bookTitle = bookTitle;
}
```

```
//setter to modify the book price
public void setBookPrice(double bookPrice) {
    this.bookPrice = bookPrice;
}
```

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

BookDemo.java

-----

```
package com.ravi.constructor_demo;

public class BookDemo {

    public static void main(String[] args)
    {
        Book b1 = new Book("Java", 900);
        System.out.println(b1);
        b1.setBookPrice(1200);
        System.out.println(b1);
    }
}
```

---

How to read the private data value from BLC class to ELC class

---

How to use getter to read private data value outside of the class

2 files :

---

Employee.java

---

```
package com.ravi.stter_getter;

//BLC
public class Employee {
    private double empSalary;

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

    public double getEmpSalary() {
        return this.empSalary;
    }

    public void setEmpSalary(double empSalary) {
        this.empSalary = empSalary;
    }

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

---

EmployeeDemo.java

---

```
package com.ravi.stter_getter;

import java.util.Scanner;

public class EmployeeDemo
```

```

{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Employee Salary :");
        double salary = sc.nextDouble();

        Employee e1 = new Employee(salary);

        if(e1.getEmpSalary() <=20000)
        {
            System.out.println("Your designation is Tester");
        }
        else if(e1.getEmpSalary() <=40000)
        {
            System.out.println("Your designation is Designer");
        }
        else
        {
            System.out.println("Your designation is Developer");
        }
    }
}

```

---

The minimum requirements in a BLC class :

---

```

//BLC
public class Student
{

```

- 1) Instance Variables.
- 2) Parameterized Constructor for initialization.
- 3) setter and getter to read and write the private data.
- 4) `toString()` method for printing the object properties.

```
}
```

---

Create two BLC classes Person and PanCard, both must be declared with public access modifier.  
Create one ELC class Test which contains main method to test the application

Designing of Person class :

Instance Variable :

`private PanCard obj;`

create a parameterized constructor which accept PanCard class as a parameter, In this constructor

initialize the instance variable and call the non static display method of PanCard class.

Designing of PanCard class :

Instance Variable : `private String panId;`

create one no argument constructor where initialize the

panId and create the object of Person class

Define display method to print panId; Define main method in ELC class Test to test the application.

3 files :

-----  
Person.java  
-----

```
package com.ravi.lab;

public class Person
{
    private PanCard obj;

    public Person(PanCard obj)
    {
        super();
        this.obj = obj;
        this.obj.display();

    }
}
```

PanCard.java  
-----

```
package com.ravi.lab;

public class PanCard
{
    private String panId;

    public PanCard()
    {
        this.panId = "ABC123456";
        Person p = new Person(this);
    }

    public void display()
    {
        System.out.println(panId);
    }
}
```

Test.java  
-----

```
package com.ravi.lab;

public class Test {

    public static void main(String[] args)
    {
        PanCard p = new PanCard();
```

```
}
```

```
}
```

---

Create a BLC class called A

Instance variable

```
private int data = 15;
```

Create a no-argument constructor, where create the object for class B (Another BLC class) and call the non-static display method available in class B.

Create an instance method show() in class A, which will print instance variable data.

Create an another BLC class B

Instance variable

```
private A obj;
```

Create a parameterized constructor which takes class A as a parameter to initialize the instance variable

Create a display method inside class B which internally calling the show() method of class A

Create an ELC class Test which contains main method to test the application.

Note :- BLC and ELC all the classes must be declared as public.

3 files :

---

A.java

---

```
package com.ravi.constructor;
```

```
public class A
```

```
{
```

```
    private int data = 15;
```

```
    public A()
```

```
{
```

```
        B b1 = new B(this);
```

```
        b1.display();
```

```
}
```

```
    public void show()
```

```
{
```

```
        System.out.println(this.data);
```

```
}
```

```
}
```

B.java

---

```
package com.ravi.constructor;
```

```
public class B
```

```
{
```

```
    private A obj;
```

```
public B(A obj)
{
    super();
    this.obj = obj;
}

public void display()
{
    obj.show();
}

}
```

Test.java

---

```
package com.ravi.constructor;

public class Test {

    public static void main(String[] args)
    {
        A a1 = new A();

    }
}
```

---

\*Encapsulation :

Binding the data with its associated method is called Encapsulation.

Encapsulation forces us to access our data through methods only.

Without data hiding encapsulation is not possible.

Encapsulation provides security because data is hidden from outer world.

We can achieve encapsulation in our class by using 2 process.

- Declarign all the data members with private access modifier(data Hiding).
- Defining setter and getter for all the data members individually.

Note :-

If we declare all the data members with private access modifier then it is called TIGHTLY ENCAPSULATED CLASS.

On the other hand if some data members are declared with private but other are declared with protected or default or public then it is called LOOSLY ENCAPSULATED CLASS.

---

Passing an object reference to the method/Constructor :

---

HAS-A Relation Program :

---

What is HAS-A relation ?

---

If we pass a reference variable of another class type as a property to another class then it is called HAS-A Relation.

```
class Address
{
    private String flatNo;
    private String street;
    private String area;
    private int pincode;
}
```

```
class Student
{
    private int studentNumber;
    private String studentName;
    private double studentFees;
    private Address address; //HAS A Relation
}
```

---

3 files :

---

College.java

---

```
package com.ravi.has_a_relation;

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

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

    @Override
    public String toString() {
        return "College [collegeName :" + collegeName + ", collegeAddress :" +
collegeAddress + "]";
    }
}
```

Student.java

---

```
package com.ravi.has_a_relation;

public class Student {
    private int studentId;
    private String studentName;
    private College clg; // HAS-A Relation
```

```
public Student(int studentId, String studentName, College clg) //clg = c1
{
    super();
    this.studentId = studentId;
    this.studentName = studentName;
    this.clg = clg;
}

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

}

Main.java

```
-----
package com.ravi.has_a_relation;

public class Main
{
    public static void main(String[] args)
    {
        College c1 = new College("NIT", "Hyd");

        Student s1 = new Student(1,"A",c1);
        System.out.println(s1);
    }
}
```

21-03-2024

Program on HAS-A relation :

-----  
3 files :

-----  
Company.java

```
-----
package com.ravi.has_a_reln;

//BLC
public class Company {
    private String companyName;
    private String companyLocation;

    public Company(String companyName, String companyLocation) {
        super();
        this.companyName = companyName;
        this.companyLocation = companyLocation;
    }
}
```

```
@Override  
public String toString() {  
    return "Company [companyName=" + companyName + ", companyLocation=" +  
companyLocation + "]";  
}  
}
```

### Employee.java

---

```
package com.ravi.has_a_reln;  
  
//BLC  
public class Employee {  
    private int employeeNumber;  
    private String employeeName;  
    private double employeeSalary;  
    private Company company; // HAS-A Relation  
  
    public Employee(int employeeNumber, String employeeName, double employeeSalary,  
Company company) //company = comp  
    {  
        super();  
        this.employeeNumber = employeeNumber;  
        this.employeeName = employeeName;  
        this.employeeSalary = employeeSalary;  
        this.company = company;  
    }  
  
    @Override  
    public String toString() {  
        return "Employee [employeeNumber=" + employeeNumber + ", employeeName=" +  
employeeName + ", employeeSalary=" +  
            + employeeSalary + ", company=" + company + "]";  
    }  
}
```

### Main.java

---

```
package com.ravi.has_a_reln;  
  
public class Main  
{  
    public static void main(String[] args)  
    {  
        Company comp = new Company("TCS", "Hyderabad");  
  
        Employee e1 = new Employee(1, "Scott", 40000, comp);  
        System.out.println(e1);  
  
        Employee e2 = new Employee(2, "Smith", 45000, comp);  
        System.out.println(e2);  
    }  
}
```

```
    }  
}
```

---

Copy the data from one object to another object :

---

(Diagram is available for both the Program (21-MARCH))

2 files :

---

```
package com.ravi.has_a_reln;  
  
public class Product  
{  
    int productId;  
    String productName;  
  
    @Override  
    public String toString() {  
        return "Product [productId=" + productId + ", productName=" + productName + "]";  
    }  
}
```

ProductDemo.java

---

```
package com.ravi.has_a_reln;  
  
public class ProductDemo  
{  
    public static void main(String[] args)  
    {  
        Product p = new Product();  
        p.productId = 1;  
        p.productName = "Mobile";  
        System.out.println(p);  
  
        p = new Product();  
        System.out.println(p);  
    }  
}
```

---

2 files :

---

2 files :

---

```
package com.ravi.has_a_reln;  
  
public class Product  
{  
    int productId;  
    String productName;  
  
    @Override  
    public String toString() {  
        return "Product [productId=" + productId + ", productName=" + productName + "]";  
    }  
}
```

```
}
```

## ProductDemo.java

---

```
package com.ravi.has_a_reln;
public class ProductDemo
{
    public static void main(String[] args)
    {
        Product p1 = new Product();
        p1.productId = 1;
        p1.productName = "Mobile";

        Product p2 = p1;
        p2.productId = 2;
        p2.productName = "Laptop";
        System.out.println(p1);
        System.out.println(p2);

    }
}
```

---

## Description of System.out.println() :

---

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

System.out.println();
```

In System class, we have a PrintStream class available in java.io package which creates HAS-A Relation with System class.

## Program which describes System.out.println()

---

```
package com.ravi.has_a_reln;

public class Test
{
    public static String out = "Hyderabad";

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

---

## Passing an object reference to the Constructor (Copy Constructor)

---

The main purpose to pass an object reference to the constructor is to copy the content of one object to another object. Here we are copying the data of Employee to Manager class.

3 files :

-----  
Employee.java  
-----

```
package com.ravi.passing_object_ref;

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;
    }
}
```

Manager.java  
-----

```
package com.ravi.passing_object_ref;

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 +
    "]";
    }
}
```

PassingObjectRef.java  
-----

```
package com.ravi.passing_object_ref;

public class PassingObjectRef {

    public static void main(String[] args)
    {
        Employee e1 = new Employee(101, "Virat");

        Manager m1 = new Manager(e1);
        System.out.println(m1);
    }
}
```

---

Copy the content of same object by passing the reference to the

---

Constructor:

---

2 files :

---

Player.java

---

```
package com.ravi.obj_ref;

public class Player
{
    private String name1, name2;

    public Player(String name1, String name2)
    {
        super();
        this.name1 = name1;
        this.name2 = name2;
    }

    public Player(Player p) //p = p1
    {
        this.name1 = p.name2;
        this.name2 = p.name1;
    }

    @Override
    public String toString() {
        return "Player [name1=" + name1 + ", name2=" + name2 + "]";
    }
}
```

PlayerDemo.java

---

```
package com.ravi.obj_ref;

public class PlayerDemo {

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

```

        Player p1 = new Player("Rohit","Virat");
        Player p2 = new Player(p1);

        System.out.println(p1);
        System.out.println(p2);
    }
}

```

Note :- Here we copied the Player object to another Player object.

---

Working with method return type as a class :

---

As we know we can take different types of return type while defining a method which are as follows :

- 1) void
- 2) Any primitive type (byte to boolean)
- 3) Any class name, interface name or user defined data type

2 files :

---

Customer.java

---

```

package com.ravi.method_return;

//BLC
public class Customer
{
    private int customerId;
    private String customerName;
    private double customerBill;

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

    public static Customer getCustomerObject()
    {
        return new Customer(1, "Scott", 12000);
    }

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

```

CustomerDemo.java

```
-----  
package com.ravi.method_return;  
  
//ELC  
public class CustomerDemo  
{  
    public static void main(String[] args)  
    {  
        Customer object = Customer.getCustomerObject();  
        System.out.println(object);  
    }  
}
```

-----  
23-03-2024

-----  
Program that describes method return type as a class :

-----  
2 files :

-----  
Book.java

```
-----  
package com.ravi.method_return;  
  
import java.time.LocalDate;  
import java.util.Scanner;  
  
public class Book {  
    private String bookTitle;  
    private String authorName;  
    private LocalDate publishedDate; // HAS-A relation  
  
    public Book(String bookTitle, String authorName, LocalDate publishedDate) {  
        super();  
        this.bookTitle = bookTitle;  
        this.authorName = authorName;  
        this.publishedDate = publishedDate;  
    }  
  
    public static Book getBookObject()  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter Title of the book :");  
        String title = sc.nextLine();  
        System.out.print("Enter Author of the book :");  
        String author = sc.nextLine();  
  
        LocalDate date = LocalDate.now();  
  
        return new Book(title, author, date);  
    }  
}
```

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

### BookDemo.java

---

```
package com.ravi.method_return;

import java.util.Scanner;

public class BookDemo {

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter how many book objects :");
        int numberOfObjects = sc.nextInt();

        for(int i=1; i<=numberOfObjects; i++)
        {
            Book book = Book.getBookObject();
            System.out.println(book);
        }
    }
}
```

---

### Programm on setter and getter :

---

2 files :

---

### Course.java

---

```
package com.ravi.method_return;

public class Course
{
    private String subjectName;
    private String facultyName;
    private int courseDuration;

    public Course(String subjectName, String facultyName, int courseDuration) {
        super();
        this.subjectName = subjectName;
        this.facultyName = facultyName;
        this.courseDuration = courseDuration;
    }

    @Override
```

```

public String toString() {
    return "Course [subjectName=" + subjectName + ", facultyName=" + facultyName +
", courseDuration="
        + courseDuration + "]";
}

public void setSubjectName(String subjectName) {
    this.subjectName = subjectName;
}

public void setFacultyName(String facultyName) {
    this.facultyName = facultyName;
}

public void setCourseDuration(int courseDuration) {
    this.courseDuration = courseDuration;
}

public String getSubjectName() {
    return subjectName;
}

}

SetterExample.java
-----
```

```

package com.ravi.method_return;

public class SetterExample {

    public static void main(String[] args)
    {
        Course javaPlacement = new Course("C", "Kishore Sir", 45);
        System.out.println(javaPlacement);

        if(javaPlacement.getSubjectName().equals("C"))
        {
            System.out.println("Ready for Core Java");
        }

        javaPlacement.setSubjectName("Core Java");
        javaPlacement.setFacultyName("Ravi");
        javaPlacement.setCourseDuration(90);
        System.out.println(javaPlacement);

        if(javaPlacement.getSubjectName().equals("Core Java"))
        {
            System.out.println("Be ready for Adv Java");
        }
    }
}
```

---

Lab Program :(Method return type as a class + Passing Object ref)

---

A class called Customer is given to you.

The task is to find the Applicable Credit card Type and create CardType object based on the Credit Points of a customer.

Define the following for the class.

Attributes :

customerName : String, private  
creditPoints: int, private

Constructor :

parameterizedConstructor: for both cusotmerName & creditPoints in that order.

Methods :

Name of the method : getCreditPoints  
Return Type : int  
Modifier : public  
Task : This method must return creditPoints

Name of the method : toString, Override it,  
Return type : String  
Task : return only customerName from this.

Create another class called CardType. Define the following for the class

Attributes :

customer : Customer, private  
cardType : String, private

Constructor :

parameterizedConstructor: for customer and cardType attributes in that order

Methods :

Name of the method : toString Override this.  
Return type : String  
Modifier : public  
Task : Return the string in the following format.  
The Customer 'Rajeev' Is Eligible For 'Gold' Card.

Create One more class by name CardsOnOffer and define the following for the class.

Method :

Name Of the method : getOfferedCard  
Return type : CardType  
Modifiers: public,static  
Arguments: Customer object

Task : Create and return a CardType object after logically finding cardType from creditPoints as per the below rules.

creditPoints	cardType
100 - 500	- Silver
501 - 1000	- Gold
1000 >	- Platinum

< 100 - EMI

Create an ELC class which contains Main method to test the working of the above.

-----  
Customer.java  
-----

```
package com.ravi.credircard_program;

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

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

    public int getCreditPoints()
    {
        return this.creditPoints;
    }

    @Override
    public String toString()
    {
        return this.customerName;
    }
}
```

CardType.java  
-----

```
package com.ravi.credircard_program;

public class CardType {
    private Customer customer; // HAS-A relation
    private String cardType;

    public CardType(Customer customer, String cardType) {
        super();
        this.customer = customer;
        this.cardType = cardType;
    }

    @Override
    public String toString()
    {
        //The Customer 'Rajeev' Is Eligible For 'Gold' Card.
        return "The Customer "+this.customer+" Is Eligible For "+this.cardType+" Card.";
    }
}
```

CardsOnOffer.java

```
-----
package com.ravi.credircard_program;

public class CardsOnOffer
{
    public static CardType getOfferedCard(Customer c)
    {
        if(c.getCreditPoints() >=100 && c.getCreditPoints() <=500)
        {
            return new CardType(c, "Silver");
        }
        else if(c.getCreditPoints() >500 && c.getCreditPoints() <=1000)
        {
            return new CardType(c, "Gold");
        }
        else if(c.getCreditPoints() > 1000)
        {
            return new CardType(c, "Platinum");
        }
        else
        {
            return new CardType(c, "EMI");
        }
    }
}
```

package com.ravi.credircard\_program;

import java.util.Scanner;

public class CreditCard {

```
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Customer Name :");
        String name = sc.nextLine();
        System.out.print("Enter Customer Credit Points :");
        int creditPoint = sc.nextInt();

        Customer c1 = new Customer(creditPoint, name);
        CardType obj = CardsOnOffer.getOfferedCard(c1);
        System.out.println(obj);
    }
}
```

-----  
26-03-2024

-----  
Instance Block OR Non Static block in java :

-----  
What is Instance OR non-static block in Java ?

---

```
//Instance Block OR Non-static block
```

```
{  
}
```

It is a special block in java which is executed automatically whenever an object is created.  
[Depends upon object creation]

It will be automatically placed in the 2nd line of the constructor, if it is available in the class.

The main purpose of instance block to initialize the instance variable of the class before constructor body execution so, it is also known as instance initializer.

Always the instance block will be executed before the constructor body execution.

If we have n number of instance block available in the class then it would be executed according to the order.

If we write the instance block in the body of the constructor then compiler will not placed in the 2nd line of Constructor.

---

```
InstanceDemo1.java
```

---

```
package com.ravi.instance_demo;  
  
class Test  
{  
    {  
        System.out.println("Instance block");  
    }  
}
```

```
public class InstanceDemo1  
{  
    public static void main(String[] args)  
    {  
        new Test();  
    }  
}
```

---

```
InstanceDemo2.java
```

---

```
package com.ravi.instance_demo;  
  
class Demo  
{  
    int x = 100;  
}
```

```
        System.out.println("x value by instance block :" +this.x);
        x = 200;
        System.out.println("x value by instance block :" +this.x);
    }

    Demo()
{
    x = 300;
    System.out.println("x value by No Arg Constructor :" +this.x);
}
}
```

```
public class InstanceDemo2
{
    public static void main(String[] args)
    {
        new Demo();
    }
}
```

---

### InstanceDemo3.java

---

```
package com.ravi.instance_demo;

class Foo
{
    int y = 10;

    {
        System.out.println(y);
        y = 20;
        System.out.println(y);
    }

    {
        y = 30;
        System.out.println(y);
    }

    {
        y = 40;
        System.out.println(y);
    }

    Foo()
    {
        y = 50;
        System.out.println(y);
    }
}
```

```
}
```

```
public class InstanceDemo3 {
```

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

```
}
```

---

```
InstanceDemo4.java
```

---

```
package com.ravi.instance_demo;

class Instance
{
    int x = 10;

    public Instance()
    {
        x = 30;
        System.out.println(this.x);

        {
            x = 20;
            System.out.println(this.x);
        }
    }

}
```

```
}
```

```
public class InstanceDemo4
```

---

```
{
```

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

---

Can we write a constructor with private access modifier :

---

Yes, We can declare a constructor with all access modifiers like private, default, protected and public.

As per our requirement, we can declare a constructor with private access modifier which is having two advantages

- 1) Object will be created internally so, any end user can't create an object for the class. If we create an object only once then that class is known as Singleton class.
- 2) If we are declaring only static methods and static variables inside a class then we should declare the constructor as a private

constructor.

```
package com.ravi.instance_demo;

public class SingleTon
{
    private SingleTon()
    {

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

---

27-03-2024

---

Garbage Collector :

---

In C++, It is the responsibility of the programmer to allocate as well as to de-allocate the memory otherwise the corresponding memory will be blocked and we will get OutOfMemoryError.

In Java, Programmer is responsible to allocate the memory, memory de-allocation will be automatically done by garbage collector.

Garbage Collector will scan the heap area, identify which objects are not in use (The objects which does not contain any references) and it will delete those objects which are not in use.

---

How many ways we can make an object eligible for garbage collector :

There are 3 ways to make an object eligible for Garbage Collector.

1) Assigning null literal to the reference variable :

```
Employee e1 = new Employee();
e1 = null;
```

2) Creating an object inside the method :

```
public void createObject()
{
    Employee e2 = new Employee();
}
```

Here we are creating the employee object inside a method so, once method execution is over the employee object is eligible for GC.

3) Assigning another object to the existing reference variable :

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

---

## HEAP and STACK Diagram for CustomerDemo.java

---

```
class Customer
{
    private String name;
    private int id;

    public Customer(String name , int id) //constructor
    {
        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 i.e 3000x is eligible for GC]

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

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

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

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

---

28-03-2024

---

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

//Output 20 9

---

#### HEAP and STACK Diagram for Employee.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 = 500;

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

```

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

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

public static void update(Employee e) // e = e1
{
e.id = 500;
    e = new Employee();
    e.id = 400;
}

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

```

//500 500 500

---

#### HEAP and STACK Diagram 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; //3000x
        t3.t = t4; //4000x
        t1.t = t2.t; //3000x
    }
}

```

```
t2.t = t4.t; //2000x

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

---

01-04-2024

---

Relationship between the classes :

---

In Java, in between the classes we have 2 types of relation

- 1) IS-A Relation
- 2) HAS-A Relation

IS-A relation we can achieve using Inheritance.

HAS-A relation we can achieve using Association.

Example of IS-A relation :

```
class Vehicle
{
}

class Car extends Vehicle
{
}
```

In between Vehicle and Car we have IS-A relation because Car IS-A Vehicle.

Example of HAS-A relation :

```
class Vehicle
{
    private Engine engine;
}
```

In between Vehicle and Engine, We have HAS-A relation because Vehicle has an engine.

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 important 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 begining 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.

---

In java, We can classify inheritance into 5 types

- 1) Single Level Inheritance
- 2) Multilevel Inheritance
- 3) Hierarchical Inheritance
- 4) Multiple Inheritance (Not supported by using classes)
- 5) Hybrid Inheritance

Note :- In java, by default `java.lang.Object` is the super class of all the classes we have in java.

---

Program on Single level Inheritance :

---

3 Files :

---

-----  
Super.java

```
-----  
package com.ravi.inheritance;  
  
public class Super  
{  
    private int x, y;  
  
    public void acceptData()  
    {  
        x = 100;  
        y = 200;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}
```

Sub.java

---

```
-----  
package com.ravi.inheritance;  
  
public class Sub extends Super  
{
```

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

## SingleLevel.java

---

```
package com.ravi.inheritance;

public class SingleLevel {

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

}
```

---

02-04-2024

---

How to initialize the super class property :

---

In order to call the member of super class we need to create sub class object but in case of constructor, it is not inherited so we compiler will add super keyword to maintain the hierarchy.

How to call the member of super class :

---

Here we can use super keyword by using 3 ways :

- 1) To call the super class constructor.
- 2) To call the super class variable.
- 3) To call the super class method.

1) To call the super class constructor :

---

\*\*To call 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 constructor to the class.

THE FIRST LINE OF ANY CONSTRUCTOR IS RESERVERD EITHER FOR super() or this() keyword.

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() : Calling the No argument OR default constructor of super class.

ConstructorDemo.java [Single File Approach]

-----  
package com.ravi.constructor\_chain;

```
class A
{
    public A()
    {
        System.out.println("class A");
    }
}
class B extends A
{
    public B()
    {
        System.out.println("class B");
    }
}

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

Note :- In the first line of constructor, automatically the compiler will add super() to call super class constructor.

-----  
Case 2 :

-----  
super(15) : It is used to call the parameterized constructor of super class.

ConstructorDemo.java

-----  
package com.ravi.constructor\_chain;

class Alpha
{
 public Alpha(int x)
 {
 super();
 System.out.println("Super class Alpha :" + x);
 }
}
class Beta extends Alpha

```

{
    public Beta()
    {
        super(9); //parameterized constructor of super class
        System.out.println("Sub class Beta constructor..");
    }
}

public class ConstructorDemo
{
    public static void main(String[] args)
    {
        new Beta();
    }
}

```

---

### Case 3 :

`this()` : It is used to call current class no argument constructor.

```

package com.ravi.constructor_chain;

class Parent
{
    public Parent()
    {
        System.out.println("Super class no argument constructor!!");

    }

    public Parent(String str)
    {
        this();
        System.out.println("Super class Parameterized constructor!!"+str);
    }
}

class Child extends Parent
{
    public Child()
    {
        super("Batch 30");
        System.out.println("Sub class no argument constructor!!");
    }
}

public class ConstructorDemo
{
    public static void main(String[] args)
    {
        new Child();
    }
}

```

---

#### Case 4 :

---

this(String str) : It is used to call parameterized constructor of current class.

```
package com.ravi.constructor_chain;

class Base
{
    public Base()
    {
        this("NIT");
        System.out.println("No Arg constructor of Super class!!");

    }
    public Base(String str)
    {
        System.out.println("Parameterized constructor of Super class :" +str);
    }
}
class Derived extends Base
{
    public Derived()
    {
        System.out.println("No Arg constructor of Sub class!!");
    }
}

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

---

#### Program on Single Level Inheritance:

---

##### SingleLevel.java

---

```
package com.ravi.single_level;

import java.util.Scanner;

class Shape
{
    private int x;

    public Shape(int x) //x = side
    {
        super();
        this.x = x;
    }
}
```

```

        public int getX() {
            return x;
        }

    }

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

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

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

            Square ss = new Square(side);
            ss.getAreaOfSquare();
        }
    }

```

---

03-04-2024

---

Program on Hierarchical Inheritance:

---

```

package com.ravi.hierarchical;

import java.util.Scanner;

class Shape
{
    protected int x;

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

    class Square extends Shape
    {

```

```

public Square(int side)
{
    super(side);
}

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

class Rectangle extends Shape
{
    protected int breadth;

    public Rectangle(int l , int b)
    {
        super(l);
        this.breadth = b;
    }

    public void getAreaOfRectangle()
    {
        System.out.println("Area of Rectangle is :" +(x*breadth));
    }
}

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

        Square ss = new Square(side);
        ss.getAreaOfSquare();

        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 rr = new Rectangle(length, breadth);
        rr.getAreaOfRectangle();
    }
}

```

---

2) calling the super class variable :

---

If the super class variable name and sub class variable name both are same then it is called variable shadow, Here sub class variable will hide super class variable.

If we create an object for sub class then by default it will access sub class variable, if we want to access super class variable then we should use super keyword.

super keyword always refers to its immediate super class.

Just like this keyword we cannot use super keyword from static context.

```
package com.ravi.hierarchical;
```

```
class Father
{
    protected double balance = 45000;
}

class Son extends Father
{
    protected double balance = 12000;

    public void getBalance()
    {
        System.out.println(this.balance);
        System.out.println(super.balance);
    }

    public void addBalance()
    {
        System.out.println("Sum of balance is "+this.balance+ super.balance);
    }
}
```

```
public class Supervariable {
```

```
    public static void main(String[] args)
    {
        Son s = new Son();
        s.getBalance();
        s.addBalance();
    }
}
```

---

### 3) Calling the super class method :

---

Whenever super class method name and sub class method name both are same and if we create an object for sub class then by defulat it will access sub class method.

If we want to access super class method then we should use super keyword.

---

//Program on super class method :

---

```
package com.ravi.hierarchical;
```

```
class Super
{
    public void show()
    {
        System.out.println("super class show method!!");
    }
}
class Sub extends Super
{
    public void show()
    {
        System.out.println("Sub class show method!!");
        super.show();
    }
}
```

```
public class SuperMethod {

    public static void main(String[] args)
    {
        new Sub().show();
    }
}
```

---

//Program on instance block :

---

```
package com.ravi.hierarchical;

class Alpha
{
    public Alpha()
    {
        System.out.println("A");
    }

    {
        System.out.println("B");
    }
}
```

```
class Beta extends Alpha
{
    public Beta()
    {
        System.out.println("C");
    }

    {
        System.out.println("D");
    }
}
```

```
public class InstanceBlockDemo {
```

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

}

}
```

Output is //B A D C

---

Program on Single Level Inheritance :

---

3 files :

---

Employee.java

---

```
package com.nit.inheritance;

public class Employee {
    protected int employeeNumber;
    protected String employeeName;
    protected double employeeSalary;

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

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

}

PermanentEmployee.java

---

```
package com.nit.inheritance;

public class PermanentEmployee extends Employee {
    protected String department;
    protected String designation;

    public PermanentEmployee(int employeeNumber, String employeeName, double
employeeSalary, String department,
            String designation) {
        super(employeeNumber, employeeName, employeeSalary);
        this.department = department;
    }
}
```

```
        this.designation = designation;
    }

    @Override
    public String toString() {
        return super.toString()+"PermanentEmployee [department=" + department +",
designation=" + designation + "]";
    }

}
```

### SingleLevelInheritance.java

---

```
package com.nit.inheritance;

public class SingleLevelInheritance {

    public static void main(String[] args)
    {
        PermanentEmployee p = new PermanentEmployee(1, "Scott", 56000, "Computer",
"Java Developer");
        System.out.println(p);

    }
}
```

## 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() //All the objects will be initialized
    {                               with same value
        x = 100; y = 200;
    }
}
```

```
Test t1 = new Test(); t1.setData(); [x = 100 y = 200]
Test t2 = new Test(); t2.setData(); [x = 100 y = 200]
```

#### 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 = 100; y = 200;  
}  
}
```

```
Test t1 = new Test(); [x = 100 y = 200]  
Test t2 = new Test(); [x = 100 y = 200]
```

#### 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();
```

---

04-04-2024

---

Program on multi level inheritance -----

MultiLevel.java

---

```
package com.ravi.multi_level;

class GrandFather
{
    public void land()
    {
        System.out.println("2 acres land");
    }
}
class Father extends GrandFather
{
    public void house()
    {
```

```

        System.out.println("3 BHK House");
    }
}
class Son extends Father
{
    public void car()
    {
        System.out.println("Audi Car");
    }
}

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

```

---

Program on multi level inheritance -----

```
package com.ravi.multi_level;
```

```

class Student
{
    protected int studentNumber;
    protected String studentName;
    protected String studentAddress;
    protected double studentFees;

    public Student()
    {

    }

    public Student(int studentNumber, String studentName, String studentAddress, double
studentFees) {
        super();
        this.studentNumber = studentNumber;
        this.studentName = studentName;
        this.studentAddress = studentAddress;
        this.studentFees = studentFees;
    }
}
class Science extends Student
{
    protected int phy, che;

    public Science(int studentNumber, String studentName, String studentAddress, double
studentFees, int phy, int che) {
        super(studentNumber, studentName, studentAddress, studentFees);
        this.phy = phy;
        this.che = che;
    }
}
```

```

}

class PCM extends Science
{
    protected int math;

    public PCM(int studentNumber, String studentName, String studentAddress, double
studentFees, int phy, int che,
              int math) {
        super(studentNumber, studentName, studentAddress, studentFees, phy, che);
        this.math = math;
    }

    @Override
    public String toString() {
        return "PCM [math=" + math + ", phy=" + phy + ", che=" + che + ", studentNumber="
+ studentNumber
                + ", studentName=" + studentName + ", studentAddress=" +
studentAddress + ", studentFees=" + studentFees
                + "]";
    }
}

```

```

public class MultiLevelInheritance
{
    public static void main(String[] args)
    {
        PCM p = new PCM(1,"A","S.R Nagar",12000,78,89,99);
        System.out.println(p);
    }
}

```

---

Program on Hierarchical inheritance -----  
 package com.ravi.hierarchical;

```

class Employee
{
    protected double salary;

    public Employee(double salary)
    {
        super();
        this.salary = salary;
    }
}
class Developer extends Employee
{
    public Developer(double salary)
    {
        super(salary);
    }
}

```

```

        public double getSalary()
        {
            return this.salary;
        }

    }

class Designer extends Employee
{
    public Designer(double salary)
    {
        super(salary);

    }

    public double getSalary()
    {
        return this.salary;
    }
}

public class Hierarchical
{
    public static void main(String[] args)
    {
        Developer d = new Developer(55000);
        System.out.println("Developer Salary is :" + d.getSalary());

        Designer d1 = new Designer(25000);
        System.out.println("Designer Salary is :" + d1.getSalary());
    }
}

```

\* Why java does not support multiple Inheritance :

Java does not support multiple inheritance using classes, because if a sub class inherits two or more than two super classes then the default constructor added by the compiler will generate ambiguity issue to call the super class constructor.(04-APRIL)

It is also known as Diamond Problem in java.

We can achieve multiple Inheritance by using interface concept.

05-04-2024

Access Modifier in java :

It is used to describe the accessibility level of the class as well as the member of the class.

In terms of accessibility, java software people has provided 4 access modifiers.

- a) private (Accessible within the same class only)
- b) default (Accessible within the same package only)
- c) protected (Accessible from another package also but using inheritance)
- d) public (No restriction accessible from everywhere)

private :-

-----  
It is an access modifier and 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 and sealed (must have sub class) 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.

2 files :-

-----  
Test.java (It is available in pack1 package)  
package pack1;

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

Main.java(It is available in pack2 package)  
package pack2;

```
import pack1.Test;

public class Main {

    public static void main(String[] args)
    {
        Test t1 = new Test();
        System.out.println(t1.x); //error because x is
                               default AM
    }
}
```

-----  
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.

Here to access the protected member of super class, we need to create the object for sub class.

2 files :

---

Test.java (It is available in pack1 package)

```
package pack1;

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

Main.java (It is available in pack2 package)

```
package pack2;

import pack1.Test;

public class Main extends Test
{
    public static void main(String[] args)
    {
        Main m = new Main();
        System.out.println(m.x);

    }
}
```

---

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.

---

06-04-2024

---

HAS-A relation between the classes :

---

In order to achieve HAS-A relation concept we should use Association.

---

Association (Relationship between the classes through Object reference)

---

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 regNumber;
    private String studentName;
    private double studentFees;
    private String studentAddress;

    public Student(int regNumber, String studentName, double studentFees, String
studentAddress) {
        super();
        this.regNumber = regNumber;
        this.studentName = studentName;
        this.studentFees = studentFees;
        this.studentAddress = studentAddress;
    }

    public int getRegNumber() {
        return regNumber;
    }

    public void setRegNumber(int regNumber) {
        this.regNumber = regNumber;
    }

    public String getStudentName() {
        return studentName;
    }

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

    public double getStudentFees() {
        return studentFees;
    }

    public void setStudentFees(double studentFees) {
        this.studentFees = studentFees;
    }
}
```

```
}

public String getStudentAddress() {
    return studentAddress;
}

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

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

#### Faculty.java

```
-----
package com.ravi.association;

import java.util.Scanner;

public class Faculty
{
    public static void viewStudentProfile(Student s)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Student Registration Number :");
        int regNo = sc.nextInt();

        if(regNo == s.getRegNumber())
        {
            System.out.println(s);
        }
        else
        {
            System.out.println("Sorry!!! Student is not available");
        }
    }
}
```

#### AssociationDemo.java

```
-----
package com.ravi.association;

public class AssociationDemo {

    public static void main(String[] args)
    {
        Student scott = new Student(1, "Scott", 12000, "Ameerpet");

        Faculty.viewStudentProfile(scott);
    }
}
```

```
}
```

```
}
```

---

### 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 where the child object is an integral part of the parent object. It means that the child object cannot exist without the parent object.

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.

---

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 int carModel;  
    private String carName;  
    private final Engine engine; //HAS-A Relation  
  
    public Car(int carModel, String carName)  
    {  
        super();  
        this.carModel = carModel;
```

```

        this.carName = carName;
        this.engine = new Engine("Battery", 1000);

    }

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

```

### CompositionDemo.java

```

-----
package com.ravi.composition;

public class CompositionDemo {

    public static void main(String[] args)
    {
        Car c1 = new Car(2024, "Kia");
        System.out.println(c1);
    }
}

```

Note :- Composition provides tightly coupled relation between the classes so if we create an object for Car class automatically Engine class object will be created.

### Assignment :

- 
- 1) Person and Heart.
  - 2) Motherboard and Laptop.
- 

### Aggregation (Weak Reference) :

Aggregation in Java is another form of association between classes that represents a "HAS-A" relationship, but with a weaker bond compared to composition.

In aggregation, one class contains an object of another class, but the contained object can exist independently of the container. If the container object is destroyed, the contained object can still exist.

### 3 files :

#### Address.java

```

-----
package com.ravi.aggregation;

public class Address {
    private String houseNo;
    private String streetNo;
    private String area;
}

```

```
public Address(String houseNo, String streetNo, String area) {
    super();
    this.houseNo = houseNo;
    this.streetNo = streetNo;
    this.area = area;
}

@Override
public String toString() {
    return "Address [houseNo=" + houseNo + ", streetNo=" + streetNo + ", area=" + area
+ "]";
}
```

}

Customer.java

---

```
package com.ravi.aggregation;

public class Customer
{
    private int customerId;
    private String customerName;
    private Address address; // HAS-A Relation

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

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

}

AggregationDemo.java

---

```
package com.ravi.aggregation;

public class AggregationDemo {

    public static void main(String[] args)
    {
        Address addr = new Address("B-61", "M G Road", "Hyderabad");

        Customer c1 = new Customer(123, "Scott", addr);
    }
}
```

```
        System.out.println(c1);
    }
-----
```

08-04-2024

-----  
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)
```

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 AND TYPE.

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 OR Runtime Polymorphism

The polymorphism which exist at runtime is called Dynamic polymorphism 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.

IQ :

Can we overload the main/static method ?

We can overload the main/static method but JVM will always search the main method which takes String array as a parameter.

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

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

10-04-2024

Rules for overloading method :

- 1) Method name must be same and argument must be different.
- 2) Except name of the method we can change everything like method parameter, access modifier, return type of the method and so on.

Example :

```
public void fly(int numMiles) {}
public void fly(short numFeet) {}
```

```
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) {}
public void fly(short numFeet, int numMiles){}
```

Note : All the above are valid overloaded method.

3) If method and parameter all are same, only there is a change in method return type then it is not a valid overloaded method

Example :

```
public void fly(int numMiles) {}
public int fly(int numMiles) {} //Invalid (Error)
```

4) If we overload non static method with static method where method name and parameter both are same then it is also not a valid overloaded method.

Example :

```
public void fly(int numMiles) {}
public static void fly(int numMiles) {} //Invalid (Error)
```

---

Passing data among the methods :

---

Java language always works with PASS BY VALUE but not pass by reference.

Test1.java

---

```
package com.ravi.passing_data_among_methods;

public class Test1 {

    public static void main(String[] args)
    {
        int num = 4;
        modifyNumber(num);
        System.out.println(num);

    }
    public static void modifyNumber(int num)
    {
        num = 8;
    }
}
```

---

Test2.java

---

```
package com.ravi.passing_data_among_methods;

public class Test2
{
    public static void main(String[] args)
    {
        String name = "Raj";
        modifyName(name);
        System.out.println(name);
    }
}
```

```

        }
    public static void modifyName(String name)
    {
        name = "Ankit";
    }
}

-----
package com.ravi.passing_data_among_methods;

class Customer
{
    int custId; //111

    Customer(int id)
    {
        custId= id;
    }
}

public class Test3
{
    public static void main(String[] args)
    {
        Customer c1 = new Customer(111);
        changId(c1);
        System.out.println(c1.custId);
    }

    public static void changId(Customer c)
    {
        c.custId = 222;
    }
}

```

Note :- Here we will get the output as 222 because c1 and c both reference variable is pointing to the same customer object hence if we modify the object by any of the reference variable then orginal object will be modified.

```

-----
package com.ravi.passing_data_among_methods;

public class Test4
{
    public static void main(String[] args)
    {
        int original1 = 1;
        int original2 = 2;
        swap(original1, original2);
        System.out.println(original1);
        System.out.println(original2);
    }

    public static void swap(int a, int b) {

```

```
        int temp = a;
        a = b;
        b = temp;
    }
}

-----
package com.ravi.passing_data_among_methods;

public class Test6
{
    public static void main(String[] args) {
        int number = 1;
        String letters = "abc";
        number = number(number);
        letters(letters);
        System.out.println(number + letters);
    }
    public static int number(int number)
    {
        number++;
        return number;
    }
    public static String letters(String letters)//abc
    {
        letters += "d";
        return letters;
    }
}
```

#### Program on Constructor Overloading :

-----  
2 files :

-----  
Addition.java

```
package com.ravi.constructor_overloading;

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

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

    public Addition(float x, float y)
    {
        System.out.println("Sum of two float is :" +(x+y));
    }
}
```

Main.java

```
-----  
package com.ravi.constructor_overloading;  
  
public class Main {  
  
    public static void main(String [] args)  
    {  
        new Addition(2.3f, 7.8F);  
        new Addition(10, 20, 30);  
        new Addition(12,90);  
    }  
}
```

-----  
2 files :

Addition.java

```
-----  
package com.ravi.constructor_overloading1;  
  
public class Addition  
{  
    public Addition(int x, int y)  
    {  
        System.out.println("Sum of two integer is :" +(x+y));  
    }  
  
    public Addition(int x, int y, int z)  
    {  
        this(100,200);  
        System.out.println("Sum of three integer is :" +(x+y+z));  
    }  
  
    public Addition(float x, float y)  
    {  
        this(10,20,30);  
        System.out.println("Sum of two float is :" +(x+y));  
    }  
}
```

Main.java

```
-----  
package com.ravi.constructor_overloading1;  
  
public class Main {  
  
    public static void main(String [] args)  
    {  
        new Addition(2.3f, 7.8F);  
    }  
}
```

Program on Constructor overloading using constructor chaining with instance block.

In general, instance blocks are placed in the 2nd line of every constructor which is related to object creation, but if we have constructor chaining then instance block will be placed to the constructor where super() call is there, after execution of Object class constructor, instance block will be executed so in the below program, instance block will be executed only once becoz object is created only once.

2 files :

-----  
Addition.java  
-----

```
package com.ravi.constructor_overloading2;

public class Addition
{
    public Addition()
    {
        this(10);
        System.out.println("No Argument Constructor");
    }
    public Addition(int x)
    {
        this(10,20);
        System.out.println("One Argument Constructor :" +x);

    }
    public Addition(int x, int y)
    {
        super();
        //This line is reserved for Instance Block
        System.out.println("Two Argument Constructor :" +x+ ":" +y);
    }

    {
        System.out.println("Instance Block");
    }
}
```

Test.java

-----  
package com.ravi.constructor\_overloading2;

public class Test {

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

 }
}

Method Overloading by changing the return type of the Method :

-----  
2 files :

-----  
Sum.java  
-----

```
package com.ravi.method_overload;

public class Sum
{
    public int add(int x, int y)
    {
        int z = x+y;
        return z;
    }

    public String add(String x, String y) //data base
    {
        String z = x+y;
        return z;
    }

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

Main.java

```
-----  
package com.ravi.method_overload;

public class Main
{
    public static void main(String[] args)
    {
        Sum s1 = new Sum();
        String add = s1.add("Data", "base");

        int x = s1.add(12, 12);

        double y = s1.add(12.89, 12.90);

        System.out.println(add+" : "+x+" : "+y);
    }
}
```

-----  
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.

---

Program that describes var args can hold 0 to n number of parameters :

---

2 files :

---

Test.java

---

```
package com.ravi.var_args;

public class Test
{
    public void input(int... x) //Array
    {
        System.out.println("Var args executed");
    }
}
```

Main.java

---

```
package com.ravi.var_args;

public class Main {

    public static void main(String ...x)
    {
        Test t1 = new Test();
        t1.input();
        t1.input(12);
        t1.input(15,19);
        t1.input(10,20,30);
        t1.input(10,20,30,40);
        t1.input(10,20,30,40,50);

    }
}
```

---

Program that describes how to add parameter values using var args :

2 Files :

---

Test.java

---

```
//add the parameter values using variable argument
package com.ravi.var_args1;

public class Test
{
```

```
public void acceptData(int ...values)
{
    int res=0;

    for(int value : values)
    {
        res = res + value;
    }
    System.out.println("Sum is parameter is :" +res);

}
}
```

Main.java

```
-----
package com.ravi.var_args1;

public class Main
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.acceptData();
        t1.acceptData(10,20);
        t1.acceptData(10,20,30);
        t1.acceptData(100,100,100,100);

    }
}
```

-----  
Program that describes var args must be only one and last argument.

Test.java

```
-----
package com.ravi.var_args2;

public class Test
{
    //All commented codes are invalid

    /*
     * public void accept(float ...x, int ...y) { }
     *
     * public void accept(int ...x, int y) { }
     *
     * public void accept(int...x, int ...y) {}
     */
}
```

```
public void accept(int x, int... y) // valid
{
    System.out.println("x value is :" +x);
```

```
        for (int z : y)
    {
        System.out.println(z);
    }
}
```

Main.java

```
-----
package com.ravi.var_args2;

public class Main {

    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.accept(100, 200, 300, 400, 500);
    }
}
```

-----  
Program that describes var-args may also accept Heterogeneous types of Elements.

2 Files :

-----  
Test.java

```
-----
package com.ravi.var_args3;

public class Test
{
    public void acceptHetro(Object ...obj)
    {
        for(Object o : obj)
        {
            System.out.println(o);
        }
    }
}
```

-----  
package com.ravi.var\_args3;

```
public class Main {

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

        t1.acceptHetro(true, 45.90, 12, 'A', new String("Ravi"));

    }
}
```

-----  
12-04-2024

-----  
Wrapper classes in java :

---

In java we have 8 primitive data types i.e byte, short, int, long, float, double, char and boolean.

Except these primitives, everything in java is an Object.

If we remove these 8 data types from java then Java will become pure Object Oriented language.

Wrapper class is a technique through which we can convert the primitives to corresponding object. Now it can be divided into two types from 1.5 onwards.

\*All the Wrapper classes are final, they have overridden equals(Object obj) and hashCode() as well as they are immutable  
(un-changed OR Un-modifiable)

- 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	-	Character
boolean	-	Boolean

---

Converting Primitive to Wrapper type :

---

Integer class has provided predefined static method valueOf(int x), it is responsible to convert the primitive type into wrapper object.

public static Integer valueOf(int x)

---

```
//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[])
    {
        int y = 12;
        Integer x = y;
        System.out.println(x);

        double e = 45.90;
        Double d = e;
        System.out.println(d);

        boolean a = true;
        Boolean b = a;
        System.out.println(b);
    }
}

```

#### Overloaded valueOf() method :

- 1) public static Integer valueOf(int x) :- Will convert the int into Integer class.
- 2) public static Integer valueOf(String x) :- Will convert the String into Integer class.
- 3) public static Integer valueOf(String x, int base/radix) :- Will convert the String into Integer class by using specified base.

Note :- We can pass base OR radix upto 36  
 i.e A to Z (26) + 0 to 9 (10) -> [26 + 10 = 36], It can be calculated by using Character.MAX\_RADIX.  
 Output will be generated on the basis of radix

```

//Integer.valueOf(String str)
//Integer.valueOf(String str, int radix/base)
public class AutoBoxing3
{
    public static void main(String[] args)
    {
        Integer a = Integer.valueOf(15);

        Integer b = Integer.valueOf("25");

        Integer c = Integer.valueOf("111",36); //Here Base we can take upto 36

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}

```

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

**Integer.parseInt(String str) :**

By using this statement, we can convert a string into integer by using base 10 (default base is 10).

We have an overloaded method of parseInt() which is as follows :

**Integer.parseInt(String str, int radix/base) :** By using this we can convert String into integer by using the given base.

```
public class Test  
{  
    public static void main(String [] args)  
    {  
        //How to convert String into integer  
        String str = "111";  
        int no = Integer.parseInt(str,2);  
        System.out.println(no);  
    }  
}
```

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

-----  
package com.ravi.variable;

```
-----  
public class AutoBoxing6 {  
  
    public static void main(String[] args)
```

```

{
    char ch = 'A';
    Character val = Character.valueOf(ch);
    System.out.println(val);

    boolean b = true;
    Boolean isConverted = Boolean.valueOf(b);
    System.out.println(isConverted);

}

```

---

15-04-2024

---

### **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
Character	-	char
Boolean	-	boolean

---

We have total 8 Wrapper classes.

Among all these 8, 6 Wrapper classes are the sub class of `java.lang.Number` class 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 i1 = 127;  
        Integer i2 = 127;  
        System.out.println(i1==i2); //t  
  
        Integer i3 = 128;  
        Integer i4 = 128;  
        System.out.println(i3==i4); //f  
  
        Integer i5 = 128;  
        Integer i6 = 128;  
        System.out.println(i5.equals(i6)); //t  
    }  
}
```

Note :- While comparing the Integer object we should always use equals(Object obj) method of Object class.

```
-----  
public class Test  
{  
    public static void main(String [] args)  
    {  
        Float f = 2F;  
  
        Double d = 2D;  
  
        byte b = 12; //Explicit type casting (Narrowing)  
  
        long l = 12; //Implicit Type Casting (Widening)  
    }  
}
```

Ambiguity issues while overloading a method :

Whenever we work with method overloading and if we have ambiguity while calling the method then compiler has provided 3 rules to work with Method overloading.

Rule 1:

-----  
Most specific type (Specific type will get more priority)

```
double > float  
float > long  
long > int  
int > short  
int > char  
short > byte
```

Rule 2 :

-----

While overloading a method, if we have an ambiguity issue then compiler provides the priority in the following order :

WAV [Widening -> Autoboxing -> Var args]

Rule 3 :

-----  
Nearest Type Rule

While Overloading a method if we are getting sub class reference in comparison to super class reference then sub class will be treated as Nearest type.

-----  
Programs :

```
package com.ravi.ambiguity;

class Test
{
    public void accept(byte b)
    {
        System.out.println("byte");
    }
    public void accept(short b)
    {
        System.out.println("short");
    }
}

public class AmbiguityDemo {

    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.accept(15); //Error
    }
}
```

Note :- 15 which is by default of type int, we cannot assign to byte or short directly.

```
-----  
package com.ravi.ambiguity;

class Test
{
    public void accept(byte b)
    {
        System.out.println("byte");
    }
    public void accept(short b)
    {
        System.out.println("short");
    }
}
```

```
public class AmbiguityDemo {  
    public static void main(String[] args)  
    {  
        Test t1 = new Test();  
        t1.accept((byte)15);  
        t1.accept((short)19);  
    }  
}
```

---

```
package com.ravi.ambiguity;
```

```
class Test  
{  
    public void accept(int b)  
    {  
        System.out.println("int");  
    }  
    public void accept(long b)  
    {  
        System.out.println("long");  
    }  
}
```

```
public class AmbiguityDemo {  
    public static void main(String[] args)  
    {  
        Test t1 = new Test();  
        t1.accept(15);  
    }  
}
```

Note :- Here int will be executed because int is the most specific type

---

```
package com.ravi.ambiguity;  
  
class Test  
{  
    public void accept(String b)  
    {  
        System.out.println("String");  
    }  
    public void accept(Object b)  
    {  
        System.out.println("object");  
    }  
}
```

```

public class AmbiguityDemo {

    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.accept("India"); t1.accept(null);

    }
}

```

Here in both the cases String will be executed because String is the sub class Object class

---

```

class Demo
{
    public void accept(String b)
    {
        System.out.println("String");
    }

    public void accept(Integer b)
    {
        System.out.println("Integer");
    }
}

public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept(null); //error
    }
}

```

Note :- Here Compiler is unable to find out which one is the nearest type because both are sub class of Object hence we will get compilation error

---

```

class Demo
{
    public void accept(Number b)
    {
        System.out.println("Number");
    }

    public void accept(Integer b)
    {
        System.out.println("Integer");
    }
}

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

```

```
        Demo d = new Demo();
        d.accept(null);
    }
}
```

Note :- Here Integer is the sub class of number so Integer is the nearest type

---

```
class Demo
{
    public void accept(int ...x)
    {
        System.out.println("int");
    }

    public void accept(float ...x)
    {
        System.out.println("float");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept();
    }
}
```

Note :- int will be executed becoz int is more specific type

---

```
class Test
{
    public void accept(double ...x)
    {
        System.out.println("double");
    }

    public void accept(float ...x)
    {
        System.out.println("float");
    }
}
```

```
public class AmbiguityDemo {

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

    }
}
```

Note :- float will be executed becoz float is more specific type

---

```
class Demo
{
    public void accept(int ...x)
    {
        System.out.println("int");
    }

    public void accept(boolean ...x)
    {
        System.out.println("boolean");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept();
    }
}
```

Here We will get Compilation error, because there is no comparison between boolean and int

---

```
class Demo
{
    public void accept(char ...x)
    {
        System.out.println("char");
    }

    public void accept(short ...x)
    {
        System.out.println("short");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept();
    }
}
```

Here We will get Compilation error, because there is no comparison between char and short

---

```
class Demo
{
    public void accept(char ...x)
    {
        System.out.println("char");
    }

    public void accept(int ...x)
```

```

        {
            System.out.println("int");
        }
    }
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept();
    }
}

```

Here char will be executed because char is more specific type

---

```

class Demo
{
    public void accept(float a, int b)
    {
        System.out.println("float - int");
    }

    public void accept(int a, float b)
    {
        System.out.println("int - float");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept(2,5);
    }
}

```

---

```

class Demo
{
    public void accept(long l)
    {
        System.out.println("long [Widening]");
    }

    public void accept(Integer i)
    {
        System.out.println("Integer [Autoboxing]");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept(15);
    }
}

```

```
}
```

Note :- Here Widening is having more priority than Autoboxing

---

```
class Demo
{
    public void accept(long l)
    {
        System.out.println("long [Widening]");
    }

    public void accept(int ...x)
    {
        System.out.println("int [var args]");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept(15);
    }
}
```

Note :- Here Widening is having more priority than var args

---

```
class Demo
{
    public void accept(Integer i)
    {
        System.out.println("int [Autoboxing]");
    }

    public void accept(int ...x)
    {
        System.out.println("int [var args]");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept(15);
    }
}
```

---

```
class Demo
{
    public void accept(char i)
    {
        System.out.println("char");
    }

    public void accept(int x)
```

```

        {
            System.out.println("int ");
        }
    }
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept(15);
    }
}

```

Here 15 is of int type so it is giving more priority to int

```

class Demo
{
    public void accept(Object i)
    {
        System.out.println("Object");
    }

    public void accept(Integer x)
    {
        System.out.println("Integer ");
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.accept("NIT");
    }
}

```

---

16-04-2024

---

**Method Signature = Method Name + Method Parameter**

\* **Method Overriding :**

---

Writing two or more methods in the super and sub class in such a way that method signature(method name along with method parameter) of both the methods must be same in the super and sub classes.

While working with method overriding generally we can't change the return type of the method but from JDK 1.5 onwards we can change the return type of the method in only one case that is known as Co-Variant.

Without inheritance method overriding is not possible that means if there is no inheritance there is no method overriding.

---

What is the advantage of Method Overriding ?

---

The advantage of Method Overriding is, each class is specifying its own specific behavior.

---

Upcasting :-

---

It is possible to assign sub class object to super class reference variable using dynamic polymorphism. It is known as Upcasting.

Example:- Animal a = new Dog(); //valid [upcasting]

Downcasting :

---

By default downcasting is not possible, Here we are trying to assign super class object to sub class reference variable but the same we can achieve by using explicit type casting. It is known as downcasting.

Eg:- Dog d = new Animal(); //Invalid

Dog d = (Dog) new Animal(); //Valid because Explicit type casting

But by using above statement (Downcasting) whenever we call a method we will get a runtime exception called java.lang.ClassCastException. [Animal cann't be cast to Dog]

---

17-04-2024

---

MethodOverridingDemo.java

---

Program on Method Overriding :

---

```
package com.ravi.method_overriding;

class Animal
{
    public void roam()
    {
        System.out.println("Generic Roaming");
    }
}

class Lion extends Animal
{
    public void roam()
    {
        System.out.println("Roaming in the forest");
    }
}

public class MethodOverridingDemo
{
    public static void main(String[] args)
    {
        Animal a = new Lion();
        a.roam();

    }
}
```

---

## Method Overriding by using Dynamic Method Dispatch :

---

```
package com.ravi.method_overriding;

class Bird
{
    public void fly()
    {
        System.out.println("Generic Bird is flying...");
    }
}
class Peacock extends Bird
{
    public void fly()
    {
        System.out.println("Peacock Bird is flying...");
    }
}
class Parrot extends Bird
{
    public void fly()
    {
        System.out.println("Parrot Bird is flying...");
    }
}

public class MethodOverridingDemo1
{
    public static void main(String[] args)
    {
        Bird b = null;

        b = new Peacock(); b.fly(); //Dynamic Method Dispatched
        b = new Parrot(); b.fly(); //Dynamic Method Dispatched
    }
}
```

---

## @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 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 comment 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.

## OverrideAnnotationDemo.java

---

```
package com.ravi.method_overriding;
```

```

class RBI
{
    public void loan()
    {
        System.out.println("Bank should provide loan");
    }
}

class SBI extends RBI
{
    @Override
    public void loan()
    {
        System.out.println("SBI Provides loan @ 9.2% ROI");
    }
}

class BOB extends RBI
{
    @Override
    public void loan()
    {
        System.out.println("BOB Provides loan @ 11.2% ROI");
    }
}

public class OverrideAnnotationDemo {
    public static void main(String[] args)
    {
        RBI r = null;
        r = new SBI(); r.loan();
        r = new BOB(); r.loan();
    }
}

```

Program that describes we cannot override private method:

private methods are available within the same class only, it is not visible to the child class so we cannot override. If we write @Override annotation then we will get compilation error

```

package com.ravi.method_overriding;

class Super
{
    private void show()
    {
        System.out.println("private Show method in the super class");
    }
}

class Sub extends Super
{
    @Override //error

```

```

public void show()
{
    System.out.println("Show method in the Sub class");
}
}

public class PrivateMethodDemo {

    public static void main(String[] args)
    {

    }
}

```

Note :- Private method of a super class is not available to sub class so we can't inherit hence we can't override.

---

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.

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 while overriding a method.

Note :- private access modifier is not available (visible) in sub class so it is not the part of method overriding.

```

package com.ravi.method_overriding;

class Shape
{
    public void draw()
    {
        System.out.println("Generic Draw");
    }
}
class Rectangle extends Shape
{
    @Override
    protected void draw() //error only public is applicable
    {
        System.out.println("Drawing Rectangle");
    }
}
class Square extends Shape
{
    @Override
    protected void draw() //error only public is applicable
    {
        System.out.println("Drawing Square");
    }
}

```

```

    }
}

public class ShapeDemo
{
    public static void main(String[] args)
    {
        Shape s = null;

        s = new Rectangle(); s.draw();
        s = new Square(); s.draw();
    }
}

```

Note :- In the above program we will get CE because we are reducing the visibility of the Method at the time of method Overriding.

---

18-04-2024

---

Co-Variant return type in Java :

---

In general we can't change the return type of method while overriding a method. if we try to change it will generate compilation error as shown in the program below.

CoVariant.java

---

```

class Super
{
    public void show()
    {
        System.out.println("Super class Show Method");
    }
}

class Sub extends Super
{
    @Override
    public int show() //error
    {
        System.out.println("Sub class Show Method");
        return 0;
    }
}

public class CoVariant
{
    public static void main(String args[])
    {
        Super s = new Sub(); //upcasting
        s.show();
    }
}

```

Note :- In the above program we will get compilation error because return type is not compatible 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 relation ship 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 show()
    {
        System.out.println("Super class Show Method");
        return new Alpha();
    }
}
class Sub extends Super
{
    @Override
    public Beta show()
    {
        System.out.println("Sub class Show Method");
        return new Beta();
    }
}
public class Test
{
    public static void main(String args[])
    {
        Super s = new Sub(); //upcasting
        s.show();
    }
}
```

Note :- In the above program, Method return type is different because both are co-variant(In the same Direction) i.e in inheritance relationship

---

```
package com.nit;

class Shape
{
    Object draw()
    {
        System.out.println("Object");
        return null;
    }
}
```

```

class Rectangle extends Shape
{
    @Override
    String draw()
    {
        System.out.println("String");
        return null;
    }
}
public class CoVariantDemo {

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

```

---

In Co-variance the final method return type is super class object but not sub class object otherwise we will get ClassCastException.

```

package com.nit;

class Car
{
    Car ride()
    {
        System.out.println("Riding the Car");
        return this;
    }
}
class BMW extends Car
{
    @Override
    BMW ride()
    {
        System.out.println("Riding the BMW");
        return this;
    }
}

public class CovariantDemo1
{
    public static void main(String[] args)
    {
        Car c = new BMW();
        //BMW b = c.ride(); It is invalid (ClassCastException)
        Car b = c.ride();
    }
}

```

Note :- In Co-variance always the return type must be super class

---

## Variable Shadow in Method Overriding :

---

If the super class variable name and sub class variable name both are same then it is known as variable shadow.

With the reference variable, whenever we call variable then always it will take the preference of class type so, in the below example a is a the reference variable of type Animal so Animal class variable will be invoked.

```
package com.nit;

class Animal
{
    String name = "Animal";

    public String roam()
    {
        return "Generic Animal is roaming";
    }
}

class Lion extends Animal
{
    String name = "Tiger"; //Variable Shadow

    @Override
    public String roam()
    {
        return "Lion is roaming";
    }
}

public class VariableShadow
{
    public static void main(String[] args)
    {
        Animal a = new Lion();
        System.out.println(a.name + ": " + a.roam());
    }
}
```

---

19-04-2024

---

\* Can we override static method ?

    OR

\* Can we override main method?

    OR

\* What is Method Hiding in java?

Point 1 :

---

If a super class contains any static method then by default the static method of super class is available to sub class as shown in the program.

```
package com.ravi.static_method;
```

```

class Super
{
    public static void m1()
    {
        System.out.println("M1 method of super class");
    }
}
class Sub extends Super
{
}

public class StaiticMethodPresence
{
    public static void main(String[] args)
    {
        Sub s1 = new Sub();
        s1.m1();

        Sub.m1();
    }
}

```

Point 2 :

-----  
We can't override static method with non static method and in the same way we can't override non static method with static method.

OverridingStaticMethod.java

```

-----
class Super
{
    public static void m1()
    {
    }
}
class Sub extends Super
{
    public void m1() //error
    {
    }
}

public class OverridingStaticMethod
{
    public static void main(String args[])
    {
    }
}

```

Note :- error, Overridden method is static

```

class Super
{
    public void m1()
    {
    }
}
class Sub extends Super
{
    public static void m1()
    {
    }
}

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

```

Note : error, Overriding method is static.

---

In java, we can't override static method because static method belongs to class, we can override non-static method because it belongs to Object.

If we are writing the static method in the sub class with same signature as it is already written in the super class then it looks like it is an overridden method but actually it is Method Hiding, It is not an overridden method.

We can't apply @Override annotation on static methods.

```

package com.ravi.static_method;

class Super
{
    public static void m1()
    {
        System.out.println("M1 method of super class");
    }
}
class Sub extends Super
{
    public static void m1()
    {
        System.out.println("M1 method of sub class");
    }
}

public class StaticMethodPresence
{
    public static void main(String[] args)
    {
        Super s1 = new Sub();
        s1.m1();
    }
}

```

```
}
```

```
}
```

Note :- In the above program, It looks like m1 is an overridden method but we can verify by using @Override annotation so now compilation error will be generated.

---

WAP to proof that whenever we write same static method in the sub class then it is method hiding but not method overriding

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

class Sub extends Super
{
    public static int m1() //Method Hiding [error]
    {
        return 0;
    }
}

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

error :- m1 method() in sub can't hide m1() method in super.

---

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)

---

To declare a class as a final :

---

Whenever we declare a class as a final class then we can'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 :- We will get an error, cannot inherit from final class A

---

```
final class Test
{
    private int data = 100;

    public void setData(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();
        t1.setData(200);
    }
}
```

Note :- For final class we can create the object and we can modify the data.

---

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()
    {
        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();
    }
}
```

---

```
class A
{
    protected int a = 10;
    protected int b = 20;

    final private void calculate()
    {
        int sum = a+b;
        System.out.println("Sum is :" +sum);
    }
}
class B extends A
{
    public void calculate()
    {
        int mul = a*b;
        System.out.println("Mul is :" +mul);
    }
}
public class FinalMethodEx1
{
    public static void main(String [] args)
    {
        new B().calculate();
    }
}
```

```
}
```

Above program will compile and execute.

Here Private method of super class is not visible to sub class so sub class can define its own method.

---

3) To declare a variable(field) as a final :(Re-assignment is not possible)

---

In older languages 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.

Some example of predefined final variables

Byte.MIN\_VALUE -> MIN\_VALUE is a static and final variable

Byte.MAX\_VALUE -> MAX\_VALUE is a static and final variable

Example:- final int DATA = 10; (Now we can not perform re-assignment )

```
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)
    {
        A a1 = new A();
        a1.setData();
    }
}
```

Note : final variables are initialized only one time by the user.

---

What is Blank final variable ?

---

If a final variable is not initialized at the time of declaration then it is called Blank final variable.

```
public class Test
{
    final int A; //Blank final variable
}
```

\* A blank final variable must be initialized by the user in the following 3 places :

- 1) At the time of declaration
- 2) Inside an instance block
- 3) Inside a constructor body.

If the blank final variable is not initialized by the user in the above 3 places then we will get compilation error.

\* A blank final variable can't be initialized by default constructor.

\* A blank final variable must be initialized in all the constructors by user (If we have more than one constructor).

---

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

Note :- We will get error, default constructor can't initialize the final variable A.

---

A blank final variable can also have default value as shown in the program.

```
public class Test
{
    final int A;

    {
        foo();
        A = 100;
    }

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

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

---

20-04-2024

---

Program to show that Blank final variable must be initialized by the user in all the constructors.

```

class Demo
{
    final int A; // blank final variable

    public Demo()
    {
        A = 15;
        System.out.println(A);
    }

    public Demo(int x)
    {
        A = x;
        System.out.println(A);
    }
}

public class BlankFinalVariable
{
    public static void main(String[] args)
    {
        Demo d1 = new Demo(); //d1 ---> A = 15

        Demo d2 = new Demo(8); //d2 ---> A = 8
    }
}

```

---

How to create our own Immutable(Un-changed) class in java ?

---

In order to create our own immutable class we should follow the following steps :

- 1) Declare the class as a final
- 2) Declare all the instance variable with private and final modifier.
- 3) The class should not contain any setter
- 4) We should declare reference variable as a final so we can't assign another object.

**ImmutableClass.java**

---

```

//Program to create an immutable(unchanged) class

final class Citizen
{
    private final String aadharNumber; //Blank final variable

    public Citizen(final String aadharNumber)
    {
        if(aadharNumber.length() == 12)
        {
            this.aadharNumber = aadharNumber;
        }
        else
        {

```

```

        this.aadharNumber = aadharNumber;
        System.err.println("Invalid Aadhar Number");
    }
}

public String getAadharNumber()
{
    return aadharNumber;
}
}
public class ImmutableClass
{
    public static void main(String ar[])
    {
        final Citizen ravi = new Citizen("987667897848");
        System.out.println("Ravi Aadhar Number is :" +ravi.getAadharNumber());
    }
}

```

---

**Object class and its Methods :**

---

**Working with Object class and its methods :**

---

There is a predefined class called Object available in java.lang package, this Object class is by default the super class of all the classes we have in java.

```

class Test
{
}

```

Note :- Object is the super class for this Test class. by default this Object class is super class so explicitly we need not to mention.

Since, Object is the super class of all the classes in java that means we can override the method of Object class (Except final methods) as well as we can use the methods of Object class anywhere in java because every class is sub class of Object class.

The Object class provides some common behavior to each sub class Object like we can compare two objects , we can create clone (duplicate) objects, we can print object properties(instance variable), providing a unique number to each and every object(hashCode()) and so on.

---

**What is Method Chaining in Java ?**

---

By using this concept we can call n number of method in a single statement.

The next method call always depends upon current method return type.

The final return type of the method will depend upon last method call.

---

**MethodChaining.java**

---

```

package com.nit.jar_demo;

```

```
public class MethodChaining
{
    public static void main(String[] args)
    {
        String s1 = "india";
        int length = s1.concat(" is great").toUpperCase().length();
        System.out.println(length);

        String s2 = "Hyderabad";
        char ch = s2.toLowerCase().charAt(0);
        System.out.println(ch);
    }
}
```

MethodChaning2.java

```
-----
package com.nit.jar_demo;

public class MethodChaning2 {

    public static void main(String[] args)
    {
        char ch = m1().charAt(1);
        System.out.println(ch);
    }

    public static String m1()
    {
        return "NIT";
    }
}
```

22-04-2024

-----  
public native final java.lang.Class getClass() :

-----  
It is a predefined method of Object class.

This method returns the runtime class of the object, the return type of this method is java.lang.Class.

This method will provide class keyword + Fully Qualified Name (package name + class name)

This getClass() method return type is java.lang.Class so further we can apply any other method of java.lang.Class class method.

GetClassDemo.java

```
-----
package com.ravi.object_class_methods;

class Test
{
```

```
}

public class GetClassDemo {

    public static void main(String[] args)
    {
        Test t1 = new Test();
        Class cls = t1.getClass();
        System.out.println(cls); //class keyword + FQN
        System.out.println(cls.getName()); //FQN(Fully Qualified name)
    }
}
```

---

2 Files (Both are in different packages)

Demo.java

---

```
package com.ravi.arraylist;
```

```
public class Demo
{
}
```

com.ravi.arraylist.Demo(Available in different Package)

```
package com.ravi.object_class_methods;
```

```
import com.ravi.arraylist.Demo;
```

```
public class com.ravi.arraylist.Demo {
```

```
    public static void main(String[] args)
    {
        Demo d1 = new Demo();
        System.out.println(d1.getClass().getName());
    }
}
```

---

```
public native int hashCode() :-
```

It is a predefined method of Object class.

Every Object contains a unique number generated by JVM at the time of Object creation is called hashCode.

we can find out the hashCode value of an Object by using hashCode() method of Object class, return type of this method is int.

---

```
package com.ravi.object_class;
```

```
class Demo
```

```
{  
}  
  
public class HashCodeDemo1 {  
  
    public static void main(String[] args)  
    {  
        Demo d1 = new Demo();  
        Demo d2 = new Demo();  
        Demo d3 = new Demo();  
        System.out.println(d1.hashCode());  
        System.out.println(d2.hashCode());  
        System.out.println(d3.hashCode());  
    }  
}
```

---

### 3) public String toString() :

---

It is a predefined method of Object class.

it returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object.

The result should be a concise but informative representation that is easy for a person to read

`toString()` method of Object class contains following logic.

```
public String toString()  
{  
    return getClass().getName()+" @ "+Integer.toHexString(hashCode());  
}
```

Please note internally the `toString()` method is calling the `hashCode()` and `getClass()` method of Object class and `getName()` method of `java.lang.Class` class.

In java whenever we print any Object reference by using `System.out.println()` then internally it will invoke the `toString()` method of Object class as shown in the following program.

---

```
package com.ravi.object_class;  
  
class Foo  
{  
}  
  
public class ToStringDemo  
{  
    public static void main(String[] args)  
    {  
        Foo f1 = new Foo();  
        System.out.println(f1.toString());  
    }  
}
```

```
}
```

---

```
package com.ravi.object_class;

class Foo
{
    @Override
    public String toString()
    {
        String str = super.toString();
        return "NIT "+str;
    }
}
```

```
public class ToStringDemo
{
    public static void main(String[] args)
    {
        Foo f1 = new Foo();
        System.out.println(f1.toString());

        Object obj = new Foo();
        System.out.println(obj);

    }
}
```

---

```
public boolean equals(Object obj) :
```

---

It is predefined method of Object class.

It is mainly used to compare two objects based on the memory address just like == operator because internally, It uses == operator only.

The following program explains how to use equals(Object obj) method for Customer comparison.

---

```
package com.ravi.object_class;

class Customer
{
    private int custId;
    private String custName;

    public Customer(int custId, String custName) {
        super();
        this.custId = custId;
        this.custName = custName;
    }
}
```

```
public class EqualsMethodDemo
```

```

{
    public static void main(String[] args)
    {
        Customer c1 = new Customer(111,"Scott");
        Customer c2 = new Customer(111,"Scott");

        System.out.println(c1==c2); //false
        System.out.println(c1.equals(c2)); //false

    }
}

```

Note :- Here in both the cases we will get false because Object class equals(Object obj) method internally uses == operator only.

---

In the above program c1 and c2 is having same content but due to memory address comparison it is providing false.

If we want to compare these two objects based the content but not based on the memory address then we should override equals(Object obj) method of Object class.

---

```

package com.ravi.object_class;

class Customer
{
    private int custId;
    private String custName;

    public Customer(int custId, String custName) {
        super();
        this.custId = custId;
        this.custName = custName;
    }

    //Overriding the equals(Object obj) method for content comparison

    @Override
    public boolean equals(Object obj) //obj = c2
    {
        //First Object Data
        int id1 = this.custId;
        String name1 = this.custName;

        Customer c2 = (Customer) obj; //Down casting
        //2nd Object data
        int id2 = c2.custId;
        String name2 = c2.custName;

        if(id1==id2 && name1.equals(name2))
        {
            return true;
        }
        else
        {

```

```

        return false;
    }
}

public class EqualsMethodDemo
{
    public static void main(String[] args)
    {
        Customer c1 = new Customer(111,"Scott");
        Customer c2 = new Customer(111,"Scott");

        System.out.println(c1==c2);
        System.out.println(c1.equals(c2)); // == Memory Address com

    }
}

```

Overriding equals(Object obj) for content comparison :

---

```

package com.ravi.object_class;

class Product
{
    private int id;
    private String name;

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

    @Override
    public boolean equals(Object obj)
    {
        Product p2 = (Product) obj; //Down Casting

        if(this.id == p2.id && this.name.equals(p2.name))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

public class EqualsDemo2 {

```

```
public static void main(String[] args)
{
    Product p1 = new Product(1, "Camera");
    Product p2 = new Product(1, "Camera");

    System.out.println(p1.equals(p2));

}
-----
```

23-04-2024

instanceof operator :

It is an operator as well as keyword.

It always return boolean value i.e true or false

It is mainly used to verify whether a reference variable is referring to a particular type of object or not.

Example :

```
Test t1 = new Test();

if(t1 instanceof Test)
{
    System.out.println("t1 is referring Test type of Object or not");
}
```

Here in between reference variable and class/interface we must have IS-A relation otherwise we will get compilation error.

```
package com.ravi.instance_demo;

public class InstanceDemo1
{
    public static void main(String[] args)
    {
        String s1 = "india";

        if(s1 instanceof String)
        {
            System.out.println("s1 is pointing to String object");
        }

        String s2 = new String("Hyd");

        if(s2 instanceof Object)
        {
            System.out.println("s2 is String type");
        }

        Integer i = 45;
```

```
if(i instanceof Number)
{
    System.out.println("i is Integer type");
}

}

-----
package com.ravi.instance_demo;

class A{}

class B extends A{}

class C extends B{}

class D{}

public class InstanceDemo1
{
    public static void main(String[] args)
    {
        A a1 = new A();
        B b1 = new B();
        C c1 = new C();
        D d1 = new D();

        if(d1 instanceof Object)
        {
        }
        if(c1 instanceof B)
        {

        }
        if(c1 instanceof A)
        {

        }
        if(b1 instanceof A)
        {

        }
        if(a1 instanceof Object)
        {

        }
    }
}
```

All the above statements are valid.

---

WAP to compare two different objects using equals(Object obj) method by implementing instanceof operator.

```
package com.ravi.instance_demo;

class Student
{
    private int id;
    private String name;

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

    @Override
    public boolean equals(Object obj)
    {
        if(obj instanceof Student)
        {
            Student s2 = (Student) obj;

            if(this.id==s2.id && this.name.equals(s2.name))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            System.err.println("Comparison not possible");
            return false;
        }
    }
}

class Employee
{
    private int id;
    private String name;

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

```
}

public class EqualsMethod
{
    public static void main(String[] args)
    {
        Student s1 = new Student(1,"Scott");
        Student s2 = new Student(2,"Smith");

        Employee e1 = new Employee(1,"Scott");

        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(e1));
        System.out.println(s1.equals(null));
    }
}
```

Note : instanceof operator will help us to compare only same type of object.

instanceof operator with null always returns false.

---

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 extends 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 modifier :

1) sealed : Can be extended only through permitted class.

2) non-sealed : Can be extended by any sub class, if a user wants to give permission to its sub classes

3) permits : We can provide permission to the sub classes, which are inheriting through Sealed class.

4) final : we can declare permitted sub class as final so, it cannot be extended further.

---

### SealedDemo.java

---

```
package com.ravi.instance_demo;

sealed class Bird permits Parrot,Peacock
{
    public void fly()
    {
        System.out.println("Generic Bird is flying");
    }
}
non-sealed class Parrot extends Bird
{
    public void fly()
    {
        System.out.println("Parrot Bird is flying");
    }
}

final class Peacock extends Bird
{
    public void fly()
    {
        System.out.println("Peacock is flying");
    }
}

public class SealedDemo
{
    public static void main(String[] args)
    {
        Bird b = null;
        b = new Parrot(); b.fly();
        b = new Peacock(); b.fly();
    }
}

-----  
package com.ravi.instance_demo;

sealed class OnlineClass permits Laptop, Mobile
{
    public void attendOnlineClass()
    {
        System.out.println("Online class");
    }
}
non-sealed class Laptop extends OnlineClass
{
    public void attendOnlineClass()
    {
        System.out.println("Attending Online class through Laptop");
    }
}
```

```
final class Mobile extends OnlineClass
{
    public void attendOnlineClass()
    {
        System.out.println("Attending Online class through Mobile");
    }
}

public class SealedDemo1 {

    public static void main(String[] args)
    {
        OnlineClass cls = null;

        cls = new Laptop(); cls.attendOnlineClass();

        cls = new Mobile(); cls.attendOnlineClass();

    }
}
```

---

24-04-2024

---

Record class in java :

---

public abstract class Record extends Object.

It is a new feature introduced from java 17.(In java 14 preview version)

As we know only objects are moving in the network from one place to another place so we need to write BLC class with nessacery requirements to make BLC class as a Data carrier class.

Records are immutable data carrier so, now with the help of record we can send our immutable data from one application to another application.

It is also known as DTO (Data transfer object) OR POJO 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 variable, static method and static block in record.

We can't define instance variable, instance block but instance method we can write.

We can't extend or inherit records because by default every record is implicitly final. It is extending from java.lang.Record 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 :

-----  
CustomerClass.java(C)

```
package com.ravi.record_demo;

import java.util.Objects;

public class CustomerClass {
    private int customerId;
    private String customerName;
    private double customerBill;

    public CustomerClass(int customerId, String customerName, double customerBill) {
        super();
        this.customerId = customerId;
        this.customerName = customerName;
        this.customerBill = customerBill;
    }

    public int getCustomerId() {
        return customerId;
    }

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

    public String getCustomerName() {
        return customerName;
    }

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

    public double getCustomerBill() {
        return customerBill;
    }

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

    @Override
    public String toString() {
        return "CustomerClass [customerId=" + customerId + ", customerName=" +
customerName + ", customerBill="
```

```

        + customerBill + "]";
    }

@Override
public int hashCode() {
    return Objects.hash(customerBill, customerId, customerName);
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    CustomerClass other = (CustomerClass) obj;
    return Double.doubleToLongBits(customerBill) ==
Double.doubleToLongBits(other.customerBill)
        && customerId == other.customerId &&
Objects.equals(customerName, other.customerName);
}
}

```

CustomerRecord.java(R)

---

```

package com.ravi.record_demo;

public record CustomerRecord(int custId, String custName, double custBill)
{
    //Compact Constructor
    public CustomerRecord
    {
        if(custId <=0)
        {
            System.err.println("Invalid Customer ID");
        }
    }
}

```

ELC.java(C)

---

```

package com.ravi.record_demo;

public class ELC
{
    public static void main(String[] args)
    {
        CustomerClass cc = new CustomerClass(1, "A", 1200);
        CustomerClass cc1 = new CustomerClass(1, "A", 1200);
        System.out.println(cc);
        cc.setCustomerName("Scott");
        System.out.println(cc.getCustomerName());
        System.out.println(cc.equals(cc1));
    }
}

```

```
System.out.println(".....");
CustomerRecord cr = new CustomerRecord( 2, "B", 1800);
CustomerRecord cr1 = new CustomerRecord(2, "B", 1800);
System.out.println(cr); //toString()
System.out.println(cr.custName()); //getter
System.out.println(cr.equals(cr1)); //equals
}
}
```

---

#### 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 complexity 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 compulsorily 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 achieve partial abstraction that means by using abstract classes we can achieve 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 abstract class as a final class.

---

ShapeDemo.java

---

abstract class Shape  
{

```
    public abstract void draw();
}

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

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

public class ShapeDemo
{
    public static void main(String[] args)
    {
        Shape s = null;

        s = new Rectangle(); s.draw();
        s = new Circle(); s.draw();
    }
}
```

---

Interview Question :

---

```
package com.ravi.iq;

abstract class Car
{
    protected int speed = 100;

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

    public void getDetails()
    {
        System.out.println("It has 4 wheels...");
    }

    public abstract void run();
}

class Honda extends Car
```

```

{
    @Override
    public void run()
    {
        System.out.println("Honda car is Running ");
    }
}
public class AbstractIQ
{
    public static void main(String[] args)
    {
        Car c = new Honda();
        System.out.println("Car Speed is :" + c.speed);
        c.getDetails();
        c.run();
    }
}

```

Note :- Inside an abstract class we can write a constructor and it will be executed by sub class through super keyword.

---

What is the advantage of writing Constructor in the abstract class

---

Even though we can't create an object for abstract class but we can write constructor inside the abstract class to initialize the properties of abstract class by using super keyword inside sub class.

Abstract class can have properties (state OR Data) and we can initialize also without object creation.

---

An abstract class may or may not have abstract method but an abstract method must have abstract class.

```
package com.ravi.iq;
```

```
public abstract class Demo
{
    public abstract void m1();
}
```

---

WAP to show that abstract method must be overridden in the sub class.

```
package com.ravi.iq;

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 AbstractDemo
{
    public static void main(String[] args)
    {
        Gamma g = new Gamma();
        g.show(); g.demo();
    }
}

```

Note :- All the abstract method of super class must be overridden in the sub classes otherwise sub class will become as an abstract class.

---

5 files :

---

Shape.java

---

```

package com.ravi.abstract_demo;

public abstract class Shape
{
    protected int data;

    public Shape(int data)
    {
        this.data = data;
    }

    public abstract void area();
}

```

Rectangle.java

---

```

package com.ravi.abstract_demo;

public class Rectangle extends Shape
{
    protected int breadth;
    public Rectangle(int length, int breadth)

```

```
{  
    super(length);  
    this.breadth = breadth;  
}  
  
@Override  
public void area()  
{  
    double areaOfRect = super.data * this.breadth;  
    System.out.println("Area of rect :" + areaOfRect);  
}  
}
```

### Circle.java

```
-----  
package com.ravi.abstract_demo;  
  
public class Circle extends Shape  
{  
    final double PI = 3.14;  
    public Circle(int radius)  
    {  
        super(radius);  
    }  
  
    @Override  
    public void area()  
    {  
        double areaOfCircle = PI * super.data * super.data;  
        System.out.println("Area of Circle is :" + areaOfCircle);  
    }  
}
```

### Square.java

```
-----  
package com.ravi.abstract_demo;  
  
public class Squre extends Shape  
{  
    public Squre(int side)  
    {  
        super(side);  
    }  
    @Override  
    public void area()  
    {  
        double areaOfSquare = super.data * super.data;  
        System.out.println("Area of Square is :" + areaOfSquare);  
    }  
}
```

### ShapeDemo.java

```
-----  
package com.ravi.abstract_demo;  
  
public class ShapeDemo {  
  
    public static void main(String[] args)  
    {  
        Shape s = null;  
        s = new Rectangle(4, 5); s.area();  
        s = new Circle(2); s.area();  
        s = new Square(15); s.area();  
    }  
}
```

-----  
Program on Array with abstract method :

```
-----  
package abstract_with_array;  
  
abstract class Animal  
{  
    public abstract void checkup();  
}  
class Lion extends Animal  
{  
    @Override  
    public void checkup()  
    {  
        System.out.println("Lion Checkup");  
    }  
}  
class Bird extends Animal  
{  
    @Override  
    public void checkup()  
    {  
        System.out.println("Bird Checkup");  
    }  
}  
class Dog extends Animal  
{  
    @Override  
    public void checkup()  
    {  
        System.out.println("Dog Checkup");  
    }  
}  
class AnimalCheckup  
{  
    public static void checkup(Animal animals[])
```

```
{  
    for(Animal animal : animals)  
    {  
        animal.checkup();  
    }  
}  
}  
public class AnimalDemo  
{  
    public static void main(String[] args)  
    {  
        Lion []lions = {new Lion(), new Lion(), new Lion()};  
  
        Bird []birds = {new Bird(), new Bird()};  
  
        Dog []dogs = {new Dog(), new Dog(), new Dog(), new Dog()};  
  
        AnimalCheckup.checkup(lions);  
        AnimalCheckup.checkup(birds);  
        AnimalCheckup.checkup(dogs);  
    }  
}
```

---

interface upto java 1.7

---

An interface is a keyword in java which is similar to a class. It defines working functionality of the class.

Upto JDK 1.7 an interface contains only abstract method that means there is a guarantee that inside an interface 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 achieve 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.

-----  
Program :

-----  
3 files :

-----  
Moveable.java(I)

-----  
package com.ravi.interface\_demo;

-----  
public interface Moveable  
{  
 int SPEED = 120; //public + static + final  
 void move(); //public + abstract

}

Car.java(C)

-----  
package com.ravi.interface\_demo;

-----  
public class Car implements Moveable  
{

@Override  
 public void move()  
 {  
 System.out.println("Car speed is :" + SPEED);  
 }

}

InterfaceDemo.java(C)

-----  
package com.ravi.interface\_demo;

-----  
public class InterfaceDemo {

public static void main(String[] args)  
 {  
 Moveable m = new Car();  
 m.move();  
 System.out.println("Speed of the Car is :" + Moveable.SPEED);

}

}

-----  
29-04-2024

-----  
package com.ravi.interface\_demo;

-----  
interface Calculator

{  
 void doSum(int x, int y);

```

        void doSub(int x, int y);
        void doMul(int x, int y);
    }

    class Calculate implements Calculator
    {

        @Override
        public void doSum(int x, int y)
        {
            int z = x + y;
            System.out.println("Sum is :" + z);
        }

        @Override
        public void doSub(int x, int y)
        {
            int z = x - y;
            System.out.println("Sub is :" + z);
        }

        @Override
        public void doMul(int x, int y)
        {
            int z = x * y;
            System.out.println("Mul is :" + z);
        }
    }
}

```

```

public class InterfaceDemo
{
    public static void main(String[] args)
    {
        Calculator c = new Calculate();
        c.doSum(12,36);
        c.doSub(200, 100);
        c.doMul(2, 9);
    }
}

```

---

Assignment :

Banking Application Program :

---

```

interface Bank
{
    void withdraw(int amount);
    void deposit(int amount);
}

```

---

Program that describes, we can achieve loosely coupled application using interface :

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.

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.

6 files :

-----  
HorDrink.java(I)

```
package com.ravi.loose_coupling;

public interface HotDrink
{
    void prepare();
}
```

-----  
Tea.java(C)

```
package com.ravi.loose_coupling;

public class Tea implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preparing Tea!!!");
    }
}
```

-----  
Coffee.java(C)

```
package com.ravi.loose_coupling;

public class Coffee implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preparing Coffee");
    }
}
```

-----  
Horlicks.java(C)

```
package com.ravi.loose_coupling;
```

```
public class Horlicks implements HotDrink {  
  
    @Override  
    public void prepare()  
    {  
        System.out.println("Preparing Horlicks");  
    }  
  
}
```

Restaurant.java(c)

```
-----  
package com.ravi.loose_coupling;  
  
public class Restaurant  
{  
    public static void prepareHotDrink(HotDrink hd) //loose coupling  
    {  
        hd.prepare();  
    }  
}
```

ELC.java(C)

```
-----  
package com.ravi.loose_coupling;  
  
public class ELC  
{  
    public static void main(String[] args)  
    {  
        Restaurant.prepareHotDrink(new Tea());  
        Restaurant.prepareHotDrink(new Coffee());  
        Restaurant.prepareHotDrink(new Horlicks());  
    }  
}
```

Method return 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.....  
}
```

Multiple Inheritance using interface :

Upto java 7, interface does not contain any method body that means all the methods are abstract method so we can achieve multiple inheritance by providing the logic in the implementer class as shown in the below program (Diagram 129-APR-24)

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.

```
package com.ravi.mi;

interface A
{
    void m1();
}

interface B
{
    void m1();
}

class Implementer implements A,B
{
    @Override
    public void m1()
    {
        System.out.println("Multiple Inheritance is Possible");
    }
}

public class MultipleInheritance
{
    public static void main(String[] args)
    {
        new Implementer().m1();
    }
}
```

---

Extending one interface to another interafce :

---

One interface can extends another interface, it cannot implement because interface cannot provide implementation for the abstract method.

ExtendingInterface.java

---

```
package com.ravi.mi;

interface Alpha
{
    void m1();
}

interface Beta extends Alpha
{
    void m2();
```

```
}

class Demo implements Beta
{
    @Override
    public void m1()
    {
        System.out.println("M1 method implemented");
    }

    @Override
    public void m2()
    {
        System.out.println("M2 method implemented");
    }
}

public class ExtendingInterface {

    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.m1(); d.m2();

    }
}
```

---

#### New Features of Java (interface from java 1.8 onwards)

---

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 inside an interface?

---

default method is just like concrete method which contains method body and we can write inside an interface from java 8 onwards.

default method is used to provide specific implementation for the implementer classes which are implementing from interface because we can override default method inside the sub classes to provide our own specific implementation.

\*By using default method there is no boundation to override the default method in the sub class, if we really required it then we can override to provide my own implementation.

by default, default method access modifier is public so at the time of overriding we should use public access modifier.

---

4 files :

---

Vehicle.java(I)

---

```
package com.ravi.abstractLimitation;
```

```
public interface Vehicle
```

```
{
```

```
    void run();
```

```
    void horn();
```

```
    default void digitalMeter() //java 8
```

```
{
```

```
    System.out.println("Digital Meter Facility");
```

```
}
```

```
}
```

Car.java(C)

---

```
package com.ravi.abstractLimitation;
```

```
public class Car implements Vehicle {
```

```
    @Override
```

```
    public void run()
```

```
{
```

```
    System.out.println("Car is Running");
```

```
}
```

```
    @Override
```

```
    public void horn()
```

```
{
```

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

```
}
```

```
    @Override
```

```
    public void digitalMeter()
```

```
{
```

```
    System.out.println("Car is having digital meter");
```

```
}
```

```
}
```

Bike.java(C)

---

```
package com.ravi.abstractLimitation;
```

```
public class Bike implements Vehicle {
```

```
@Override  
public void run()  
{  
    System.out.println("Bike is Running");  
}  
  
@Override  
public void horn()  
{  
    System.out.println("PEEP PEEP");  
}  
}
```

### AbstractMethodLimitation.java

---

```
package com.ravi.abstractLimitation;  
  
public class AbstractMethodLimitation  
{  
    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();  
    }  
}
```

---

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 implemented in all the sub classes.

---

The following program explains that default methods are having low priority than normal methods (Concrete Method). class is having more power than interface.

4 files :

---

### A.java(I)

---

```
package com.ravi.priority;  
  
public interface A  
{  
    default void m1()  
    {  
        System.out.println("m1 method of interface A");  
    }  
}
```

### B.java(C)

---

```
package com.ravi.priority;
```

```
public class B
{
    public void m1()
    {
        System.out.println("m1 method of class B");
    }
}
```

C.java(C)

```
-----
package com.ravi.priority;

public class C extends B implements A
{
}
```

Main.java(C)

```
-----
package com.ravi.priority;

public class Main {

    public static void main(String[] args)
    {
        C c1 = new C();
        c1.m1();
    }
}
```

Can we override Object class method using default method?

No, we cannot override object class method as a default method inside an interface.

```
interface Alpha
{
    public default String toString()
    {
        return "NIT";
    }
}
```

Here we will get compilation error because a default method cannot override method of Object class.

01-05-2024

Multiple inheritance using default method :

Multiple inheritance is possible in java by using default method inside an interface, here we need to use super keyword to differentiate the super interface methods.

## MultipleInheritance.java

---

```
package com.ravi.multiple_inheritance;

interface A
{
    public default void m1()
    {
        System.out.println("default method of interface A");
    }
}

interface B
{
    public default void m1()
    {
        System.out.println("default method of interface B");
    }
}

class C implements A,B
{
    @Override
    public void m1()
    {
        System.out.println("Overridden Method");
        A.super.m1();
        B.super.m1();
    }
}

public class MultipleInheritance
{
    public static void main(String[] args)
    {
        C c1 = new C();
        c1.m1();
    }
}
```

---

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 where it is defined that means we cannot invoke static method from the implementer classes with the help of implementer class.

It is used to provide common functionality which we can apply/invoke from any BLC/ELC class.

By default static method of an interface contains public access modifier.

---

//Program that describes we can write static method inside interface and it will be accessible from entire application.

```
package com.ravi.static_demo;

interface Drawable
{
    static void draw() //JDK 1.8 [By default public]
    {
        System.out.println("Drawing");
    }
}

public class StaticMethodDemo implements Drawable
{
    public static void main(String[] args)
    {
        //StaticMethodDemo.draw(); //Invalid

        //StaticMethodDemo s = new StaticMethodDemo();
        //s.draw(); //Invalid

        Drawable.draw();
    }
}
```

---

Program that describes static method of an interface is available to interface only, It can't be accessible by implementer classes

StaticMethodDemo.java

---

```
package com.ravi.static_demo;

interface Drawable
{
    public static void draw() //JDK 1.8
    {
        System.out.println("Drawing");
    }
}

public class StaticMethodDemo implements Drawable
{
    public static void main(String[] args)
    {
        //StaticMethodDemo.draw(); //Invalid

        //StaticMethodDemo s = new StaticMethodDemo();
        //s.draw(); //Invalid
    }
}
```

```
    Drawable.draw();  
  
}  
-----  
//Program that describes we can write main method inside an interface and it will be executed  
  
package com.ravi.static_demo;  
  
public interface Printable  
{  
    public static void main(String[] args)  
    {  
        System.out.println("interface Main method");  
    }  
}
```

#### Interface Static Method:

- a) Accessible using the interface name.
- b) Cannot be overridden 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.

```
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)  
    {  
        B.m1(); //valid  
        new B().m1(); //valid  
    }  
}
```

#### Anonymous inner class (Class without Name):

- 
- 1) The main purpose of anonymous inner class to extend a class or implement an interface.
  - 2) Anonymous inner class must be terminated with ;
  - 3) Anonymous inner class we can take inside a method which is known as local class extension.
- 

//Creating an anonymous inner class to override super class method

```
package com.ravi.anonymous_demo;

class Super
{
    public void m1()
    {
        System.out.println("Super class m1 method");
    }
}

public class AnonymousDemo1
{
    public static void main(String[] args)
    {
        //Anonymous inner class (Hidden class Or class without name)
        Super sub = new Super()
        {
            @Override
            public void m1()
            {
                System.out.println("Sub class m1 method");
            }
        };
        sub.m1();
    }
}
```

---

//Creating an anonymous inner class to override abstract class method.

```
package com.ravi.anonymous_demo;

abstract class Bird
{
    public abstract void fly();
}

public class AnonymousDemo2
{
```

```
public static void main(String[] args)
{
    Bird parrot = new Bird()
    {
        @Override
        public void fly()
        {
            System.out.println("Parrot is flying");
        }
    };
    parrot.fly();
}
```

---

```
//Creating an anonymous inner class to override interface Method
```

```
package com.ravi.anonymous_demo;
```

```
interface Vehicle
{
    void run();
}
```

```
public class AnonymousDemo3
{
    public static void main(String[] args)
    {
        Vehicle car = new Vehicle()
        {
            @Override
            public void run()
            {
                System.out.println("Car is Running");
            }
        };

        Vehicle bike = new Vehicle()
        {
            @Override
            public void run()
            {
                System.out.println("Bike is Running");
            }
        };

        car.run(); bike.run();
    }
}
```

---

```
02-05-2024
```

---

```
What is Functional interface in java ?
```

---

A functional interface is an interface which contains exactly one abstract method.

It may contain 'n' number of default and static method but it must contain only one abstract method. (SAM => Single abstract Method)

It can be represented by @FunctionalInterface annotation.

FunctionalInterface.java

---

```
package com.ravi.functionalinterface;

@FunctionalInterface
interface Student
{
    void writeExam();

}

public class FunctionalInterface
{
    public static void main(String[] args)
    {
        Student science = new Student()
        {
            @Override
            public void writeExam()
            {
                System.out.println("Science Student Exam..");
            }
        };

        Student commerce = new Student()
        {
            @Override
            public void writeExam()
            {
                System.out.println("Commerce Student Exam..");
            }
        };

        Student arts = new Student()
        {
            @Override
            public void writeExam()
            {
                System.out.println("Arts Student Exam..");
            }
        };
    }

    science.writeExam(); commerce.writeExam();
}
```

```
        arts.writeExam();  
    }  
}
```

---

### Lambda Expression :

---

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.

---

### Lambda1.java

---

```
package com.ravi.lambda;  
@FunctionalInterface  
interface Printable  
{  
    void print();  
}  
  
public class Lambda1  
{  
    public static void main(String[] args)  
    {  
        Printable p = ()->  
        {  
            System.out.println("Printing");  
        };  
        p.print();  
    }  
}
```

---

### Lambda2.java

---

```
package com.ravi.lambda;  
  
@FunctionalInterface  
interface Calculate  
{  
    void doSum(int x, int y);  
}
```

```
}
```

```
public class Lambda2
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Calculate c = (a, b) -> System.out.println(a+b);
```

```
        c.doSum(12, 12);
```

```
    }
```

```
}
```

---

```
Lambda3.java
```

---

```
package com.ravi.lambda;
```

```
@FunctionalInterface
```

```
interface Length
```

```
{
```

```
    public int getLength(String str);
```

```
}
```

```
public class Lambda3
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Length l = str -> str.length();
```

```
        System.out.println(l.getLength("India"));
```

```
    }
```

```
}
```

---

```
@FunctionalInterface
```

```
interface Calculate
```

```
{
```

```
    int getSquare(int num);
```

```
}
```

```
public class Lambda
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Calculate c = x -> x*x;
```

```
        System.out.println("Square is :" + c.getSquare(4));
```

```
    }
```

```
}
```

---

```
@FunctionalInterface
```

```
interface Calculate
```

```
{
```

```
    void add(int a, int b, double c);
```

```
}

public class Lambda2
{
    public static void main(String[] args)
    {
        Calculate calc = (p, q, r) -> System.out.println("Sum is :" +(p+q+r));

        calc.add(12,89,67.90);
    }
}

-----  
import java.util.Scanner;

@FunctionalInterface
interface Length
{
    int getLength(String str);
}

public class Lambda3
{
    public static void main(String[] args)
    {
        Length l = str -> str.length();

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your Name :");
        String name = sc.next();
        System.out.println("Your Name length is :" +l.getLength(name));
    }
}
```

What is type parameter in java ?

It is a technique through which we can make our application independent of data type. It is represented by <T>

In java we can pass Wrapper classes as well as User-defined class to this type parameter.

We cannot pass any primitive type to this type parameter.

TypeParameter.java

```
-----  
package com.ravi.expr;  
  
class Accept<T>
{
    private T a;  
  
    public Accept(T a) //Employee a
    {
        super();
        this.a = a;
    }
}
```

```
public T getA()
{
    return a;
}

public class TypeParameter
{
    public static void main(String[] args)
    {
        Accept<Integer> intType = new Accept<Integer>(12);
        System.out.println(intType.getA());

        Accept<Boolean> boolType = new Accept<Boolean>(true);
        System.out.println(boolType.getA());

        Accept<Float> floatType = new Accept<Float>(2.3f);
        System.out.println(floatType.getA());

        Accept<Employee> empType = new Accept<Employee>(new Employee(1, "A"));
        System.out.println(empType.getA());
    }
}
```

```
record Employee(int id, String name)
{}
```

---

03-05-2024

---

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>
- 2) Consumer<T>
- 3) Function<T,R>
- 4) Supplier<T>
- 5) BiPredicate<T,U>
- 6) BiConsumer<T, U>
- 7) BiFunction<T,U,R>

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<T> functional interface :

---

```
package com.ravi.functiona_interface;  
  
import java.util.Scanner;  
import java.util.function.Predicate;  
  
public class PredicateDemo1 {  
  
    public static void main(String[] args)  
    {  
        //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(num +" is even ??"+p1.test(num));  
        sc.close();  
    }  
}  
  
-----  
package com.ravi.functiona_interface;  
  
import java.util.Scanner;  
import java.util.function.Predicate;  
  
public class PredicateDemo2 {  
  
    public static void main(String[] args)  
    {  
        //Verify my name is Ravi or not?  
  
        Predicate<String> p2 = str ->str.equals("Ravi");  
  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter your Name :");  
        String name = sc.nextLine();
```

```
        System.out.println("Are you Ravi :" + p2.test(name));
        sc.close();
    }

-----
package com.ravi.functiona_interface;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo3
{
    public static void main(String[] args)
    {
        //A name starts with A character or not
        Predicate<String> p3 = str -> str.startsWith("A");

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your Name :");
        String name = sc.nextLine();

        System.out.println(name + " starts with A ?: " + p3.test(name));
        sc.close();
    }
}
```

Assignment :

By using Predicate verify whether a year is leap year or not?

By using Predicate verify whether a person is eligible for vote or not?

Consumer<T> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It contains an abstract method accept() and returns nothing. It is used to accept the parameter value or consume the value.

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T x);
}
```

```
package com.ravi.functiona_interface;

import java.util.function.Consumer;
```

```

record Product(int pid, String pname)
{
}

public class ConsumerDemo1 {

    public static void main(String[] args)
    {
        Consumer<Integer> c1 = num -> System.out.println(num);
        c1.accept(12);

        Consumer<String> c2 = str -> System.out.println(str);
        c2.accept("NIT");

        Consumer<Double> c3 = num -> System.out.println(num);
        c3.accept(12.89);

        Consumer<Product> c4 = p1 -> System.out.println(p1);
        c4.accept(new Product(111, "Laptop"));
    }
}

```

---

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.functiona\_interface;

import java.util.function.Function;

public class FunctionDemo1 {

```

    public static void main(String[] args)
    {
        //Finding the Square of the number
        Function<Integer, Integer> fn1 = x -> x*x;
        System.out.println("Square is :" +fn1.apply(5));

    }

```

```
}
```

---

```
package com.ravi.functiona_interface;
```

```
import java.util.function.Function;
```

```
public class FunctionDemo2 {
```

```
    public static void main(String[] args)
```

```
    {
```

```
        //Length of my name
```

```
        Function<String, Integer> fn2 = str -> str.length();
```

```
        System.out.println("Length is :" + fn2.apply("NIT"));
```

```
        System.out.println("Length is :" + fn2.apply("Hyderabad"));
```

```
        //Verify my name is Ravi or not
```

```
        Function<String, Boolean> fn3 = str -> str.equals("Ravi");
```

```
        System.out.println("Are you Ravi :" + fn3.apply("Ravi"));
```

```
    }
```

```
}
```

---

04-05-2024

---

Supplier<T> predefined 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 a value of type T.

```
@FunctionalInterface
```

```
public interface Supplier<T>
```

```
{
```

```
    T get();
```

```
}
```

SupplierDemo1.java

---

```
package com.ravi.supplier_demo;
```

```
import java.util.function.Supplier;
```

```
public class SupplierDemo1 {
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Supplier<String> str = () -> "1" + 2 + 3;
```

```
        System.out.println(str.get());
```

```
    }
```

```
}
```

## SupplierDemo.java

---

```
package com.ravi.supplier_demo;

import java.util.Scanner;
import java.util.function.Supplier;

record Employee(int eid, String ename)

{
}

public class SupplierDemo
{
    public static void main(String[] args)
    {
        Supplier<Employee> s1 = ()->
        {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter Employee id :");
            int id = sc.nextInt();
            System.out.print("Enter Employee Name :");
            String name = sc.nextLine();
            name = sc.nextLine();
            Employee e1 = new Employee(id,name);
            sc.close();
            return e1;
        };

        Employee obj = s1.get();
        System.out.println(obj);
    }
}
```

---

## 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 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

---

```
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.

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);

```

```
}
```

---

06-05-2024

---

Interface from java 9v version

---

Yes, From java 9 onwards we can also write private static and private non-static methods inside an interface.

These private methods will improve code re-usability inside interfaces.

For example, if two default methods needed to share common and confidential code, a private method would allow them to do so, but without exposing that private method to its implementing classes.

Using private methods in interfaces have four rules :

- 1) private interface method cannot be abstract.
  - 2) private method can be used only inside interface.
  - 3) private static method can be used inside other static and non-static interface methods.
  - 4) private non-static methods cannot be used inside private static methods.
- 

3 files :

---

Drawable.java(l)

---

package com.ravi.interface\_9;

```
public interface Drawable
{
    public abstract void m1();

    public default void m2()
    {
        m4();
        m5();
    }

    public static void m3()
    {
        System.out.println("static method of interface");
        m4();
    }

    private static void m4()
    {
        System.out.println("private static method");
    }

    private void m5()
    {
        System.out.println("private non-static method");
    }
}
```

```
}
```

Draw.java(C)

```
-----  
package com.ravi.interface_9;  
  
public class Draw implements Drawable  
{  
    @Override  
    public void m1()  
    {  
        System.out.println("m1 method overridden");  
    }  
}
```

ELC.java(C)

```
-----  
package com.ravi.interface_9;  
  
public class ELC  
{  
    public static void main(String[] args)  
    {  
        Draw w = new Draw();  
        w.m1();  
        w.m2();  
        System.out.println(".....");  
        Drawable.m3();  
    }  
}
```

\*What is marker interface in java ?

```
-----  
An interface which does not contain any method and field is called marker interface. In other words, an empty interface is known as marker interface or tag interface.
```

```
*It describes run-time type information about objects, so the JVM have additional information about the object. [like object is clonable OR object is serializable OR Object is RandomAccess]
```

Example :

```
-----  
public interface Drawable //Marker interface  
{  
}
```

Note :-In java we have Clonable, Serializable and RandomAccess are predefined marker interface.

```
interface Cloneable //Marker interface
{
}

class Customer implements Cloneable
{
}

Cloneable c = new Customer();

if(c instanceof Customer)
{
    //then perform the cloning information.
    [calling clone() method on Customer object]
}
```

---

#### Does An Interface extends Object Class In Java.?

---

No Interface does not inherits Object class, but it provide accessibility to all methods of Object class.

This is because, for every public method in Object class, there is an implicit abstract and public method declared in every interface which does not have direct super interfaces. (Java language Specification 9.2 about interface members)

---

```
package com.ravi.obj_class_method;

interface Callable
{
}

public class ObjectMethodDemo1
{
    public static void main(String[] args)
    {
        Callable c = null;
        c.equals(null);
        c.hashCode();
        c.toString();
    }
}
```

---

```
package com.ravi.obj_class_method;

@FunctionalInterface
public interface Printable
{
    void m1();
```

```
public String toString();
public int hashCode();
public boolean equals(Object obj);
//public Class getClass(); //Invalid because final
}
```

---

```
package com.ravi.obj_class_method;
```

```
public interface Printable
{
    @Override
    public String toString();

    @Override
    public int hashCode();

    @Override
    public boolean equals(Object obj);

}
```

---

```
***
```

---

```
****What is difference between abstract class and interface ?
```

---

The following are the differences between abstract class and interface.

- 1) An abstract class can contain instance variables but interface variables are by default public , static and final.
- 2) An abstract class can have state (properties) of an object but interface can't have state of an object.
- 3) An abstract class can contain constructor but inside an interface we can't define constructor.
- 4) An abstract class can contain instance and static blocks but inside an interface we can't define any blocks.
- 5) Abstract class can't refer Lambda expression but using Functional interface we can refer Lambda Expression.
- 6) We can write concrete method inside an abstract class but inside an interface we can't write concrete public method, only abstract , default, static and private methods are allowed.

---

```
JVM Architecture with Class loader sub system :
```

---

The three main components of JVM

- 1) class loader sub system
- 2) Runtime Data Areas(Memory Areas)

### 3) Execution engine

class loader sub system internally performs 3 tasks

- a) Loading    b) Linking    c) Initialization (Diagram 6th May)
- 

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 to load the required .class file from java API that means all the predefined classes (provided by java software people) .class file will be loaded by Bootstrap class loader.

It is the super class of Extension class loader as well as It has the highest priority among all the class loader.

It will load the .class file from the following Path  
C-> Program files -> java -> jdk -> lib -> jrt-fs.jar

Extension/Platform class Loader :-

---

It is responsible to load the required .class files from ext (extension) folder. Inside the extension folder we have jar file(Java level zip file) given by some third party or user defined jar file.

It is the super class of Application class loader as well as It has more priority than Application class loader.

Note :- Command to create the jar file

```
jar cf NIT.jar FileName.class
```

Here FileName.class will be placed inside the jar file.

Note :- How to compile all the .java file in a single command

```
javac *.java (It will compile all the .java files)
```

Application/System class Loader :-

---

It is responsible to load the required .class file from class path level i.e Environment variable. It has lowest priority as well as It is the sub class of Extension/Platform class loader.

## 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.jar) and then by pass the request to extension class loader, Extension class loader will load the .class file from ext folder(\*.jar) and by pass the request 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 required .class file then JVM will generate an exception i.e java.lang.ClassNotFoundException

---

The following program explains that java.lang.Class can hold any .class file.

```
package com.ravi.method_area;

class Employee{}
class Customer{}
class Student{}

public class MethodArea
{
    public static void main(String[] args)
    {
        Class cls = null;
        cls = Employee.class;
        System.out.println(cls.getName());

        cls = Student.class;
        System.out.println(cls.getName());

        cls = Customer.class;
        System.out.println(cls.getName());
    }
}
```

---

WAP in java that describes our user defined .class file is loaded by Application class loader.

```
public class Manager
{
    public static void main(String[] args)
    {
        ClassLoader loader = Manager.class.getClassLoader();
        System.out.println(loader);
    }
}
```

`getClassLoader()` is a predefined method of class called `Class` available in `java.lang` package and its return type is `ClassLoader`.

---

WAP to show that Platform class loader is the super class for Application class loader.

```
package com.ravi.method_area;

public class Manager
{
    public static void main(String[] args)
    {
        ClassLoader loader = Manager.class.getClassLoader();

        System.out.println(loader.getParent());
    }
}
```

Note : `getClassLoader()` method return types is `ClassLoader` class which is abstract class and it contains `getParent()` method whose return type is again `ClassLoader` only

---

Bootstrap class implementation is not provided by java software people because it is used for loading the predefined .class file (For internal use only)

```
package com.ravi.method_area;

public class Manager
{
    public static void main(String[] args)
    {
        ClassLoader loader = Manager.class.getClassLoader();

        System.out.println(loader.getParent().getParent());
    }
}
```

Output is : null

---

08-05-2024

---

Linking :

---

In Linking phase we have 3 modules :

- a) Verify
- b) Prepare
- c) Resolve

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 an exception 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.

---

prepare: (static variable memory allocation + Initialization )

---

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 and now it will initialize with default value i.e 0.

---

Resolve :-

---

All the symbolic references will be converted to direct references or actual reference.

javap -verbose FileName.class

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

---

Initialization :-

---

In Initialization, all the static data member will get their actual (Original) value as well as if any static block is present in the class then the static block will start executing from here.

---

static block :

---

Static Block in Java :

---

It is a very special block which will be automatically executed at the time of loading the .class file into JVM memory.

Example :

```
static
{
}
```

Static blocks are executed only once because class loading is possible only once.

The main purpose of static block to initialize the static data member of the class so it is also known static initializer.

If a class contains multiple static blocks then it will be executed according to the order.(Top to bottom)

Automatically static block will not be executed everytime, whenever class will be loaded (user request) then only static block will be executed.

static block will be executed before main method or any static method.

The compiler will generate illegal forward reference, if a user try to access the static variable value from static block without declaration.

[Without declaration of static variable, we can initialize the static variable inside static block (because memory is already allocated in the prepare phase) but accessing of static variable is not possible directly but we can acces by using class name]

---

//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 ");
    }
}

```

Note :- From the above program it is clear that static block will not be executed everytime, It will be executed when class will be loaded and class loading depends upon user request.

---

```

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(Test.x);  
}  
}
```

Note :- If we have more than one static block in the class then  
It will be executed according to the order.(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 :- From the above program it is clear that static variables are initialized with default value.

---

blank static final variable :

---

If a final static variable does not initialized at the time of declaration then it is called blank static final variable.

```
class Test  
{  
    final static int A; //Blank static final variable  
}
```

A blank static final variable must be initialized at the time of declaration or inside the static block only.

A blank static final variable also have default values.

//Program that describes blank final variable also have defult values.

```
package com.ravi.blank_final;  
  
class Test  
{  
    public static final int A;  
  
    static  
    {  
        m1();  
    }  
}
```

```

        A = 100;
    }
    public static void m1()
    {
        System.out.println(A);
    }

}

public class BlankFinalDemo
{
    public static void main(String[] args)
    {
        Test.m1();
        System.out.println(Test.A);

    }
}

-----
class Demo
{
    final static int a ;      //Blank final variable

    static
    {
        a = 100;
        System.out.println(a);
    }
}

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

```

Note :- blank final variable must be initialized either at the time of declaration OR inside static block only.

---

IQ

---

```

-- 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();
    }
}

```

Note :- At the time of class loading, first of all super class will loaded then only sub class will be loaded.

---

//illegal forward reference

```

class Demo
{
    static
    {
        i = 100;

    }

    static int i;
}

public class StaticBlockDemo5
{
    public static void main(String[] args)
    {

```

```
        System.out.println(Demo.i);
    }
}
```

Note :- From the above program it is clear that we can initialize [Performing write operation] static variable inside static block without early declaration.

---

```
class Demo
{
    static
    {
        i = 100; //Initialization is possible
        System.out.println(i); //Invalid Illegal Forward Reference
        System.out.println(Demo.i); //valid
    }

    static int i;
}

public class StaticBlockDemo6
{

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

Note : With static variable we can't perform read opartion

without pre declaration in the static block but with the help of class name we can perform read operation also otherwise compiler will generarte Illegeal Forward Reference.

---

We can't write return keyword inside static and non static block

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

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

---

Can we execute a Java program without main method ?

---

We can't execute a java program without main method, Upto jdk 1.6 it was possible to execute a java program without main method by writing the static block.

From JDK 1.7 onwards now we can't execute java program without main method because JVM verifies the presence of the main method before initializing the class.

If main method method is not available then JVM will generate a message that main method not found in the particular class.

Eg:-

```
class WithoutMain
{
    static
    {
        System.out.println("Hello world");
        System.exit(0);
    }
}
```

The above program was possible to execute upto JDK 1.6.

---

How many ways we can load the .class file into JVM memory :

---

There are multiple ways to load the .class file into JVM memory.The following are the common examples :-

- 1) By using java tools [java command]
- 2) By using Constructor [Object creation]
- 3) By accessing the static member of the class.
- 4) By using Inheritance
- 5) By Reflection API

- 1) By using Java tools

```
javac Test.java
java Test [Load the Test.class file into JVM memoy]
```

- 2) By using Constructor [Object creation]

- 3) By Calling static variable and static method using class name.

```
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)
    {
        //new Demo();
        System.out.println(Demo.x);
    }
}
```

---

#### 4) By using Inheritance :

---

In java whenever we try to load sub class then first of super class will be loaded that is the reason Object is the first class to be loaded into the JVM memory.

```
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();
    }
}
```

---

#### 5) By Reflection API (Explicit class loading)

---

Java software people has provided a predefined class called "Class" available in `java.lang` package.

This class called `Class` contains a predefined static method `forName(String className)`, through which we can load the required .class file into JVM memory dynamically.

It throws a checked exception i.e `java.lang.ClassNotFoundException`

```
Class.forName("com.ravi.Test"); //Fully Qualified Name
```

---

```
package com.ravi.loading;

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

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

```
        Class.forName("com.ravi.loading.Sample");

    }

}
```

---

What is factory Method in java ?

---

The method which returns the class name itself by creating the object for that particular class is called Factory Method.

```
Class cls = Class.forName("com.ravi.Test");
```

Here Test.class will be loaded into JVM memory and it will return Class class object so further we can call any method of java.lang.Class class.

```
public class FactoryMethod {

    public static void main(String[] args) throws Exception
    {
        Class name = Class.forName("java.lang.String");
        System.out.println(name.getName());

    }
}
```

Any method which is returning the class object then it is known as Factory Method.

---

10-05-2024

---

\*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.

\*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.

The limitation with new keyword is, It demands the class name at the beginning 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.

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.

forName(String className) is a factory method so it returns java.lang.Class so, further we can call any other method of java.lang.Class.

```
-----  
class Student{}  
  
class Employee{}  
  
class Customer{}  
  
public class DynamicObjectCreation  
{  
    public static void main(String[] args) throws Exception  
    {  
        Object obj = Class.forName(args[0]).newInstance();  
        System.out.println("Object created for :" + obj.getClass().getName());  
    }  
}  
-----
```

```
javac DynamicObjectCreation.java  
java DynamicObjectCreation Student
```

```
-----  
class Manager  
{  
    public void show()  
    {  
        System.out.println("Show Method of manager class");  
    }  
}
```

```
-----  
public class DynamicObjectCreation  
{  
    public static void main(String[] args) throws Exception  
    {  
        Object obj = Class.forName(args[0]).newInstance();  
        Manager m = (Manager) obj; //downcasting  
        m.show();  
    }  
}
```

```
javac DynamicObjectCreation.java  
java DynamicObjectCreation Manager
```

```
-----  
* What is the difference between java.lang.ClassNotFoundException and  
java.lang.NoClassDefFoundError
```

```
java.lang.ClassNotFoundException :-
```

```
-----  
It encounters when we try to load the required .class file at runtime by using Class.forName()  
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 check  
whether the required .class file is available or not.
```

```
class Foo
{
    static
    {
        System.out.println("static block gets executed...");
    }
}
public class ClassNotFoundExceptionDemo
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("Player");
    }
}
```

---

**java.lang.NoClassDefFoundError :**

---

It encounters when the class was present at the time of COMPILED but at runtime the required .class file is not available(manually deleted by user ) Or it is not available in the current directory (folder misplaced) then we will get an exception i.e java.lang.NoClassDefFoundError.

```
class Message
{
    public void greet()
    {
        System.out.println("Hello Everyone!!!");
    }
}
public class NoClassDefFoundErrorDemo
{
    public static void main(String[] args)
    {
        Message m = new Message();
        m.greet();
    }
}
```

**Note :-** After compilation delete Message.class file

So the conclusion is :

java.lang.ClassNotFoundException does not have any concern at compilation time where as java.lang.NoClassDefFoundError was checking the class name at the time of compilation.

---

**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. When JVM will shutdown then during the shutdown phase class will be un-loaded 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.

When object is eligible for GC then object is destroyed and all the non static data members are also destroyed with corresponding object. It has lower life in comparison to static data members because they belongs to object.

### 3) Local Variable

---

Memory allocation done at stack area (Stack Frame) and user is responsible to initialize the variable before use. Once method execution is over, It will be deleted from stack Frame hence 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 method execution is over, It will be deleted from stack Frame hence it has shortest life.

---

## Runtime Data Areas :

---

Runtime Data Areas are also known as Memory Area. It is divided into 5 sections

- 1) Method Area
- 2) Heap Area
- 3) Stack Area
- 4) PC Register (Program Counter Register)
- 5) Native Method Stack

### Method Area :

---

Whenever a class is loaded then the class is dumped 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.

---

14-05-2024

---

Program to Show From Method Area we can get complete information of the class. (Reflection API)

### 2 files :

---

Test.java

---

```
package com.ravi.method_area;
```

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

```
int y = 200;
int z = 300;
int a = 900;

public void display() {}

public void input() {}

public void show() {}

public static void accept() {}
}
```

ClassDescription.java

```
-----
package com.ravi.method_area;

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

public class ClassDescription {

    public static void main(String[] args) throws Exception
    {
        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();
        int count = 0;

        for(Method method : methods)
        {
            System.out.println("Method Name is :" +method.getName());
            count++;
        }
        System.out.println("Total methods are :" +count);

        Field[] fields = cls.getDeclaredFields();
        count = 0;
        for(Field field : fields)
        {
            System.out.println("Field Name is :" +field.getName());
            count++;
        }

        System.out.println("Total Fields are :" +count);

    }
}
```

Note :- `getDeclaredMethods()` is a predefined non static method available in `java.lang.Class` class , the return type of this method is `Method` array where `Method` is a predefined class available in `java.lang.reflect` sub package

`getDeclaredFields()` is a predefined method available in `java.lang.Class` class , the return type of this method is `Field` array where `Method` is a predefined class available in `java.lang.reflect` sub package

\*`getDeclaredConstructors()` is a predefined method available in `java.lang.Class` class , the return type of this method is `Constructor` array where `Method` is a predefined class available in `java.lang.reflect` sub package.

---

HEAP Area :

---

It is a special area where we can store the objects as well as all properties.

We have only one HEAP AREA per JVM.

This HEAP area is sharable by Method Area. From java9 version onwards now we have PermGen memory so this heap area can be dynamically growable so now Method area is also the part of HEAP area in a `class_data` section.

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
- 2) Frame Data
- 3) Operand Stack.

We have multiple stack area for a single JVM.

JVM creates a separate runtime stack for every thread.

---

HEAP and STACK diagram :

---

HEAP and STACK Diagram for `Test.java`

---

```
class Test
{
int x; //1
int y; //2

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();
Test t2=new Test();
t1.m1(t2);
System.out.println(t1.x+"... "+t1.y); //1 ... 2
System.out.println(t2.x+"... "+t2.y); //3 ... 4

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

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

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

```

Assignment to Complete remaining modules :

---

HEAP and STACK Diagram for Employee.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 object 2000x and 4000x are eligible for GC]

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

    public static void update(Employee e)

```

```

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

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

```

---

### 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;
    }
}

```

---

-----  
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

-----  
In java, JVM is an interpreter which executes the program line by line. JVM (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 exception at line number 9 and after solving the exception 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 OR frequently used instruction as well as native method information like method signature, variables and make it available to JVM at the time of execution so the overall execution becomes very fast.

-----  
Exception Handling :

-----  
What is an exception ?

-----  
An exception is a runtime error.

An exception is an abnormal situation or un-expected situation in a normal execution flow.

An exception encounter due to dependency, if one part of the program is dependent to another part then there might be a chance of getting Exception.

AN EXCEPTION ALSO ENCOUNTER DUE TO WRONG INPUT GIVEN BY THE USER.

-----  
Exception Hierarchy :

-----  
This Exception hierarchy is available in the diagram (Exception\_Hierarchy.png)

Note :- As a developer we are responsible to handle the Exception. System admin is responsible to handle the error because we cannot recover from error.

## Exception Criteria :

---

### 1) java.lang.ArithmaticException

whenever we divide a number by zero (an int value) then we will get  
java.lang.ArithmaticException

### 2) java.lang.ArrayIndeOutOfBoundsException

Whenever we try to access the array index which is not available, out of the bound then we will get

java.lang.ArrayIndeOutOfBoundsException

```
int []arr = {90, 78};  
System.out.println(arr[2]);
```

### 3) java.lang.NumberFormatException

If we try to convert String into integer but the String is not in a nemonic format then we will get  
java.lang.NumberFormatException

```
String str = "NIT";  
int no = Integer.parseInt(str);
```

### 4) java.lang.NullPointerException

If we try to call a non-static method on null then It will generate the exception  
java.lang.NullPointerException

```
String str = null;  
int length = str.length();
```

Note :- for static method we will not get NullPointerException

### 5) java.lang.NegativeArraySizeException

The size of the array must not be negative value.

```
int []arr = new int[-10];
```

### 6) java.util.InputMismatchException

If we take the input but the input is not in a proper format  
then we will get java.util.InputMismatchException

```
Scanner sc = new Scanner(System.in);  
System.out.println("Enter Your Age :");  
int age = sc.nextInt(); //input is eleven  
System.out.println("Age is :" + age);
```

---

WAP that describes Exception is the super class of all the exceptions we have in java.

```
package com.ravi.exception;

public class ExceptionDemo {

    public static void main(String[] args)
    {
        Exception e1 = new ArithmeticException();
        System.out.println(e1);

        Exception e2 = new ArithmeticException("Divide By Zero");
        System.out.println(e2);

        Exception e3 = new NullPointerException("Reference is null");
        System.out.println(e3);

    }

}
```

---

WAP that describes that whenever an exception encounter in the program then program will be terminated in the middle.

```
package com.ravi.exception;

import java.util.Scanner;

public class ExceptionArrival {

    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; //if y is 0 then Program
                           //will halt

        System.out.println("Result is :" + result);

        System.out.println("Main Method Ended...");
        sc.close();
    }

}
```

In the above program if we provide value of y as zero then program will terminate in the middle which is known as abnormal

termination.

JVM has default exception handler which will handle this program and generate Exception message.

---

OOP has provided some mechanism to work with exception which are as follows :

- 1) try block
- 2) catch block
- 3) finally block
- 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, 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 and 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 exception encounter then with the help of JVM, TRY BLOCK WILL CREATE APPROPRIATE EXCEPTION OBJECT, AND THROW THIS EXCEPTION OBJECT to the nearest catch block.

After the exception 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 execute if there is an exception in the try block.

WAP that describes if we use Exception handling mechanism then our program will not be terminated abnormally.

```
package com.ravi.exception;
```

```
import java.util.Scanner;
```

```
public class TryDemo {
```

```
    public static void main(String[] args)
```

```

{
    System.out.println("Main method started");
    try
    {
        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("Try block ended");
        sc.close();
    }
    catch(Exception e)
    {
        System.err.println("Inside Catch");
        System.err.println(e);
    }
    System.out.println("Main method Ended");
}
}

```

Note :- The above program is in the protection of try-catch so, even we have an exception (y=0) but program will be terminated normally.

---

```

package com.ravi.exception;

public class ThrowableDemo
{
    public static void main(String[] args)
    {
        try
        {
            //int x = 10/0;
            throw new ArithmeticException("Dividing By zero");
        }
        catch(Exception e)
        {
            System.err.println(e);
        }
    }
}

```

The above program describes how to create and throw an exception object explicitly. try block is doing the same implicitly.

---

```

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

```

```

    {
        throw new OutOfMemoryError();
    }
    catch (Throwable e)
    {
        System.err.println(e);
    }
    System.out.println("Main method Ended");
}

}

```

---

The main purpose of Exception Handling to provide user-friendly message to the client.

```

package com.ravi.exception;

import java.util.Scanner;

public class CustomerDemo
{
    public static void main(String[] args)
    {
        System.out.println("Hello Client, Welcome to my application");
        try
        {
            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);
            sc.close();
        }
        catch(Exception e)
        {
            System.err.println("Please don't put zero here");
        }

        System.out.println("Thank you 4 visiting my application");
    }
}

```

---

17-05-2024

---

Throwable class method :

---

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 message, exception class location, exception method name and exception line number.

3) public String toString() :- It will convert the exception into String representation.

---

```
package com.ravi.basic;

public class PrintStackTrace
{
    public static void main(String[] args)
    {
        System.out.println("Main method started...");
        try
        {
            String x = "Ravi";
            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.out.println(e.getMessage()); //only for Exception message
            System.out.println(".....");
            System.out.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, NullPointerException 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)
        {
            System.err.println("Input is not in proper format");
        }
        sc.close();
        System.out.println("Main ended");
    }
}

```

---

```

package com.ravi.basic;

public class SpecificException1
{
    public static void main(String[] args)
    {
        try
        {
            throw new OutOfMemoryError();
        }
        catch(Error e)
        {
            System.out.println("Inside catch block");
            System.out.println(e);
        }
    }
}

```

---

Working with Infinity and Not a number(NaN) :

---

10/0 -> Infinity (Java.lang.ArithmaticException)

10/0.0 -> Infinity (POSITIVE\_INFINITY)

-10/0.0 -> Infinity (NEGATIVE\_INFINITY)

0/0 -> Undefined (Java.lang.ArithmaticException)

0/0.0 -> Undefined (NaN)

While working with Integral literal in both the cases i.e Infinity (10/0) and Undefined (0/0) we will get java.lang.ArithmaticException because java software people has not provided any final, static variable support to deal with Infinity and Undefined so java.lang.ArithmaticException and program will terminate in the middle.

On the other hand while working 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

---

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 exceptions.

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!!!!");

}

---

\* Single try with multiple catch block :

---

According to industry standard we should write try with multiple catch block so we can provide proper information for each and every exception.

While working with multiple catch block always the super class catch block must be last catch block.

From java 1.7 this multiple catch block we can also represent by using | symbol.

If try block is having more than one exception then always try block will handle 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 = 10/0;
            System.out.println("c value is :" + c);

            int []x = {12,78,56};
            System.out.println(x[3]);
        }

        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());

    String str2 = "Ravi";
    int x = Integer.parseInt(str2);
    System.out.println("Number is :" + x);
}
catch(NumberFormatException | NullPointerException e)
{
    e.printStackTrace();
}

System.out.println("Main method ended!!");

}


```

---

18-05-2024

---

**finally block :**

---

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 resources 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 exception, on the other hand if we use try-catch and finally then exception and resources 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");
    }
}
```

Here we have an exception in the try block but finally block will be executed.

---

```
package com.ravi.basic;

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

        try
        {
            int []x = new int[-2];      //We can't pass negative size of an array in negative
            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!!!");
    }
}
```

---

Limitation of finally block :

---

The following are the limitations of finally block :

- 1) Developer is responsible to close the resources manually.
- 2) Due to finally block the length of the code will be increased.
- 3) While using finally block we should declare all our resources outside of the try block otherwise the resources will become block level variable.

```
package com.ravi.abstract_demo;

import java.io.Closeable;
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 Employee id :");
    int id = sc.nextInt();
    System.out.println("Your Id is :" + id);

}
catch(InputMismatchException e)
{
    System.err.println("Input is not in a proper format");
}
finally
{
    System.out.println("Inside finally block");
    sc.close();
}

}

```

try with resources :

To avoid all the limitation of finally block, Java software people introduced a separate concept i.e try with resources from java 1.7 onwards.

Case 1:

```

try(resource1 ; resource2) //Only the resources will be handled
{
}

```

Case 2 :

```

//Resources and Exception both will be handled
try(resource1 ; resource2)
{
}
catch(Exception e)
{
}

```

Case 3 :

```

//Resources and Exception both will be handled
Resource resource1 = new Resource();

try(resource1) //java 9 onwards
{
}
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 resource class as part of try with resources then that class must implements either `Closeable` or `AutoCloseable` interface so, try with resources will automatically call the respective class `close()` method even an exception is encountered in the try block.

```
try(ResourceClass rc = new ResourceClass())  
{  
}  
catch(Exception e)  
{  
}
```

//This `ResourceClass` must implements either `Closeable` or `AutoCloseable` interface so, try block will automatically call the `close()` method.

The following program explains how try block is invoking the `close()` method available in `DatabaseResource` class and `FileResource` class.

`DatabaseResource.java`

```
-----  
package com.ravi.resource;  
  
public class DatabaseResource implements AutoCloseable  
{  
    @Override  
    public void close() throws Exception  
    {  
        System.out.println("Database Resource closed");  
    }  
}
```

`FileResource.java`

```
-----  
package com.ravi.resource;  
  
import java.io.Closeable;  
import java.io.IOException;  
  
public class FileResource implements Closeable  
{  
    @Override  
    public void close() throws IOException  
    {  
        System.out.println("File resource closed");  
    }  
}
```

TryWithResouese.java

```
-----  
package com.ravi.resource;  
  
public class TryWithResouese {  
  
    public static void main(String[] args) throws Exception  
    {  
        DatabaseResource dr = new DatabaseResource();  
        FileResource fr = new FileResource();  
  
        try(dr ; fr)  
        {  
            System.out.println(10/0);  
        }  
        catch(ArithmeticException e)  
        {  
            System.err.println("Divide by zero");  
        }  
    }  
}
```

}

Note :- In the parameter of try, we have passed two resource classes references i.e DatabaseResource and FileResource.

Now try block is responsible to call the close() automatically even we have an exception in the try block.

WAP to automatically close the resource by using Scanner class.

```
package com.ravi.resource;
```

```
import java.util.InputMismatchException;  
import java.util.Scanner;
```

```
public class TryWithResource1 {
```

```

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

    try(sc)
    {
        System.out.println("Enter your Student number :");
        int sno = sc.nextInt();
        System.out.println("Your roll number is :" + sno);

    }
    catch(InputMismatchException e)
    {
        System.err.println("Input is not in a proper format");
    }
}

```

Note :- Scanner class internally implementing Closeable interface so it is providing auto closing facility, as a user we need to pass the reference of Scanner class inside try with resources try()

---

20-05-2024

---

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");
}
}
}
}

```

Note :- We can write Nested try but inner try catch will execute after validation of outer try block.

---

Writing try-catch inside catch block :

---

We can write try-catch inside catch block but this try-catch block will be executed if the catch block will execute 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");
    }
}

}

```

---

### 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. (Unreachable)

Always finally block return statement having more priority than 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;
        }
        catch (Exception e)
        {
            System.out.println("catch block");
            return 20; //return statement is compulsory
        }
    }
}

```

---

```

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
        }
    }
}

-----
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 = 12;
            System.out.println(x);
        }
        catch(Exception e)
        {
            x = 15;
            System.out.println(x);
        }
        System.out.println("Main completed!!!");
    }
}
```

---

\*\*Difference between Checked Exception and Unchecked Exception :

---

Checked Exception :

---

In java some exceptions are very common exceptions are called Checked exception 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.

All the checked exceptions are directly sub class of `java.lang.Exception`

Eg:

---

`FileNotFoundException`, `IOException`, `InterruptedException`, `ClassNotFoundException`, `SQLException`, `CloneNotSupportedException` and so on

Unchecked Exception :-

---

The exceptions which are rarely occurred in java and for these kinds of exception 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`

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

Unchecked Exception :

---

- 1) Rare Exception
- 2) Compiler 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 our resource)

---

When to provide try-catch or declare the method as throws :-

---

We should provide try-catch if we want to handle the exception by own as well as if we want to provide user-defined messages to the client but on the other hand we should declare the method as throws when we are not interested to handle the exception and try to send it to the JVM for handling purpose.

Note :- It is always better to use try catch so we can provide appropriate user defined messages to our client.

---

\*\*\* What is the difference between throw and throws :

---

throw :

---

In case of predefined exception, try block is responsible to create the exception object and throw the exception object to the nearest catch block but it works with predefined exception only.

If a user wants to throw an exception based on his own requirement and specification by using userdefined exception then we should write throw keyword to throw the user defined exception object explicitly. (throw new InvalidMarksException())

**THROWING THE EXCEPTION OBJECT EXPLICITLY.**

throws :-

---

In case of checked Exception if a user is not interested to handle the exception and wants to throw the exception to JVM, wants to skip from the current situation then we should declare the method as throws.

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

---

Steps to create userdefined exception :

---

In order to create user defined exception we should follow the following steps.

1) A userdefined exception class must extends either Exception(Checked Exception) Or RuntimeException(Unchecked Exception) as a super class.

a) If our userdefined class extends RuntimeException that means we are creating UncheckedException.

b) If our userdefined class extends Exception that means we are creating checkedException and exception handling is compulsory here.

2) The userdefined class must contain No argument constructor as well as parameterized constructor(in case we want to pass some userdefined error message).

We should take No argument constructor if we don't want to send any error message where as we should take parameterized constructor with super keyword if we want to send the message to the super class.

3) We should use throw keyword to throw the Exception object explicitly.

---

Program on user-defined Exception to work with Checked Exception

---

CustomCheckException.java

```

-----
package com.ravi.custom_exception;

import java.util.Scanner;

@SuppressWarnings("serial")
class InvalidAgeException extends Exception //Checked Exception
{
    public InvalidAgeException()
    {

    }

    public InvalidAgeException(String errorMessage)
    {
        super(errorMessage);
    }
}

public class CustomCheckException
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        try(sc)
        {
            System.out.print("Please Enter your Age :");
            int age = sc.nextInt();
            if(age < 18)
            {
                throw new InvalidAgeException("Age is Invalid");
            }
            else
            {
                System.out.println("Welcome to Voting Application");
            }
        }
        catch(InvalidAgeException e)
        {
            System.err.println("You are not eligible 4 voting");
            System.out.println(e.getMessage());
            System.out.println(e);
        }
        System.out.println("Program completed");
    }
}

```

IQ) When we have already if condition with error message then why we should create user-defined exception

We should always prefer user-defined exception over if condition due to the following 2 reasons :

1) In user-defined exception we can throw exception object which will contain all exception related information.

2) There is no abnormal termination, in case we are dealing with resource with checked Exception.

---

Program on userdefined unchecked Exception :

---

```
package com.ravi.custom_exception;

import java.util.Scanner;

@SuppressWarnings("serial")
class GreaterMarksException extends RuntimeException //Unchecked
{
    public GreaterMarksException()
    {

    }

    public GreaterMarksException(String errorMessage)
    {
        super(errorMessage);
    }
}

public class CustomUncheckedException
{

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        try(sc)
        {
            System.out.print("Enter your marks :");
            int marks = sc.nextInt();

            if(marks > 100)
            {
                throw new GreaterMarksException("Invalid marks");
            }
            else
            {
                System.out.println("Your Marks is :" +marks);
            }
        }
        catch(GreaterMarksException e)
        {
            System.err.println("Marks must be less or equal to 100");
            System.err.println(e);
            System.out.println(e.getMessage());
        }
    }

    System.out.println("Main method completed");
}
```

```
}
```

```
}
```

---

Some important points to remember :

---

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 :-

```
import java.io.*;
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            //try block is not throwing checked Exception
        }
        catch (InterruptedException e)
        {
        }

    }
}
```

In the catch block we are hanling checked exception without throwing from try block so compilation error

Note :- The above rule is not applicable for Unchecked Exception

```
try
{
}
catch(ArithmaticException e) //Valid
{
    e.printStackTrace();
}
```

---

b) If the try block does not throw any exception then in the corresponding catch block we can write Exception, Throwable because both are the super classes for all types of Exception whether it is checked or unchecked.

```
package com.ravi.method_related_rule;

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

        try
    }
```

```

        //throw new IOException();
    }
    catch(Exception e)
    {
        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.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 IOException [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");
    }
}

```

---

#### Rules for Method call regarding Checked Exception :

---

\* Whenever we call a method and if the method is throwing/reporting Checked Exception then the caller method must have either try catch or throws otherwise code will not compile.

```

public class Test
{
    public static void main(String[] args)
    {
        m1();
    }

    public static void m1() throws InterruptedException
    {
    }
}

```

Here m1() method throwing a checked Exception so the caller method must have either try-catch or declare the method as throws.

---

We can't write return keyword inside the initializer i.e inside non static block OR static block because initializer must be completed normally.

We can't write throw keyword without catch block inside an initializer because it will be abnormal termination.

```

public class Test
{
    static
    {
        throw new ArithmeticException(); //Abnormal termination
    }

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

```

---

Exception propagation :- [Exception object will shift from callee to caller]

---

Whenever we call a method and if the callee method contains any kind of exception and if callee method doesn't contain any kind of exception handling mechanism (try-catch) 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.

---

```

package com.ravi.method_related_rule;

public class ExceptionPropagationDemo {

    public static void main(String[] args)
    {
        System.out.println("Main method started!!!");
        try
        {
            m1();
        }
        catch(ArithmeticException e)
        {
            System.out.println("Handled in main method");
        }
        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(10/0);
    }
}
```

---

Input Output in java :

---

In order to work with input and output concept, java software people has provided a separate package called java.io package.

By using this java.io package we can read the data from the user, creating file, reading/writing the data from the file and so on.

How to take the input from the user using java.io package :

---

Scanner class is available from java 1.5 onwards but before 1.5, In order to read the data we were using the following two classes which are available in java.io package.

- 1) DataInputStream (Deprecated)
- 2) BufferedReader

How to create the object :

---

DataInputStream :

---

```
DataInputStream d = new DataInputStream(System.in);
```

BufferedReader :

---

It provides more faster technique because it internally stores the data in a buffer and it is always recommended to read the data from the buffer.

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
```

OR

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Working with Methods :

---

1) public int read() :- It is used to read a single character from the source and return the UNICODE value of the character.

If the data is not available from the source then it will return -1.

2) public String readLine() :- It is used to read multiple characters or complete line from the source.  
The  
return type of this method is String.

---

WAP to read the name from the keyboard :

---

DataInputStream

---

```
import java.io.*;
public class ReadName
{
    public static void main(String[] args) throws IOException
    {
        DataInputStream dis = new DataInputStream(System.in);
        System.out.print("Enter your Name :");
        String name = dis.readLine();
        System.out.println("Your name is :" + name);
    }
}
```

BufferedReader

```
-----
import java.io.*;
public class ReadName
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter your Name :");
        String name = br.readLine();
        System.out.println("Your name is :" + name);
    }
}
```

-----  
WAP to read age from the user :

```
-----
import java.io.*;
public class ReadName
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter your Age :");
        String ag = br.readLine();
        System.out.println("Your Age is :" + ag);

        int age = Integer.parseInt(ag);
        if(age > 18)
        {
            System.out.println("Go for a Movie");
        }
        else
        {
            System.out.println("try after some year");
        }
    }
}
```

-----  
23-05-2024

WAP to read the salary from the keyboard.

```
package com.ravi.input_output;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ReadSalary {

    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter the Salary :");
        String sal = br.readLine();
        float salary = Float.parseFloat(sal);

        System.out.println("Your Salary is :" + salary);
    }
}
```

---

WAP to read gender from the keyword :

```
package com.ravi.input_output;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ReadGender {

    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter the Gender :");
        char gen = (char) br.read();
        System.out.println("Your Geneder is :" + gen);
    }
}
```

---

WAP to read Employee Data from the keyword :

---

```
package com.ravi.input_output;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EmployeeData {

    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
System.out.print("Enter your Employee Id :");
int id = Integer.parseInt(br.readLine());

System.out.print("Enter your Employee gender :");
//char gender = (char) br.read(); //Buffer Problem
char gender = br.readLine().charAt(0);

System.out.print("Enter your Employee Name :");
String name = br.readLine();

System.out.println("Employee id is :" + id);
System.out.println("Employee Gender is :" + gender);
System.out.println("Employee Name is :" + name);
}

}
```

---

#### File Handling :

---

##### What is the need of File Handling ?

---

As we know variables are used to store some meaningful value in our program but once the execution of the program is over, now we can't get those values so to hold those values permanently in our memory we use files.

Files are stored in the secondary storage devices so, we can use/read the data stored in the file anytime according to our requirement.

In order to work with File system java software people has provided number of predefined classes like File, FileInputStream, FileOutputStream and so on. All these classes are available in java.io package. We can read and write the data in the form of Stream.

---

#### Streams in java :

---

A Stream is nothing but flow of data or flow of characters to both the end.  
Stream is divided into two categories

##### 1) byte oriented Stream :-

---

It used to handle characters, images, audio and video file in binary format.

##### 2) character oriented Stream :-

---

It is used to handle the data in the form of characters or text.

Now byte oriented or binary Stream can be categorized as "InputStream" and "OutputStream".  
input streams are used to read or receive the data where as output streams are used to write or send the data.

Again Character oriented Stream is divided into Reader and Writer. Reader is used to read() the data from the file where as Writer is used to write the data to the file.

All Streams are represented by classes in java.io package.

InputStream is the super class for all kind of input operation where as OutputStream is the super class for all kind of output Operation for byte oriented stream.

Where as Reader is the super class for all kind reading operation where as Writer is the super class for all kind of writing operation in character oriented Stream.

---

File :

-----  
File :-

-----  
It is a predefined class in java.io package through which we can create file and directory. By using this class we can verify whether the file is existing or not.

```
File f = new File("abc.txt");
```

The above statement will not create any file, It actually create the file object and perform one of the following two task.

- a) If abc.txt does not exist, It will not create it
- b) if abc.txt does exist, the new file object will be refer to the referenec variable f

Now if the file does not exist then to create the file and verify whether file is existing or not, we should use the following two methods :

- 1) public boolean exists() : Will verify whether file is existing or not and based on that, It will true or false.
- 2) public boolean createNewFile() : Will Create a new file only if file is not available, if file is already available then return false.

File class has also a predefined method called getName(), to get the name of the file.

---

```
import java.io.*;  
public class File0  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            File f = new File("C:\\\\new\\\\Batch30.txt");  
  
            if(f.exists())  
            {  
                System.out.println("File is existing");  
            }  
            else  
            {  
                System.out.println("File is not existing");  
            }  
  
            if (f.createNewFile())  
            {  
                System.out.println("File created: " + f.getName());  
            }  
            else  
            {  
                System.out.println("File already exists");  
            }  
        }  
    }  
}
```

```
        System.out.println("File is already existing....");
    }
}
catch (IOException e)
{
    System.err.println(e);
}
}
```

---

Assignment :

---

Create a folder (directory) by using File class.

---

FileOutputStream :

---

It is a predefined class available in java.io package.

It is used to create the file and write the data to the file, if the file is already available then it will replace/override the existing file.

It comes under binary stream so data must be available in binary format.

How to convert the String data into binary format :

---

String class has provided a predefined method getBytes() through which we can convert String data into binary format.

Example :

---

```
public class ConvertToBinary {

    public static void main(String[] args)
    {
        String str = "ABCDEF";

        byte[] b = str.getBytes();

        for(byte c : b)
        {
            System.out.println(c); //65 66 67 68..70
        }

    }
}
```

---

//Creating and writing the data to the file

```
import java.io.*;
public class File1
{
    public static void main(String args[]) throws IOException
```

```

{
    var fout = new FileOutputStream("C:\\new\\India.txt");
try(fout)
{
    String s = "Hyd is a popular IT City in India ";
    byte b[] = s.getBytes();

    fout.write(b);

    System.out.println("Success....");
}
catch(Exception e)
{
    e.printStackTrace();
}
}

```

---

24-05-2024

---

**FileInputStream :-**

---

It is a predefined class available in java.io package. It is used to read the file data/content. If we want to print the file data in console then data must be available in char format.

Note :- Whenever we want to write the data in the file then data must be available in byte format where as If we want to print the data to the console then the data must be converted into char format.

```

//Reading tha data from the file
import java.io.*;
public class File2
{
    public static void main(String s[]) throws IOException
    {
        var fin = new FileInputStream("C:\\new\\Employee.txt");

        try(fin)
        {
            int i = 0;
            while(true)
            {
                i = fin.read();
                if(i==-1)
                    break;
                System.out.print((char)i);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println();
    }
}
```

```
    }
}

-----
```

//wap in java to read the data from one file and to write the data to another file.

```
import java.io.*;
public class File3
{
    public static void main(String s[]) throws IOException
    {
        //Outside of try (Java 9 enhancement in try with resource)
        var fin = new FileInputStream("File2.java");

        var fout = new FileOutputStream("C:\\new\\f2.java");

        try(fin; fout)
        {
            while(true)
            {
                int i = fin.read();
                if(i==-1) break;
                System.out.print((char)i);
                fout.write((byte)i);
            }
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

#### Limitation of FileInputStream class :

As we know FileInputStream class is used to read the content from the file but it can read the data from a single file only that means if we want to read the data from two files at the same time then we should use a separate Stream called SequenceInputStream.

#### SequenceInputStream :

It is a predefined class available in java.io package. This class is used to read the data from two files at the same time.

```
//Reading the data from two files and writing the data to a single file
import java.io.*;
public class File5
{
    public static void main(String x[]) throws IOException
    {
        var f1 = new FileInputStream("File3.java");
        var f2 = new FileInputStream("File4.java");

        var fout = new FileOutputStream("C:\\new\\FileData.txt");

        var s = new SequenceInputStream(f1,f2);
```

```

int i;
    try(f1; f2; fout; s)
    {
while(true)
{
    i = s.read();
    if(i==-1)
        break;
    System.out.print((char)i);
    fout.write((byte)i);
}
    }
catch(IOException e)
{
    e.printStackTrace();
}
System.out.println("File Created Successfully");
}
}

```

---

Limitation of FileOutputStream :

---

By using this FileOutputStream class, we can write the data to single file only.  
If we want to write the data to multiple files at the same time then we should use a separate stream called ByteArrayOutputStream.

ByteArrayOutputStream :-

---

It is a predefined class available in java.io package. By using this class we can write the data to multiple files.

ByteArrayOutputStream class provides a method called writeTo(), through which we can write the data to multiple files.

```

//Program to write the data on multiple files.
import java.io.*;
public class File6
{
    public static void main(String args[]) throws IOException
    {
        var fin = new FileInputStream("File1.java");

        var f1 = new FileOutputStream("C:\\new\\a1.txt");
        var f2 = new FileOutputStream("C:\\new\\b1.txt");
        var f3 = new FileOutputStream("C:\\new\\c1.txt");

        var bout = new ByteArrayOutputStream();

        try(fin; f1; f2; f3; bout)
        {
            int i;
            while((i = fin.read()) != -1)
{
            bout.write((byte)i); //writing the data to ByteArrayOutputStream
}

```

```

        }

        bout.writeTo(f1);
        bout.writeTo(f2);
        bout.writeTo(f3);

        bout.flush(); //clear the buffer for reusing of ByteArrayOutputStream
        System.out.println("Success");
    }
}
-----
```

```

//Working with images
import java.io.*;
public class File7
{
    public static void main(String[] args) throws IOException
    {
        var fin = new FileInputStream("C:\\new\\abc.jpg");

        var f1 = new FileOutputStream("C:\\new\\a1.jpg");
        var f2 = new FileOutputStream("C:\\new\\a2.jpg");
        var f3 = new FileOutputStream("C:\\new\\a3.jpg");

        var bout = new ByteArrayOutputStream();

        try(fin; f1; f2; f3; bout)
        {
            int i;
            while((i = fin.read()) != -1)
            {
                bout.write((byte)i);
            }

            bout.writeTo(f1);
            bout.writeTo(f2);
            bout.writeTo(f3);
            System.out.println("success...");
            bout.flush();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

---

BufferedOutputStream :-

---

It is a predefined class available in java.io package.

Whenever we use the class FileOutputStream the data will be available on the Stream but not in the buffer so there may be chance of miss memory management, It is always preferable that we should perform read and write operation by using buffer.

By using this BufferedOutputStream now the data is in the buffer so the execution will become more faster.

```
//Program to put the data in the buffer for fast execution
import java.io.*;
class File8
{
    public static void main(String args[]) throws IOException
    {
        var fout = new FileOutputStream("C:\\\\new\\\\Hyderabad.txt");

        var bout = new BufferedOutputStream(fout);

        try(fout ; bout)
        {
            String s = "Hyderabad is a nice city";
            byte b[] = s.getBytes();
            bout.write(b);
            System.out.print("success...");
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

---

BufferedInputStream :-

---

It is a predefined class available in java.io package.

Whenever we use FileInputStream to read the data/content from the file the data will be available on the Stream but not in the buffer so there may be a chance of miss memory management so we should take the data into the buffer by using BufferedInputStream class so overall the execution will become faster.

```
//BufferedInputStream
import java.io.*;
public class File9
{
    public static void main(String args[]) throws IOException
    {
        var fin = new FileInputStream("File1.java");
        var bin = new BufferedInputStream(fin);

        try(fin ; bin)
        {
            int i;
            while((i = bin.read()) != -1)
            {
                System.out.print((char)i);
            }
        }
    }
}
```

```
        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
    System.out.println();
}
-----
```

25-05-2024

Writing and Reading the primitive data to the files :-

It is possible to write the primitive data(byte,short,int, long, float, double, char and boolean) to the file.

In order to write primitive data to the file we should use a predefined class available in java.io package called DataOutputStream.

```
var fos = new FileOutputStream("data.txt");
var dos = new DataOutputStream(fos);
```

now by using this we can write primitive data to the file.

It provides various methods like writeByte(), writeShort(), writeInt() and so on to write the primitive data to the file.

If we want to read the primitive data from the file we can use a predefined class available in java.io package called DataInputStream, this class provides various methods like readByte(), readShort(), readInt() and so on.

```
var fin = new FileInputStream("data.txt");
var dis = new DataInputStream(fin);
```

Note :- For writing String into a file we have writeBytes() and to read the String data from the file we have readLine() method.

[DataInputStream class readLine() method is deprecated now, so compilation warning]

```
//DataOutputStream and DataInputStream
import java.io.*;
public class File10
{
    public static void main(String args[]) throws IOException
    {
        var fout = new FileOutputStream("C:\\new\\Primitive.txt");
        var dout = new DataOutputStream(fout);

        try(fout ; dout)
        {
            dout.writeBoolean(true);
            dout.writeChar('A');
            dout.writeByte(Byte.MAX_VALUE);
            dout.writeShort(Short.MAX_VALUE);
        }
    }
}
```

```

dout.writeInt(Integer.MAX_VALUE);
dout.writeLong(Long.MAX_VALUE);
dout.writeFloat(Float.MAX_VALUE);
dout.writeDouble(Math.PI);//PI is a final static variable
    dout.writeBytes("Hello India..."); 
dout.flush();//For reuse purpose
}
catch(IOException e)
{
    e.printStackTrace();
}

System.out.println("Reading the Primitive data from the file!!!");

var fin = new FileInputStream("C:\\new\\Primitive.txt");
var din = new DataInputStream(fin);

try(fin ; din)
{
boolean f = din.readBoolean();
char c = din.readChar();
byte b = din.readByte();
short s = din.readShort();
int i = din.readInt();
long l = din.readLong();
float ft = din.readFloat();
double d = din.readDouble();
String x= din.readLine();//for reading String (deprecated)

System.out.println(f +"\n"+c+"\n"+b+"\n"+s+"\n"+i+"\n"+l+"\n"+ft+"\n"+d+"\n"+x);
}
catch(IOException e)
{
    e.printStackTrace();
}

}

```

---

#### \*\* Serialization and De-serialization :

---

It is a technique through which we can store the object data in a file. Storing the object data into a file is called **Serialization** on the other hand Reading the object data from a file is called **De-serialization**.

In order to perform serialization, a class must implements **Serializable** interface, predefined marker interface in **java.io** package.

**Java.io** package has also provided a predefined class called **ObjectOutputStream** to perform serialization i.e writing Object data to a file using **writeObject()** method.

```
public void writeObject(Object obj)
```

where as ObjectInputStream is also a predefined class available in java.io package through which we can read the Object data from a file using readObject(). The return type of readObject() is Object.

```
public Object readObject()
```

While reading the object data from the file, if the object is not available in the file then it will throw checked exception java.io.EOFException. (End of file Exception). It is a checked Exception.

3 files:

---

Employee.java

---

```
package com.ravi.serialization_and_de_serialization;

import java.io.Serializable;
import java.util.Date;
import java.util.Scanner;

@SuppressWarnings("serial")
public class Employee implements Serializable
{
    private int employeeId;
    private String employeeName;
    private double employeeSalary;
    private Date dateOfJoining; // HAS-A Relation

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

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

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

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

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

        Date d = new Date();

        return new Employee(id, name, sal, d);

    }

}
```

### SerializationDemo.java

---

```
package com.ravi.serialization_and_de_serialization;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Scanner;

public class SerializationDemo {

    public static void main(String[] args) throws IOException
    {
        var fout = new FileOutputStream("C:\\new\\Employee.txt");
        var oos = new ObjectOutputStream(fout);
        var sc = new Scanner(System.in);

        try(fout;oos; sc)
        {
            System.out.print("How many objects u want to write ");
            int noOfObjects = sc.nextInt();

            for(int i=1; i<=noOfObjects; i++)
            {
                Employee empObj = Employee.getEmployeeObject();
                oos.writeObject(empObj);
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        System.out.println("Object stored successfully");
    }
}
```

### DeserializationDemo.java

---

```
package com.ravi.serialization_and_de_serialization;
```

```

import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializationDemo
{
    public static void main(String[] args) throws IOException, ClassNotFoundException
    {
        var fin = new FileInputStream("C:\\new\\Employee.txt");
        var ois = new ObjectInputStream(fin);

        try(fin;ois)
        {
            Employee emp;

            while((emp = (Employee)ois.readObject()) !=null)
            {
                System.out.println(emp);
            }
        }
        catch(EOFException e)
        {
            System.err.println("End of File has reached");
        }
    }
}

```

-----  
transient keyword :

If we want that some of our field (variable) will not serialized then we should declare that variables with transient keyword so, we will get the defualt value for the variable.

```

public class Player
{
    private transient int playerId;
    private transient String playerName;
}

```

Now if we perform serialization operation on Player object then for playerId we will get 0 and for playerName we will get null.

2 files :

-----  
Student.java

```

package com.ravi.serialization_and_de_serialization;

import java.io.Serializable;

public class Student implements Serializable

```

```

{
    private transient int studentId;
    private String studentName;
    private double studentFees;

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

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

```

StudentDemo.java

```

-----
package com.ravi.serialization_and_de_serialization;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class StudentDemo {

    public static void main(String[] args) throws Exception
    {
        var fout = new FileOutputStream("C:\\new\\Student.txt");
        var oos = new ObjectOutputStream(fout);

        try(fout;oos)
        {
            Student s1 = new Student(1, "Raj", 15000);
            oos.writeObject(s1);
            System.out.println("Student Object Stored in file");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        var fin = new FileInputStream("C:\\new\\Student.txt");
        var ois = new ObjectInputStream(fin);

        try(fin;ois)
        {
            Student stu = (Student) ois.readObject();
            System.out.println(stu);
        }
    }
}

```

```
        }
    catch(Exception e)
    {
        e.printStackTrace();
    }

}
```

---

### Working With Character Oriented Stream :

---

In order to work with character oriented Stream, if we two super classes Reader and Writer.

Both are abstract classes and Reader is the super class for all types of reading operation, on the other hand Writer is the super class for all types of writing operation.

Here we can directly work with characters.

---

#### FileWriter class :

---

It is a predefined class available in java.io package, By using this class we can create a file and write character data to the file.

By using this class we can directly write String (collection of characters) Or character array to the file.

Actually It is a character oriented Stream where as if we work with FileOutputStream class, It is byte oriented Stream.

```
//FileWriter
import java.io.*;
public class File11
{
    public static void main(String args[]) throws IOException
    {
        var fw = new FileWriter("C:\\\\new\\\\HelloIndia.txt");
        var bw = new BufferedWriter(fw);

        try(fw; bw)
        {
            bw.write("India, It is in Asia");
            System.out.println("Success....");
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

---

```
//FileWriter
import java.io.*;
class File12
```

```
{  
    public static void main(String args[]) throws IOException  
    {  
        var fw = new FileWriter("C:\\new\\Data.txt");  
        var bw = new BufferedWriter(fw);  
  
        try(fw;bw)  
        {  
            char c[ ] = {'H','E','L','L','O', ' ', ' ', 'W','O','R','L','D'};  
  
            bw.write(c);  
            System.out.println("Success....");  
        }  
        catch(Exception e)  
        {  
            e.printStackTrace();  
        }  
    }  
}
```

---

#### FileReader class :

---

It is a predefined class available in java.io package, It is a character oriented Stream. The main purpose of this class to read the data in the character format directly from the file.

```
//FileReader  
import java.io.*;  
public class File13  
{  
    public static void main(String args[]) throws IOException  
    {  
        var fr = new FileReader(args[0]); //Command Line Arg  
        var br = new BufferedReader(fr);  
  
        try(fr ; br)  
        {  
            while(true)  
            {  
                int i = br.read();  
                if(i == -1)  
                    break;  
                System.out.print((char)i);  
            }  
        }  
        catch(IOException e)  
        {  
            e.printStackTrace();  
        }  
    }  
}
```

---

```
import java.io.*;  
public class File14  
{  
    public static void main(String[] args) throws IOException
```

```

{
var fr = new FileReader("C:\\new\\abc.jpg");
var fw = new FileWriter("C:\\new\\NIT.jpg");

try(fr;fw)
{
    int i;
    while((i=fr.read())!= -1)
    {
        fw.write(i);
    }
}
catch(Exception e)
{
}
System.out.println("Success");
}
}

```

Note :- In the above program the image file will not be created in a proper format because images are created by using binary data but FileReader and FileWriter can work with Character data.

---

**PrintWriter :**

---

It is a predefined class available in java.io package under Writer abstract class.

It is used to write primitive data into the file.

By using PrintWriter class we can create a file as well as write the primitive data into text format.

It contains a predefined non static method called printf() which accept two parameters

- 1) Formatted String
- 2) Actual Data.

```

//PrintWriter
import java.io.*;
public class File15
{
    public static void main(String[] args) throws IOException
    {
        PrintWriter writeData = new PrintWriter("C:\\new\\Roll.txt");

        try(writeData)
        {
            int roll = 15;
            //Writing primitive data into text format
            writeData.printf("My roll number is : %d ", roll);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

---

```

import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class File16
{
    public static void main(String[] args) throws IOException
    {
        var filename = "C:\\new\\append.txt";
        //wants to write the data in the existing file
        var fileWriter = new FileWriter(filename, true);
        var bufferedWriter = new BufferedWriter(fileWriter);

        try(fileWriter;bufferedWriter)
        {

            // Append text to the file
            String textToAppend = "My Name is Raj";
            bufferedWriter.write(textToAppend);

            //Moving the cursor to the next line
            bufferedWriter.newLine();

            textToAppend = "I lives in hyderabad";
            bufferedWriter.write(textToAppend);

            System.out.println("Text appended successfully to the file.");
        }
        catch (IOException e)
        {
            System.out.println("An error occurred while appending the text to the file: " +
e.getMessage());
        }
    }
}

```

At the time of creating the file using FileOutputStream Or FileWriter, Both will replace the existing file, If we want to append the data in the same existing file then we should 2nd parameter in the constructor to append new data in the existing file.

---

enum in java :

---

An enum is class in java that is used to represent group of universal constants. It is introduced from JDK 1.5 onwards.

In order to craete an enum, we can use enum keyword and all the univarsal constants of the enum must be separated by comma. Semicolon is optional at the end.

Example:-

```

enum Color
{
    RED, BLUE, BLACK, PINK //public + static + final
}

```

}

The enum constants are by default public, static and final.

An enum we can define inside the class, outside of the class and even inside of the method.

If we define an enum inside the class then we can apply public, private, protected and static.

Every enum in java extends java.lang.Enum class so an enum can implement many interfaces but can't extends a class.

By default every enum is implicitly final so we can't inherit an enum.

In order to get the constant value of an enum we can use values() method which returns enum array, but this method is added by compiler to each and every enum at the time of compilation.

In order to get the order position of enum constants we can use ordinal() method which is given inside the enum class and the return type of this method is int. The order position of enum constant will start from 0.

As we know an enum is just like a class so we can define any method, constructor inside an enum. Constructor must be either private or default.

\*All the enum constants are by default object of type enum.

Enum constants must be declared at the first line of enum otherwise we will get compilation error.

From java 1.5 onwards we can pass an enum in a switch statement.

---

```
public class Test1
{
    public static void main(String[] args)
    {
        enum Month
        {
            JANUARY, FEBRUARY, MARCH //public + static + final
        }

        System.out.println(Month.MARCH);
    }
}

enum Month
{
    JANUARY, FEBRUARY, MARCH
}

public class Test2
{
    enum Color { RED, BLUE, BLACK } //Test2$Color.class

    public static void main(String[] args)
    {
        enum Week { SUNDAY, MONDAY, TUESDAY }

        System.out.println(Month.FEBRUARY);
    }
}
```

---

```
        System.out.println(Color.RED);
        System.out.println(Week.SUNDAY);
    }
}
```

Note :- From the above Program it is clear that we can define an enum inside a class, outside of a class and inside a method as well.

---

```
//Comparing the constant of an enum
public class Test3
{
    enum Color { RED,BLUE }

    public static void main(String args[])
    {
        Color c1 = Color.RED;
        Color c2 = Color.RED;

        if(c1 == c2)
        {
            System.out.println("==");
        }
        if(c1.equals(c2))
        {
            System.out.println("equals");
        }
    }
}
```

As we know enum can't extend a class so they have given equals(Object obj) method support which internally using == operator.

---

```
public class Test4
{
    static enum Season //private, public, protected, static
    {
        SPRING, SUMMER, WINTER, RAINY;
    }

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

---

```
//Interview Question
class Hello
{
    int x = 100;
}

enum Direction extends Hello
{
    EAST, WEST, NORTH, SOUTH
}
```

```
class Test5
{
    public static void main(String[] args)
    {
        System.out.println(Direction.SOUTH);
    }
}
```

Note :- Every enum class extending from java.lang.Enum class so we can't write extends keyword with enum.

---

//All enums are by default final so can't inherit

```
enum Color
{
    RED, BLUE, PINK;
}
class Test6 extends Color
{
    public static void main(String[] args)
    {
        System.out.println(Color.RED);
    }
}
```

---

//values() to get all the values of enum

```
class Test7
{
    enum Season
    {
        SPRING, SUMMER, WINTER, FALL, RAINY
    }

    public static void main(String[] args)
    {
        Season [] values= Season.values();

        for(Season value : values)
            System.out.println(value);
    }
}
```

---

//ordinal() to find out the order position

```
class Test8
{
    static enum Season
    {
        SPRING, SUMMER, WINTER, FALL, RAINY
    }

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

```

        Season s1[] = Season.values();

        for(Season x : s1)
            System.out.println(x+" order is :" +x.ordinal());
    }

-----
```

//We can take main () inside an enum

```

enum Test9
{
    TEST1, TEST2, TEST3; //Semicolon is compulsory

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

//constant must be in first line of an enum

```

enum Test10
{
    public static void main(String[] args)
    {
        System.out.println("Enum main method");
    }
}

HR, SALESMAN, MANAGER;
```

}

-----

//Writing constructor in enum

```

enum Season
{
    WINTER, SUMMER, SPRING, RAINY; //All are object of type enum

    Season()
    {
        System.out.println("Constructor is executed....");
    }
}
```

class Test11

```

{
    public static void main(String[] args)
    {
        System.out.println(Season.WINTER);
        System.out.println(Season.SUMMER);

    }
}
```

-----

//Writing constructor with message

```

enum Season
```

```

{
    SPRING("Pleasant"), SUMMER("UnPleasant"), RAINY("Rain"), WINTER;

String msg;

Season(String msg)
{
    this.msg = msg;
}

Season()
{
    this.msg = "Cold";
}

public String getMessage()
{
    return msg;
}
}

class Test12
{
    public static void main(String[] args)
    {
        Season s1[] = Season.values();

        for(Season x : s1)
            System.out.println(x+" is :" +x.getMessage());
    }
}

```

---

```

enum MyType
{
    ONE
    {
        @Override
        public String toString()
        {
            return "this is one";
        }
    },
    TWO
    {
        @Override
        public String toString()
        {
            return "this is two";
        }
    }
}
public class Test13
{
    public static void main(String[] args)
    {

```

```
        System.out.println(MyType.ONE); //Passing Object ref
        System.out.println(MyType.TWO); //Passing Object ref
    }
}

-----
enum Color
{
    RED
    {
        @Override
        public String toString()
        {
            return "RED";
        }
    }
    ,
    BLUE
    {
        @Override
        public String toString()
        {
            return "BLUE";
        }
    }
}
public class EnumTest
{
    public static void main(String[] args)
    {
        System.out.println(Color.RED);
        System.out.println(Color.BLUE);
    }
}

-----
public class Test14
{
    enum Day
    {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
    }

    public static void main(String args[])
    {
        Day day = Day.MONDAY;

        switch(day)
        {
            case SUNDAY:
                System.out.println("Sunday");
        }
    }
}
```

```
break;
case MONDAY:
System.out.println("Monday");
break;
default:
System.out.println("other day");
}
}
-----
```

29-05-2024

Nested class OR Inner class in java :

In java it is possible to define a class (inner class) inside another class (outer class) called nested class.

In the same way it is also possible to define an interface OR enum OR Annotation or record inside another interface OR enum OR Annotation or record is called inner class or nested class.

Inner classes in Java create a strong encapsulation and HAS-A relationship between the inner class and its outer class.

An inner class, .class file will be represented by \$ symbol.

Advantages of inner class :

- 1) It is a way of logically grouping classes that are only used in one place.
- 2) It is used to achieve encapsulation.
- 3) It enhance the readability and maintainability of the code.

Types of Inner classes in java :

There are 4 types of inner classes in java :

- 1) Member Inner class OR Nested Inner class OR Regular class
- 2) Local inner class OR Method local inner class
- 3) static Nested inner class
- 4) Anonymous inner class.

1) Member Inner class OR Nested Inner class OR Regular class :

A non-static class that is created inside a class but outside of a method is called Member Inner class OR Nested Inner class OR Regular class.

It can be declared with access modifiers like private, default, protected, public, abstract and final.

It is also called as Regular Inner class.

An inner class can also access the private member of outer class.

Note :- The .class file of an inner class will be represented by \$ symbol at the time of compilation.

An outer class can be declared as public, abstract, final, sealed and non-sealed only.

---

```
class Outer
{
    private int a = 15;

    class Inner
    {
        public void displayValue()
        {
            System.out.println("Value of a is " + a);
        }
    }
}

public class Test1
{
    public static void main(String... args)
    {
        //Outer mo = new Outer(); //Outer class Object is created
        //Outer.Inner inner = mo.new Inner(); //Inner class object is created

        Outer.Inner inner = new Outer().new Inner();
        inner.displayValue();
    }
}
```

---

```
class MyOuter
{
    private int x = 7;
    public void makeInnner()
    {
        MyInnner in = new MyInnner();
        System.out.println("Inner y is "+in.y);
        in.seeOuter();
    }

    class MyInnner
    {
        private int y = 15;
        public void seeOuter()
        {
            System.out.println("Outer x is "+x);
        }
    }
}

public class Test2
{
    public static void main(String args[])
    {
        MyOuter m = new MyOuter();
        m.makeInnner();
    }
}
```

Note :- From the above program it is clear that we can access the private data of Outer class using inner class and inner class can also access the private data of Outer class.

---

```
class MyOuter
{
    private int x = 15;
    class MyInner
    {
        public void seeOuter()
        {
            System.out.println("Outer x is "+x);
        }
    }
}
public class Test3
{
    public static void main(String args[])
    {
        //Creating inner class object in a single line
        MyOuter.MyInner m = new MyOuter().new MyInner();
        m.seeOuter();
    }
}
```

---

```
class MyOuter
{
    static int x = 7;
    class MyInner
    {
        public static void seeOuter() //MyInner.seeOuter();
        {
            System.out.println("Outer x is "+x);
        }
    }
}
```

```
public class Test4
{
    public static void main(String args[])
    {
        MyOuter.MyInner.seeOuter();
    }
}
```

---

```
class Car
{
    private String make;
    private String model;
    private Engine engine;

    public Car(String make, String model, int horsePower)
    {
        this.make = make;
        this.model = model;
        this.engine = new Engine(horsePower);
    }
}
```

```

}

//Inner class
private class Engine
{
    private int horsePower;

    public Engine(int horsePower)
    {
        this.horsePower = horsePower;
    }

    public void start()
    {
        System.out.println("Engine started! Horsepower: " + horsePower);
    }

    public void stop()
    {
        System.out.println("Engine stopped.");
    }
}

public void startCar()
{
    System.out.println("Starting " + make + " " + model);
    this.engine.start();
}

public void stopCar()
{
    System.out.println("Stopping " + make + " " + model);
    this.engine.stop();
}
}

public class Test5
{

    public static void main(String[] args) {

        Car myCar = new Car("Swift", "Desire", 1200);

        myCar.startCar();

        myCar.stopCar();
    }
}

-----
class Laptop
{
    private String brand;
    private String model;
    private Motherboard motherboard;

    public Laptop(String brand, String model, String motherboardModel, String chipset)

```

```

    {
        this.brand = brand;
        this.model = model;
        this.motherboard = new Motherboard(motherboardModel, chipset);
    }

    public void switchOn()
    {
        System.out.println("Turning on " + brand + " " + model);
        this.motherboard.boot();
    }

    //Motherboard inner class
    public class Motherboard
    {
        private String model;
        private String chipset;

        public Motherboard(String model, String chipset)
        {
            this.model = model;
            this.chipset = chipset;
        }

        public void boot()
        {
            System.out.println("Booting " + brand + " " + model + " with " + chipset + " chipset");
        }
    }
}

public class Test6
{
    public static void main(String[] args)
    {

        Laptop laptop = new Laptop("HP", "ENVY", "IRIS", "Intel");

        laptop.switchOn();
    }
}

```

30-05-2024

---

```

class Person
{
    private String name;
    private int age;
    private Heart heart;

    public Person(String name, int age)
    {
        this.name = name;
    }
}
```

```

        this.age = age;
        this.heart = new Heart();
    }

    public void describe()
    {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Heart beats per minute: " + heart.getBeatsPerMinute());
    }

    // Inner class representing the Heart
    private class Heart
    {
        private int beatsPerMinute;

        public Heart()
        {
            this.beatsPerMinute = 72;
        }

        public int getBeatsPerMinute()
        {
            return this.beatsPerMinute;
        }

        public void setBeatsPerMinute(int beatsPerMinute)
        {
            this.beatsPerMinute = beatsPerMinute;
        }
    }
}

public class Test7
{
    public static void main(String[] args)
    {
        Person person = new Person("Virat", 30);
        person.describe();
    }
}

-----
class University
{
    private String name;

    public University(String name)
    {
        this.name = name;
    }

    public void displayUniversityName()
    {
        System.out.println("University Name: " + name);
    }
}

```

```

public class Department
{
    private String name;
    private String headOfDepartment;

    public Department(String name, String headOfDepartment)
    {
        this.name = name;
        this.headOfDepartment = headOfDepartment;
    }

    // Method to display department details
    public void displayDepartmentDetails()
    {
        displayUniversityName();
        System.out.println("Department Name: " + name);
        System.out.println("Head of Department: " + headOfDepartment);

    }
}

public class Test8
{
    public static void main(String[] args)
    {

        University university = new University("JNTU");

        University.Department cs = university.new Department("Computer Science", "Dr. John");

        University.Department ee = university.new Department("Electrical Engineering", "Dr. Scott");

        cs.displayDepartmentDetails();
        ee.displayDepartmentDetails();
    }
}

-----
class OuterClass
{
    private int x = 200;

    class InnerClass
    {
        public void display() //Inner class display method
        {
            System.out.println("Inner class display method");
        }

        public void getValue()
        {
            display();
            System.out.println("Can access outer private var :" + x);
        }
    }
}

```

```

        public void display() //Outer class display method
        {
            System.out.println("Outer class display");
        }
    }
public class Test9
{
    public static void main(String [] args)
    {
        OuterClass.InnerClass inobj = new OuterClass().new InnerClass();
        inobj.getValue();

        new OuterClass().display();
    }
}

class OuterClass
{
    int x;
    public class InnerClass //private, def, prot, public, abstract
    {
        //final
        int x;
    }
}
public class Test10
{
}

```

---

## 2) Method local inner class :

---

If a class is declared inside the method then it is called method local inner class.

We can't apply any access modifier on method local inner class but they can be marked as abstract and final.

A local inner class we can't access outside of the method that means the scope of method local inner class within the same method only.

```

//program on method local inner class
class MyOuter3
{
    private String x = "Outer class private data";

    public void doStuff()
    {
        String z = "local variable";

        class MyInner //only final and abstract is possible
        {
            public void seeOuter()
            {
                System.out.println("Outer x is "+x);
                System.out.println("Local variable z is : "+z);
            }
        }
    }
}
```

```
        }
        MyInner mi = new MyInner();
        mi.seeOuter();
    }

}
```

```
}
```

```
public class Test11
{
    public static void main(String args[])
    {
        MyOuter3 m = new MyOuter3();
        m.doSttuff();
    }
}
```

---

```
//local inner class we can't access outside of the method
```

```
class MyOuter3
{
    private String x = "Outer class Data";

    public void doSttuff()
    {
        String z = "local variable";

        class MyInner
        {
            String z = "CLASS variable";
            public void seeOuter()
            {
                System.out.println("Outer x is "+x);
                System.out.println("Local variable z is : "+z);
            }
        }
        MyInner mi = new MyInner();
        mi.seeOuter();
    }
}
```

```
}
```

```
public class Test12
{
    public static void main(String args[])
    {
        MyOuter3 m = new MyOuter3();
        m.doSttuff();
    }
}
```

---

```
3) Static Nested Inner class :
```

---

A static inner class which is declared with static keyword inside an outer class is called static Nested inner class.

It can't access non-static variables and methods i.e (instance members) of outer class because it is a static area.

For static nested inner class, Outer class object is not required.

If a static nested inner class contains static method then object is not required for inner class. On the other hand if the static inner class contains instance method then we need to create an object for static nested inner class.

```
//static nested inner class
class BigOuter
{
    static class Nest //static nested inner class
    {
        void go() //Instance method of static inner class
        {
            System.out.println("Hello welcome to static nested class");
        }
    }
}
class Test13
{
    public static void main(String args[])
    {
        BigOuter.Nest n = new BigOuter.Nest();
        n.go();
    }
}
```

---

```
class Outer
{
    static int x = 15;

    static class Inner
    {
        void msg()
        {
            System.out.println("x value is "+x);
        }
    }
}
class Test14
{
    public static void main(String args[])
    {
        Outer.Inner obj=new Outer.Inner();
        obj.msg();
    }
}
```

```
-----  
class Outer  
{  
    static int x = 25;  
    static class Inner  
    {  
        static void msg()  
        {  
            System.out.println("x value is "+x);  
        }  
    }  
}  
class Test15  
{  
    public static void main(String args[])  
    {  
        Outer.Inner.msg();  
    }  
}
```

---

```
class Outer  
{  
    int x=15; //error (not possible because try to access instance variable)  
    static class Inner  
    {  
        void msg()  
        {  
            System.out.println("x value is "+x);  
        }  
    }  
}  
class Test16  
{  
    public static void main(String args[])  
    {  
        Outer.Inner obj=new Outer.Inner();  
        obj.msg();  
    }  
}
```

---

31-05-2024

---

#### 4) Anonymous inner class :

---

It is an inner class which is defined inside a method without any name and for this kind of inner class only single Object is created. (Singleton class)

\*An anonymous inner class is mainly used for extending a class or implementing an interface that means creating sub type of a class or interface.

\*A normal class can implement any number of interfaces but an anonymous inner class can implement only one interface at a time.

A normal class can extend one class and implement any number of interfaces at the same time but an anonymous inner class can extend one class or can implement one interface at a time.

Inside an anonymous inner class we can write static , non static variable, static block and non-static block. Here we can't write abstract method and constructor.

---

```
//Anonymous inner class
class Tech
{
    public void tech()
    {
        System.out.println("Tech");
    }
}
public class Test17
{
    public static void main(String... args)
    {

        Tech a = new Tech() //Anonymous inner class
        {
            @Override
            public void tech()
            {
                System.out.println("anonymous tech");
            }
        };
        a.tech();
    }
}
```

---

```
@FunctionalInterface
interface Vehicle
{
    void move(); //SAM(Single Abstract Method)
}

class Test18
{
    public static void main(String[] args)
    {
        Vehicle car = new Vehicle()
        {
            @Override
            public void move()
            {
                System.out.println("Moving with Car...");
            }
        };
        car.move();

        Vehicle bike = new Vehicle()
        {
            @Override
```

```
        public void move()
        {
            System.out.println("Moving with Bike...");
        }
    };
    bike.move();
}
-----
```

Multithreading :

-----

Uniprocessing :-

-----

In uniprocessing, only one process can occupy the memory So the major drawbacks are

- 1) Memory is wastage
- 2) Resources are wastage
- 3) Cpu is idle

To avoid the above said problem, multitasking is introduced.

In multitasking multiple task 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

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.

Thread :-

-----

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 at the same time 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.

Program that describes that main is a Thread :

---

Whenever we define main method then JVM will create main thread internally, the purpose of this main thread to execute the entire main method.

In java there is a predefined class called Thread available in java.lang package, this class contains a predefined static method currentThread() which will provide currently executing Thread.

```
Thread t = Thread.currentThread(); //Factory Method
```

Thread class has provided predefined method getName() to get the name of the Thread.

```
package com.ravi.thread_demo;
```

```
public class MainThread {
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Thread t = Thread.currentThread();
```

```
        System.out.println("Current Thread name is :" + t.getName());
```

```
        //OR
```

```
        String name = Thread.currentThread().getName();
```

```
        System.out.println("Current Thread name is :" + name);
```

```
}
```

```
}
```

---

How to create a userdefined Thread in java ?

---

As we know whenever we define the main method then JVM internally creates a thread called main thread.

The purpose of main thread to execute the entire main method so at the time of execution of main method (main Thread) a user can create our own userdefined thread.

In order to create the userdefined Thread we can use one of the following two ways :-

- 1) By extending java.lang.Thread class
- 2) By implementing java.lang.Runnable interface

---

Note :- Thread is a predefined class available in java.lang package where as Runnable is a predefined interface available in java.lang Package.

---

```
//Program to create user thread by extending Thread class
package com.ravi.mt;
```

```
class MyThread 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");
        MyThread mt = new MyThread();
        mt.start();
        System.out.println("Main Thread ended");
    }
}
```

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.

In the above program main thread and Thread-0 threads are created both threads are executing their own Stack Memory as shown in the diagram

(01-JUN-24)

---

//Program that describes we have two different thread which we can separate by name

```
package com.ravi.mt;

class MyThread extends Thread
{
    @Override
    public void run()
    {
        System.out.println("Child thread is running");
        String name = Thread.currentThread().getName();
        System.out.println("Running thread name is :" + name);
    }
}
```

```
public class UserThread
{
    public static void main(String[] args)
    {
        System.out.println("Main Thread started");
        MyThread mt = new MyThread();
        mt.start();
        System.out.println("Main Thread ended");

        String name = Thread.currentThread().getName();
        System.out.println(name + " thread is running ");
    }
}
```

```
}
```

---

```
public boolean isAlive() :-
```

---

It is a predefined 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 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.mt;
```

```
class Foo extends Thread  
{
```

```
    @Override  
    public void run()  
    {  
        System.out.println("Child Thread is running with separate stack");  
    }  
}
```

```
public class IsAliveDemo {
```

```
    public static void main(String[] args)  
    {  
        Foo f1 = new Foo();  
        System.out.println("Is thread started before start :" + f1.isAlive());  
        f1.start();  
        System.out.println("Is thread started after start :" + f1.isAlive());  
        // f1.start(); java.lang.IllegalThreadStateException  
    }
```

```
}
```

---

```
package com.ravi.basic;
```

```
class Stuff extends Thread  
{
```

```
    @Override  
    public void run()  
    {  
        System.out.println("Child Thread is Running!!!!");  
    }  
}
```

```
public class ExceptionDemo
```

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

```

Stuff s1 = new Stuff();
Stuff s2 = new Stuff();

s1.start();
s2.start();

System.out.println(10/0);

System.out.println("Main Thread Ended");
}
}

```

Note :- Here main thread is interrupted due to AE but still child thread will be executed because child thread is executing with separate Stack

---

```

package com.ravi.basic;

class Sample extends Thread
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        for(int i=1; i<=10; i++)
        {
            System.out.println(i+" by "+name+ " thread");
        }
    }
}

public class ThreadLoop
{
    public static void main(String[] args)
    {
        System.out.println("Main Thread Started!!!");

        Sample s1 = new Sample();
        s1.start();

        String name = Thread.currentThread().getName();

        for(int i=1; i<=10; i++)
        {
            System.out.println(i+" by "+name+ " thread");
        }

        int x = 1;
        do
        {
            System.out.println("India");
            x++;
        }
        while(x<=10);
    }
}

```

```
}
```

---

Note :- From the above program, It is clear that CPU can frequently move from one thread to another thread. (Main thread to child thread-> Thread-0 thread)

---

How to set and get the name of the Thread :

---

Whenever we create a userdefined Thread in java then by default JVM assigns the name of thread is Thread-0, Thread-1, Thread-2 and so on.

If a user wants to assign some user defined name of the Thread, then Thread class has provided a predefined method called `setName(String name)` to set the name of the Thread.

On the other hand we want to get the name of the Thread then Thread class has provided a predefined method called `getName()`.

```
public final void setName(String name) //setter
```

```
public final String getName() //getter
```

---

```
package com.ravi.basic;
class DoStuff extends Thread
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " thread is running Here!!!!");
    }
}
```

```
public class ThreadName
{
    public static void main(String[] args)
    {
        DoStuff t1 = new DoStuff();
        DoStuff t2 = new DoStuff();

        t1.start();
        t2.start();
    }
}
```

Note :- If we don't provide our user-defined name for the thread then by default the name would be Thread-0, Thread-1 and so on.

---

```
package com.ravi.basic;
class Demo extends Thread
{
    @Override
```

```

public void run()
{
    System.out.println(Thread.currentThread().getName()+" thread is running....");
}
}

public class ThreadName1
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        t.setName("Parent"); //Changing the name of the main thread

        Demo d1 = new Demo();
        Demo d2 = new Demo();

        d1.setName("Child1");
        d2.setName("Child2");

        d1.start(); d2.start();

        String name = Thread.currentThread().getName();
        System.out.println(name + " Thread is running..");
    }
}

```

---

03-06-2024

---

**Thread.sleep(long millisecond) :**

---

If we want to put a thread into temporarily 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.

Thread.sleep(1000); //Thread will wait for 1 second

It is a static method of Thread class.

It is throwing a checked Exception i.e InterruptedException because there may be chance that this sleeping thread may be interrupted by some another thread so provide either try-catch or declare the method as throws.

---

```

package com.ravi.thread;

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.err.println("Thred interrupted :"+e);
    }
}
}

public class SleepDemo
{
    public static void main(String[] args)
    {
        Sleep s = new Sleep();
        s.start();

    }
}

-----
package com.ravi.thread;

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.err.println("Thred interrupted :"+e);
            }
        }
    }
}

public class SleepDemo
{
    public static void main(String[] args)
    {
        Sleep s = new Sleep();
        s.start();

    }
}
```

}

Assignment :

-----  
WAP to implement sleep(long millis, long nanos) method which is accepting two parameters.

-----  
Thread life cycle in java :

As we know a thread is well known for Independent execution and it contains a life cycle which internally contains 5 states (Phases).

During the life cycle of a thread, It can pass from these 5 states. At a time a thread can reside to only one state of the given 5 states.

- 1) NEW State (Born state)
- 2) RUNNABLE state (Ready to Run state) [Thread Pool]
- 3) RUNNING state
- 4) WAITING / BLOCKED state
- 5) EXIT/Dead state

New State :-

-----  
Whenever we create a thread instance(Thread Object) a thread comes to new state OR born state. New state does not mean that the Thread has started yet only the object or instance of Thread has been created.

Runnable state :-

-----  
Whenever we call start() method on thread object, A thread moves to Runnable state i.e Ready to run state. Here Thread scheduler is responsible to select/pick a particular Thread from Runnable state and sending that particular thread to Running state for execution.

Running state :-

-----  
If a thread is in Running state that means the thread is executing its own run() method.

From Running state a thread can move to waiting state either by an order of thread scheduler or user has written some method(wait(), join() or sleep()) to put the thread into temporarily waiting state.

From Running state the Thread may also move to Runnable state directly, if user has written Thread.yield() method explicitly.

Waiting state :-

-----  
A thread is in waiting state means it is waiting for its time period to complete. Once the time period will be completed then it will re-enter inside the Runnable state to complete its remaining task.

Dead or Exit :

---

Once a thread has successfully completed its run method then the thread will move to dead state. Please remember once a thread is dead we can't restart a thread in java.

IQ :- If we write Thread.sleep(1000) then exactly after 1 sec the Thread will re-start?

Ans :- No, We can't say that the Thread will directly move from waiting state to Running state.

The Thread will definitely wait for 1 sec in the waiting mode and then again it will re-enter into Runnable state which is controlled by Thread Scheduler so we can't say that the Thread will re-start just after 1 sec.

---

Anonymous inner class with Thread class :

---

Instead of taking a separate external class we can take the help of Anonymous inner class.

Anonymous inner class we can create by using the following 2 ways :

1) Anonymous inner class with reference :

```
package com.ravi.mt;

public class AnonymousThreadWithReference
{
    public static void main(String[] args)
    {
        //Anonymous inner class with reference
        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println("Running thread name is :" + name);
            }
        };

        t1.start();
        String name = Thread.currentThread().getName();
        System.out.println(name + " thread is running Here");
    }
}
```

---

2) Anonymous inner class without reference :

```
package com.ravi.mt;

public class AnonymousThreadWithoutRef {

    public static void main(String[] args)
    {
        //Anonymous inner class without reference
        new Thread()
```

```

    {
        @Override
        public void run()
        {
            String name = Thread.currentThread().getName();
            System.out.println("Running thread name is :" + name);
        }
    }.start();

}

-----

```

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 an instance method so we can call this method with the help of Thread object reference.

JoinDemo.java

```

-----
package com.ravi.basic;

class Join extends Thread
{
    @Override
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("i value is :" + i);
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }

    public class JoinDemo
    {
        public static void main(String[] args) throws InterruptedException

```

```

{
    System.out.println("Main Thread started!!");

    Join j1 = new Join();
    Join j2 = new Join();
    Join j3 = new Join();

    j1.start();

    j1.join();

    j2.start();
    j3.start();

    System.out.println("Main Thread completed");

}

-----
package com.ravi.basic;

public class JoinDemo1
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main thread started");
        Thread thread = Thread.currentThread();

        String name = thread.getName();

        for(int i=1; i<=5; i++)
        {
            System.out.println(i + " by "+name+ " thread ");
            Thread.sleep(1000);
        }

        thread.join(); //Deadlock
        System.out.println("Main thread ended");
    }
}

```

Here, It is a deadlock state because main thread is waiting for main thread to complete.

```

-----
package com.ravi.basic;

class Alpha extends Thread
{
    @Override
    public void run()
    {
        Thread t = Thread.currentThread();

```

```

String name = t.getName();           //Alpha_Thread

Beta b1 = new Beta();
b1.setName("Beta_Thread");
b1.start();
try
{
    b1.join(); //Alpha thread is Blocked
    System.out.println("-----");
}
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<=10; i++)
        {
            System.out.println(i+" by "+name);
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(".....");
    }
}
-----
```

Assignment :

```
join(long millis)
join(long millis, long nanoseconds)
```

---

Creating thread by using Runnable interface approach :

---

```
package com.ravi.basic;

class Ravi implements Runnable
{
    public void run()
    {
        System.out.println("Running");
    }
}
public class RunnableDemo
{
    public static void main(String [] args)
    {

        Thread t1 = new Thread(new Ravi());
        t1.start();
    }
}
```

---

Assigning different targets by using Runnable :

---

```
package com.ravi.basic;

class Tatkal implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println("Tatkal ticket is booked by :" + name);
    }
}

class General implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println("General ticket is booked by :" + name);
    }
}

public class RunnableDemo1
{
    public static void main(String[] args)
    {
```

```
        Thread t1 = new Thread(new Tatkal(),"Scott");
        t1.start();

        Thread t2 = new Thread(new General(),"Smith");
        t2.start();

    }

}
```

---

Anonymous inner class approach using Runnable :

---

```
package com.ravi.thread;

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("Running Thread name is :" + name);
            }
        };

        Thread t1 = new Thread(r1);
        t1.start();

        String name = Thread.currentThread().getName();
        System.out.println(name + " Thread is running Here");
    }
}
```

---

Runnable interface implementation using Lambda :

---

```
package com.ravi.thread;

public class RunnableLambda
{
    public static void main(String[] args)
    {
        Runnable r1 = () ->
        {
            String name = Thread.currentThread().getName();
            System.out.println("Running Thread name is :" + name);
        };

        Thread t1 = new Thread(r1); t1.start();
    }
}
```

---

Thread class constructor using Lambda OR Runnable inner class

```
package com.ravi.thread;

public class LambdaWithThreadWithoutReference
{
    public static void main(String[] args)
    {
        new Thread(()-> System.out.println(Thread.currentThread().getName()),"Child1").start();

        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println("Running Thread name is :" + name);
            }
        }, "child2").start();
    }
}
```

The above approach is very useful for single purpose.

---

05-06-2024

---

In between extends Thread and implements Runnable which one is better and why?

In between extends Thread and implements Runnable, implements Runnable is more better due to the following reasons :

1) In extend Thread class approach, further we can't extend any other class but while working with implements Runnable we can extend a single class.

class Ravi extends Thread

class Raj extends MyClass implements Runnable

2) By using extends Thread our target is run method of sub class

but by using implements Runnable we can assign different targets as per our requirement so it provides loose coupling.

Thread t1 = new Thread(Runnable target);

3) By using extends Thread, we cann't implement Lambda Expression but the same we can achieve by using Runnable interface.

---

Thread class Constructor :

---

In Thread class we have total 9 constructors are available but commonly we are using the following 7 constructors

- 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(ThreadGroup tg, String name);
  - 6) Thread t6 = new Thread(ThreadGroup tg, Runnable target);
  - 7) Thread t7 = new Thread(ThreadGroup tg, Runnable target, String name);
- 

#### Drawback of Multithreading :

---

Multithreading is very good to complete our task as soon as possible but in some situation, It provides some wrong data or wrong result.

In Data Race or Race condition, all the threads try to access the resource at the same time so the result will 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.(05-JUNE)

---

#### Drawback of Multithreading by Railway reservation System

---

```
package com.ravi.mt;

class Customer implements Runnable
{
    private int availableSeat = 1;
    private int wantedSeat;

    public Customer(int wantedSeat)
    {
        this.wantedSeat = wantedSeat;
    }

    @Override
    public void run()
    {
        String name = null;

        if(availableSeat >= wantedSeat)
        {
            name = Thread.currentThread().getName();
            System.out.println(wantedSeat+" seat is reserved for "+name);
            availableSeat = availableSeat - wantedSeat;
        }
        else
        {
    
```

```

        name = Thread.currentThread().getName();
        System.err.println("Sorry!! "+name+" berth is not available");
    }
}

public class RailwayReservation
{
    public static void main(String[] args) throws InterruptedException
    {
        Customer obj = new Customer(1);

        Thread t1 = new Thread(obj, "Scott");
        Thread t2 = new Thread(obj, "Smith");

        t1.start();
        t2.start();

    }
}

```

---

#### Drawback of Multithreading by Banking Application

---

```

package com.ravi.mt;

class BankCustomer
{
    int availableBalance = 20000;
    int withdrawAmount;

    public BankCustomer(int withdrawAmount)
    {
        this.withdrawAmount = withdrawAmount;
    }
}

public class BankApplication {

    public static void main(String[] args)
    {
        BankCustomer b = new BankCustomer(20000);

        Runnable r1 = ()->
        {
            String name = null;

            if(b.availableBalance >= b.withdrawAmount)
            {
                name = Thread.currentThread().getName();
                System.out.println("Amont is withdraw by :" +name);
                b.availableBalance = b.availableBalance - b.withdrawAmount;
            }
            else
            {

```

```

        name = Thread.currentThread().getName();
        System.err.println(name + " u have insufficient balance");
    }

};

Thread t1 = new Thread(r1,"Scott");
Thread t2 = new Thread(r1,"Smith");

t1.start();
t2.start();

}

-----
package com.ravi.advanced;

class MyThread implements Runnable
{
    private String str;

    public MyThread(String str)
    {
        this.str=str;
    }

    @Override
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println(str+ " : "+i);
            try
            {
                Thread.sleep(100);
            }
            catch (Exception e)
            {
                System.err.println(e);
            }
        }
    }
}

public class Theatre
{
    public static void main(String [] args)
    {
        MyThread obj1 = new MyThread("sell the Ticket");
        MyThread obj2 = new MyThread("Allocate the Seat");

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);
    }
}

```

```
        t1.start();
        t2.start();
    }
}
```

We can't perform two dependent task from Thread.

---

\* Synchronization :

---

In order to solve the problem of multithreading java software people has introduced synchronization concept.

In order to achieve synchronization in java we have a keyword called "synchronized".

It is a technique through which we can control multiple threads but accepting only one thread at all the time.

Synchronization allows only one thread to enter inside the synchronized area for a single object.

Synchronization can be divided into two categories :-

1) Method level synchronization

2) Block level synchronization

1) Method level synchronization :-

---

In method level synchronization, the entire method gets synchronized so all the thread will wait at method level and only one thread will enter inside the synchronized area as shown in the diagram.(05-JUN-24)

---

2) Block level synchronization

---

In block level synchronization the entire method does not get synchronized, only the part of the method gets synchronized so all the thread will enter inside the method but only one thread will enter inside the synchronized block as shown in the diagram (05-JUN-24)

---

Note :- In between method level synchronization and block level synchronization, block level synchronization is more preferable because all the threads can enter inside the method so only the PART OF THE METHOD GETS synchronized so only one thread will enter inside the synchronized block.

---

06-06-2024

How synchronization logic controls multiple threads ?

---

Every Object has a lock(monitor) in java environment and this lock can be given to only one Thread at a time.

Actually this lock is available with each individual object provided by Object class.

The thread who acquires the lock from the object will enter inside the synchronized area, it will complete its task without any disturbance because at a time there will be only one thread inside the synchronized area(for single Object). \*This is known as Thread-safety in java.

The thread which is inside the synchronized area, after completion of its task while going back will release the lock so the other threads (which are waiting outside for the lock) will get a chance to enter inside the synchronized area by again taking the lock from the object and submitting it to the synchronization mechanism.

This is how synchronization mechanism controls multiple Threads.

Note :- Synchronization logic can be done by senior programmers in the real time industry because due to poor synchronization there may be chance of getting deadlock.

---

Program on Method Level Synchronization :

---

```
package com.ravi.thread;

class Table
{
    public synchronized void printTable(int num)
    {
        for(int i = 1; i<=10; i++)
        {
            System.out.println(num+" X "+i+" = "+(num*i));
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }
        System.out.println(".....");
    }
}

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

        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.advanced;

//Block level synchronization

class ThreadName
{
    public void printThreadName()
    {
        String name = Thread.currentThread().getName();
        System.out.println("Thread inside the method is :" + name);

        synchronized(this) //synchronized Block
        {
            for(int i=1; i<=9; i++)
            {
                System.out.println("i value is :" + i + " by :" + name);
            }
            System.out.println(".....");
        }
    }
}

public class BlockSynchronization
{
    public static void main(String[] args)
    {
        ThreadName obj1 = new ThreadName(); //lock is created

        Runnable r1 = () -> obj1.printThreadName();

        Thread t1 = new Thread(r1, "Child1");
        Thread t2 = new Thread(r1, "Child2");
        t1.start(); t2.start();
    }
}

```

---

### Drawback with Object level synchronization :

---

From the given diagram it is clear that there is no interference between t1 and t2 thread because they are passing through Object1 where as on the other hand there is no interference 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, t3 or t4 thread can enter inside the synchronized area at the same time, similarly it is also possible that with t2 thread, t3 or t4 thread can enter

inside the synchronized area so the conclusion is, synchronization mechanism does not work with multiple Objects.(Diagram 06-JUNE-24)

```
package com.ravi.advanced;
class PrintTable
{
    public synchronized void printTable(int n)
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println(n+" X "+i+" = "+(n*i));
            try
            {
                Thread.sleep(500);
            }
            catch(Exception e)
            {
            }
        }
        System.out.println(".....");
    }
}

public class ProblemWithObjectLevelSynchronization
{
    public static void main(String[] args)
    {
        PrintTable pt1 = new PrintTable(); //lock1
        PrintTable pt2 = new PrintTable(); //lock2

        Thread t1 = new Thread() //Anonymous inner class concept
        {
            @Override
            public void run()
            {
                pt1.printTable(2); //lock1
            }
        };

        Thread t2 = new Thread()
        {
            @Override
            public void run()
            {
                pt1.printTable(3); //lock1
            }
        };

        Thread t3 = new Thread()
        {
            @Override
            public void run()
            {
                pt2.printTable(6); //lock2
            }
        };
    }
}
```

```

        };

        Thread t4 = new Thread()
        {
            @Override
            public void run()
            {
                pt2.printTable(9); //lock2
            }
        };
        t1.start();    t2.start();    t3.start(); t4.start();
    }
}

```

So, from the above program it is clear that synchronization will not work with multiple objects.

Now, to avoid this Static Synchronization is came into the picture.

---

**Static Synchronization :**

---

If We declare a synchronized method as a static method then it is called static synchronization.

Now with static synchronization lock will be available at class level but not Object level.

To call the static synchronized method, object is not required so we can call the static method with the help of class name.

Unlike objects we can't create multiple classes for the same application.

---

```

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 = new Runnable()
{
    @Override
    public void run()
    {
        MyTable.printTable(15);
    }
};

Thread t3 = new Thread(r3);

t1.start();
t2.start();    t3.start();

}
}

```

Here the thread will take the lock from Table class.

---

**\*\* Inter Thread Communication(ITC) :**

---

It is a mechanism to communicate 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 will put a thread into temporarily waiting state and it will release the lock.  
It will wait till the another thread invokes notify() or notifyAll() for this object.

---

public native final void notify() :-

---

It will wake up the single thread that is waiting on the same object.

---

public native final void notifyAll() :-

---

It will wake up all the threads which are waiting on the same object.

\*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.

---

\*What is the difference between sleep() and wait()

---

(Given in the diagram 07-JUN-24)

---

//Program that describes if we don't use ITC then the problem is ...

```
class Test implements Runnable
{
    int var = 0;
    @Override
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            var = var + i; //var -> 1 3 6 10 15
            try
            {
                Thread.sleep(200);
            }
            catch (Exception e)
            {
            }
        }
    }
}

public class ITCProblem
{
    public static void main(String[] args)
    {
        Test t = new Test();
        Thread t1 = new Thread(t);
        t1.start();
        try
        {
            Thread.sleep(200);
        }
        catch (Exception e)
```

```

    {
}
System.out.println(t.var);
}
}

```

In the above program, there is no communication between main thread and child thread because value of var will modify with loop iteration.

wait(), notify() and notifyAll(), we should use all these methods from the synchronized area only otherwise we will get an exception i.e java.lang.IllegalMonitorStateException

---

//Communication between main thread and child thread using ITC

```

class SecondThread extends Thread
{
    int x = 0;

    @Override
    public void run()
    {
        //Child thread will wait Here for the lock
        synchronized(this)
        {
            for(int i=1; i<=10; i++)
            {
                x = x + i;
            }
            System.out.println("Sending notification");
            notify();
        }
    }
}

public class InterThreadComm
{
    public static void main(String [] args)
    {
        SecondThread b = new SecondThread();
        b.start();

        synchronized(b) //lock is taken by main Thread
        {
            //Main thread is suspended by Thread Scheduler
            try
            {
                System.out.println("Waiting for b to complete...");
                b.wait(); //main thread is in wait mode and lock is
                //released

                System.out.println("Main thread wake up");
            }
            catch (InterruptedException e)
            {

```

```

        }
        System.out.println("Value is: " + b.x);
    }
}

class Customer
{
    int balance = 10000;

    public synchronized void withdraw(int amount) //amount = 15000
    {
        System.out.println("going to withdraw... ");
        if(balance < amount)
        {
            System.out.println("Less balance; waiting for deposit... ");
            try
            {
                wait(); //Thread will release the lock
            }
            catch(Exception e){}
        }
        balance = balance - amount;
        System.out.println("withdraw completed..." +balance+" is remaining balance");
    }

    public synchronized void deposit(int amount)
    {
        System.out.println("going to deposit... ");
        balance = balance + amount;
        System.out.println("Balance after deposit is :" +balance);
        System.out.println("deposit completed... ");
        notify();
    }
}
public class InterThreadBalance
{
    public static void main(String args[])
    {
        Customer c = new Customer(); //lock is created here

        Thread son = new Thread()
        {
            @Override
            public void run()
            {
                c.withdraw(15000);
            }
        };
        son.start();

        Thread father = new Thread()
        {
            public void run()
            {

```

```

        c.deposit(9000);
    }
};

father.start();
}
}
-----
```

Program on notifyAll() method :

---

```

class Resource
{
    private boolean flag = false;

    public synchronized void waitMethod()
    {
        System.out.println("Wait");
        while (!flag)
        {
            try
            {
                System.out.println(Thread.currentThread().getName() + " is waiting...");
                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 InterThreadNotifyAll
{
    public static void main(String[] args)
    {
        Resource r1 = new Resource();

        Thread t1 = new Thread(() -> r1.waitMethod(), "Child1");
        Thread t2 = new Thread(() -> r1.waitMethod(), "Child2");
        Thread t3 = new Thread(() -> r1.waitMethod(), "Child3");

        Thread setter = new Thread(() -> r1.setMethod(), "Setter_Thread");

        t1.start();
        t2.start();
    }
}
```

```
t3.start();

try
{
    Thread.sleep(2000);
}
catch (InterruptedException e)
{
    e.printStackTrace();
}

setter.start();
}
```

---

08-06-2024

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 :-

`Thread.MIN_PRIORITY` :- 01

`Thread.NORM_PRIORITY` : 05

`Thread.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`.

WAP that describes whenever we create a thread by default its prority would be 5

```
package com.ravi.advanced;

public class MainPriority
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();

        System.out.println(t.getPriority());
    }
}
```

```
        Thread t1 = new Thread();
        System.out.println(t1.getPriority());
    }

}
```

---

WAP that describes child thread always acquire main thread priority.

```
package com.ravi.advanced;

class ThreadP extends Thread
{
    @Override
    public void run()
    {
        int priority = Thread.currentThread().getPriority();

        System.out.println("Child Thread priority is :" + priority);
    }
}

public class MainPriority1
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        t.setPriority(8);

        //t.setPriority(11); Invalid java.lang.IllegalArgumentException

        System.out.println("Main thread priority is :" + t.getPriority());

        ThreadP t1 = new ThreadP();
        t1.start();
    }
}
```

---

package com.ravi.advanced;

```
class ThreadPrior1 extends Thread
{
    @Override
    public void run()
    {
        int count = 0;

        for(int i=1; i<=1000000; i++)
        {
            count++;
        }

        System.out.println("Thread name is:" + Thread.currentThread().getName());
        System.out.println("Thread priority is:" + Thread.currentThread().getPriority());
    }
}
```

```

public static void main(String args[])
{
    ThreadPrior1 m1 = new ThreadPrior1();
    ThreadPrior1 m2 = new ThreadPrior1();

    m1.setPriority(Thread.MIN_PRIORITY);//1
    m2.setPriority(Thread.MAX_PRIORITY);//10

    m1.setName("Last");
    m2.setName("First");

    m1.start();
    m2.start();
}

```

Most of time the thread having highest priority will complete its task but we can't say that it will always complete its task first.

---

**Thread.yield() :**

---

It is a static method of Thread class.

It will send a notification to thread scheduler 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. Here The running Thread will directly move from Running state to Runnable state.

The Thread scheduler can 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 throwing InterruptedException]

If the thread which is in runnable state is having low priority then the current executing thread in Running state, will continue its execution.

It is mainly used to avoid the over-utilisation a CPU by the current Thread.

---

```

class Test implements Runnable
{
    @Override
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            String name = Thread.currentThread().getName();

            System.out.println("i value is :" + i + " by thread :" + name);

            if(name.equals("Child1"))
            {
                Thread.yield(); //Give a chance to Child2 Thread
            }
        }
    }
}

```

```

        }
    }
}

public class ThreadYieldMethod
{
    public static void main(String[] args)
    {
        Test obj = new Test();

        Thread t1 = new Thread(obj, "Child1");
        Thread t2 = new Thread(obj, "Child2");

        t1.start(); t2.start();
    }
}

```

Note :- In real time if a thread is acquiring more time of CPU then to release that Thread we call yield() method the currently executing Thread.

---

**ThreadGroup :-**

---

There is a predefined class called ThreadGroup available in java.lang package.

In Java it is possible to group multiple threads in a single object so, we can perform a particular operation on a group of threads by a single method call.

The target of thread may be same or may be different.

The Thread class has the following constructor for ThreadGroup

```
new Thread(ThreadGroup groupName, Runnable target, String name);
```

ThreadGroup class has provided the following two methods :

public String getName() : Will provide group name

public int activeCount() : Will provide total number of active threads  
in that 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+ " by "+name+ " thread");
        }
    }
}

```

```

public class ThreadGroupDemo1
{
    public static void main(String[] args)
    {
        ThreadGroup tg = new ThreadGroup("NIT_Thread");

        Thread t1 = new Thread(tg, new Foo(), "Child1");
        Thread t2 = new Thread(tg, new Foo(), "Child2");
        Thread t3 = new Thread(tg, new Foo(), "Child3");

        t1.start(); t2.start(); t3.start();

        System.out.println("Thread Group Name :" + tg.getName());
        System.out.println("Active threads are :" + tg.activeCount());
    }
}

-----
package com.ravi.group;

public class ThreadGroupDemo2 {

    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println(t.toString());

    }
}

-----
package com.ravi.group;

class Tatkal implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " is booked Tatkal Ticket");
    }
}
class General implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " is booked Genearl Ticket");
    }
}

```

```
public class ThreadGroupDemo3
{
    public static void main(String[] args)
    {
        ThreadGroup tatkal = new ThreadGroup("Tatkal");
        ThreadGroup general = new ThreadGroup("General");

        Thread t1 = new Thread(tatkal, new Tatkal(), "Scott");
        Thread t2 = new Thread(general, new General(), "Smith");

        t1.start(); t2.start();
    }
}
```

---

10-06-2024

---

interrupt Method of Thread class :

---

It is a predefined 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 interrupted 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<=10; i++)
        {
            System.out.println(i);
        }
    }
}
```

```

        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException 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());
        it.start();
        it.interrupt(); //main thread is interrupting the child thread
    }
}

```

Note :- Here main thread is interrupting child thread.

---

```

class Interrupt extends Thread
{
    public void run()
    {
        try
        {
            Thread.currentThread().interrupt();

            for(int i=1; i<=10; i++)
            {
                System.out.println("i value is :" + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.err.println("Thread is Interrupted :" + e);
        }
        System.out.println("Child thread completed...");
    }
}

public class InterruptThread1
{
    public static void main(String[] args)
    {
        System.out.println("Main thread is started");
        Interrupt it = new Interrupt();
        it.start();
        System.out.println("Main thread is ended");
    }
}

```

```
}
```

Here child Thread is interrupting itself.

---

```
public class InterruptThread2
{
    public static void main(String[] args)
    {
        Thread thread = new Thread(new MyRunnable());
        thread.start();

        try
        {
            Thread.sleep(3000);
        }
        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...");
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Thread interrupted gracefully.");
        }
        finally
        {
            System.out.println("Thread resource can be release here.");
        }
    }
}
```

Note :- Here main thread is in sleeping mode for 3 sec, after wake up main thread is interrupting child thread so child thread will come out from infinite loop and if any resource is attached with child thread that will be released because child thread execution completed.  
finally block is there to close the resource.

---

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 its 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. (10-JUNE-24)

```
public class DeadlockExample
{
    public static void main(String[] args)
    {
        String resource1 = "Ameerpet";
        String resource2 = "S R Nagar";

        // t1 tries to lock resource1 then resource2

        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                synchronized (resource1)
                {
                    System.out.println("Thread 1: locked resource 1");
                    try
                    {
                        Thread.sleep(1000);
                    }
                    catch (Exception e)
                    {}

                    synchronized (resource2) //Nested synchronized block
                    {
                        System.out.println("Thread 1: locked resource 2");
                    }
                }
            }
        };

        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread()
        {
            @Override
            public void run()
            {
                synchronized (resource2)
                {
                    System.out.println("Thread 2: locked resource 2");
                    try
                    {
                        Thread.sleep(1000);
                    }
                    catch (Exception e)
                    {}
                }
            }
        };
    }
}
```

```

        {}
    synchronized (resource1) //Nested synchronized block
    {
        System.out.println("Thread 2: locked resource 1");
    }
}
};

t1.start();
t2.start();
}
}

```

Note : Here this situation is known as Deadlock situation because both the threads are waiting for infinite state.

---

----- 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)

---

```

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...");
}
}

```

---

Remaining method of Object class :

---

Object cloning in java :

---

Object cloning is the process of creating an exact copy of an existing object in the memory.

Object cloning can be done by the following process :

- 1) Creating Shallow copy
- 2) Creating Deep copy
- 3) Using clone() method of java.lang.Object class
- 4) Passing Object reference to the Constructor.

Shallow Copy :

---

In shallow copy, we create a new reference variable which will point to same old existing object so if we make any changes through any of the reference variable then original object content will be modified.

Here we have one object and multiple reference variables.

Hashcode of the object will be same because all the reference variables are pointing to the same object.

---

```

package com.ravi.clone_method;

class Student
{

```

```

int id;
String name;

@Override
public String toString()
{
    return "Id is :" + id + "\nName is :" + name ;
}

}

public class ShallowCopy
{
    public static void main(String[] args)
    {
        Student s1 = new Student();
        s1.id = 111;
        s1.name = "Ravi";

        System.out.println(s1);

        System.out.println("After Shallow Copy");

        Student s2 = s1; //shallow copy
        s2.id = 222;
        s2.name = "Shankar";

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

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

```

---

### Deep Copy :

---

In deep copy, We create a copy of object in a different memory location. This is called a Deep copy.

Here objects are created in two different memory locations so if we modify the content of one object it will not reflect another object.

Here hashCode of both the objects will be different.

```

package com.ravi.clone_method;

class Employee
{
    int id;
    String name;

    @Override
    public String toString()
    {

```

```

        return "Employee [id=" + id + ", name=" + name + "]";
    }
}

public class DeepCopy
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        e1.id = 111;
        e1.name = "Ravi";

        Employee e2 = new Employee();
        e2.id = e1.id;
        e2.name = e1.name;

        System.out.println(e1 + " : " + e2);

        e2.id = 222;
        e2.name = "shankar";
        System.out.println(e1 + " : " + e2);

        System.out.println(e1.hashCode() + " : " + e2.hashCode());
    }
}

```

---

**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].

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 follow deep copy concept so hashcode will be different.

clone() method of Object class has protected access modifier so we need to override clone() method in sub class.

---

package com.ravi.clone\_method;

class Customer implements Cloneable  
{

```

int id;
String name;

@Override
protected Object clone() throws CloneNotSupportedException
{
    return super.clone();
}

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

public class CloneMethod
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Customer c1 = new Customer();
        c1.id = 222;
        c1.name = "Rahul";

        Customer c2 = (Customer) c1.clone(); //deep copy
        c2.id = 333;
        c2.name = "Rohit";

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

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

```

**protected void finalize() throws Throwable :**

It is a predefined method of Object class.

Garbage Collector automatically call this method just before an object is eligible for garbage collection to perform clean-up activity.

Here clean-up activity means closing the resources associated with that object like file connection, database connection, network connection and so on we can say resource de-allocation.

Note :- JVM calls finalize method only one per object.

Diagram (11-JUNE-24)

```
-----  
package com.ravi.finalize_method;  
  
public class Student  
{  
    int id;  
    String name;  
  
    public Student(int id, String name)  
    {  
        this.id= id;  
        this.name = name;  
    }  
  
    @Override  
    public String toString()  
    {  
        return "Id is :" +id+ "\nName is :" +name;  
    }  
  
    @Override  
    protected void finalize()  
    {  
        System.out.println("JVM call this finalize method...");  
    }  
  
    public static void main(String[] args) throws InterruptedException  
    {  
        Student s1 = new Student(111,"Ravi");  
        System.out.println(s1.hashCode());  
        System.out.println(s1);  
  
        s1 = null;  
        System.gc(); //Explicitly calling Garbage Collector  
        Thread.sleep(3000);  
        System.out.println(s1);  
    }  
}
```

\*What is the difference between final, finally and finalize

final :- It is a keyword which is used to provide some kind of restriction like class is final, Method is final, variable is final.

finally :- if we open any resource as a part of try block then that particular resource must be closed inside finally block otherwise program will be terminated abnormally and the corresponding resource will not be closed (because the remaining lines of try block will not be executed)

finalize() :- It is a method which JVM is calling automatically just before object destruction so if any resource (database, file and network) is associated with

that particular object then it will be closed or de-allocated by JVM by calling finalize().

---

### String Handling in java :

---

A string literal in Java is basically a sequence of characters. These characters can be anything like alphabets, numbers or symbols which are enclosed with double quotes. So we can say String is a collection of alpha-numeric character.

How we can create String in Java :-

---

In java String can be created by using 3 ways :-

1) By using String Literal

```
String x = "Ravi";
```

2) By using new keyword

```
String y = new String("Hyderabad");
```

3) By using character array

```
char z[] = {'H','E','L','L','O'};
```

### Immutability in String (Diagram 11-JUNE-24)

---

In java Strings Objects are immutable means unchanged so, whenever we create a String object in java it can't be modifiable.

Strings literals are created in a very special memory of HEAP called String Constant Pool(SCP) and it is not eligible for garbage collection.

String once created can't be modifiable.

Note :- Whenever we create the String Object using literal then internally intern() method is invoked to place the String object into String Constant Pool Area.

---

12-06-2024

---

### Facts about String and Memory :

---

In java Whenever we create a new String object by using String literal, first of all JVM will verify whether the String we want to create is pre-existing (already available ) in the String constant pool or not.

If the String is pre-existing (already available) in the String Constant pool then JVM will not create any new String object, the same old existing String object would be refer by new reference variable as shown in the diagram(12-JUN)

Note :- In SCP area we can't have duplicate String Object.

---

\* Why String objects are immutable :

---

As we know a String object in the String constant pool can be refer by multiple reference variables, if any of the reference variable will modify the String Object value then it would be very difficult for the another reference variables pointing to same String object to get the original value, what they have defined earlier as shown in the diagram.(15-MAR)  
That is the reason Strings are immutable in java.

Note :- While working with Map interface we can hold String object as a Map key so that time also it must be immutable.

```
HashMap<String, Integer> hm = new HashMap<>();  
hm.put("Ravi", 1234);
```

---

All the Wrapper classes are immutable in java.

```
public class Immutable {  
  
    public static void main(String[] args)  
    {  
        Integer obj = 25;  
        accept(obj);  
        System.out.println(obj);  
    }  
  
    public static void accept(Integer i)  
    {  
        i = 50;  
    }  
}
```

---

WAP in java to show String objects are not eligible for GC.

---

```
package com.ravi.exception;  
  
public class StringGC {  
  
    public static void main(String[] args) throws InterruptedException  
    {  
        String str = "india";  
        System.out.println(str.hashCode());  
  
        str = null;  
  
        System.gc(); //Explicitly calling GC  
        Thread.sleep(5000);  
  
        String s1 = "india";  
        System.out.println(s1.hashCode());  
  
    }  
}
```

---

From the above program it is clear that Strings are not eligible for GC.

\*\*\*What is the difference between following two statements :

```
String s1 = new String("Hyd");
```

```
String s2 = "Hyd";
```

```
String s2 = "Hyd";
```

It will create one String object and one reference variable and String object will be created in the String constant pool.

```
String s1 = new String("Hyd");
```

It will create two String objects one is inside the heap memory(non SCP area) which will be referred by s1 reference variable and the same String object will be placed in the String constant pool area, if it is not available there.

Hence two String Objects and one reference variable will be created.

---

Program in java that describes whenever we create String object by using new keyword then two String objects are created with same content with same hash code.

```
package com.ravi.exception;

public class StringComparison
{
    public static void main(String[] args)
    {
        String str1 = new String("Hyderabad");

        String str2 = "Hyderabad";

        System.out.println(str1==str2);

        System.out.println(str1.hashCode());
        System.out.println(str2.hashCode());

    }
}
```

Here in the above program Hashcode will be same.

---

The String object created by using new keyword, how they are automatically placed inside SCP area.

In Java whenever we create the String Object by using new keyword then one object will be created in the non SCP area and same object will be placed inside SCP area, Actually Here JVM internally performs intern process. By using this intern process, JVM is placing the String into SCP Area.

The following program explains how to use intern() method explicitly (By user) to place the String created by new keyword into SCP Area.

---

Diagram [12-JUNE]

```
package com.ravi.exception;
```

```
public class StringIntern {  
  
    public static void main(String[] args)  
    {  
        String str1 = new String("Java");  
        String str2 = "Java";  
  
        System.out.println(str1==str2); //false  
  
        str1 = str1.intern();  
        System.out.println(str1==str2); //true  
    }  
}
```

---

```
package com.ravi.exception;
```

```
public class StringIntern {  
  
    public static void main(String[] args)  
    {  
        String s1 = "india";  
        String s2 = new String("india");  
  
        String s3 = s2.intern();  
  
        System.out.println(s1==s2);//false  
        System.out.println(s1==s3);//true  
    }  
}
```

Note :- Java automatically interns the string literals but we can manually use the intern() method on String object created by new keyword so all the Strings which are having same content will get the same String and return the same memory address(Canonical representation for the String ).

From the above program it is clear that, All that String having same content will represent the same memory address so the hashCode value will be same as shown in the program below.

```
package com.nit.oop;
```

```
public class StringHashCode {
```

```
    public static void main(String[] args)  
    {  
        String str1 = "india";  
        String str2 = new String("india");
```

```
String str3 = new String("india");
System.out.println(str1.hashCode() + ":" +str2.hashCode() + ":" +str3.hashCode());
}
-----
```

13-06-2024

### Collection Framework in java (40 - 45% IQ):

Collections framework is nothing but handling individual Objects(Collection Interface) and Group of objects(Map interface).

We know only object can move from one network to another network.

A collections framework is a class library to handle group of Objects.

It is implemented by using `java.util` package.

It provides an architecture to store and manipulate group of objects.

All the operations that we can perform on data such as searching, sorting, insertion and deletion can be done by using collections framework because It is the data structure of Java.

The simple meaning of collections is single unit of Objects.

It provides the following sub interfaces :

- 1) List (Accept duplicate elements)
- 2) Set (Not accepting duplicate elements)
- 3) Queue (Storing and Fetching the elements based on some order i.e FIFO)

Note : Collection is a predefined interface available in `java.util` package where as Collections is a predefined class which is available from JDK 1.2V which contains only static methods (Constructor is private)

14-06-2024

### Methods of Collection interface :

- a) public boolean add(E element) :- It is used to add an item/element in the collection.
- b) public boolean addAll(Collection c) :- It is used to insert the specified collection elements in the existing collection(For merging the Collection)
- c) public boolean retainAll(Collection c) :- It is used to retain all the elements from existing element. (Common Element)
- d) public boolean removeAll(Collection c) :- It is used to delete all the elements from the existing collection.
- e) public boolean remove(Object element) :- It is used to delete an element from the collection based on the object.

f) public int size() :- It is used to find out the size of the Collection [Total number of elements available]

g) public void clear() :- It is used to clear all the elements at once from the Collection.

All the above methods of Collection interface will be applicable to all the sub interfaces like List, Set and Queue.

---

List interface :

---

It is the sub interface of Collection interface.

It stores the elements based on the index.

It accepts duplicate elements.

We can perform sorting operation on list interface by using sort method of Collections class.

Most of the List interface classes are dynamically Growable.

---

Hierarchy of List interface :

---

Available in the paint diagram 14-JUNE

---

Behavior of List interface implemented classes :

---

1) It stores the elements based on the index.

2) It accepts duplicate elements.

3) It accepts homogeneous and heterogeneous elements.

4) It accepts null.

5) It accepts everything in the form of Object.

6) We can use generics to eliminate compilation warning and with

generics we can accept homogeneous and heterogeneous (<Object>)

7) Classes are dynamically Growable (Except LinkedList)

Methods of List interface :

---

1) public boolean isEmpty() :- Verify whether List is empty or not

2) public void clear() :- Will clear all the elements

3) public int size() :- To get the size of the Collections

4) public void add(int index, Object o) :- Insert the element based on the index position.

5) public boolean addAll(int index, Collection c) :- Insert the Collection based on the index position

6) public Object get(int index) :- To retrieve the element based on the index position

7) public Object set(int index, Object o) :- To override or replace the existing element based on the index position

8) public Object remove(int index) :- remove the element based on the index position

9) public boolean remove(Object element) :- remove the element based on the object element, It is the Collection interface method extended by List interface

10) public int indexOf() :- index position of the element

11) public int lastIndex() :- last index position of the element

12) public Iterator iterator() :- To fetch or iterate or retrieve the elements from Collection in forward direction only.

13) public ListIterator listIterator() :- To fetch or iterate or retrieve the elements from Collection in forward and backward direction.

---

\*\*\* How many ways we can fetch 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 for Each loop. [JDK 1.5]

4) By using `Enumeration(I)` interface [JDK 1.0]

5) By using `Iterator` interface [JDK 1.2]

6) By using `ListIterator` interface [JDK 1.2]

\*7) By using `SplitIterator` [JDK 1.8]

\*8) By using `forEach(Consumer<T> cons)`

\*9) By using Method Reference(:)

Note : Among all these `Enumeration`, `Iterator`, `ListIterator`, `SplitIterator` are the cursors (Cursor means 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.

```
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.

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.

```
public Iterator iterator();
```

Example :

```
-----  
Iterator itr = v.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.
```

-----  
ListIterator interface :

-----  
It is a predefined interface available in java.util package and it is the sub interface of Iterator available from JDK 1.2v.

It is used to retrieve the Collection object in both the direction i.e in forward direction as well as in backward direction.

```
public ListIterator listIterator();
```

Example :

```
-----  
ListIterator lit = v.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 ListIterator interface

-----  
SplitIterator :

-----  
SplitIterator interface :

-----  
It is a predefined interface available in java.util package from java 1.8 version.

It is a cursor through which we can fetch the elements from the Collection [Collection, array, Stream]

It is the combination of hasNext() and next() method.

It is using forEachRemaining(Consumer <T>) method for fetching the elements.

---

By using forEach() method :

---

From java 1.8 onwards every collection class provides a method forEach() method, this method takes Consumer functional interface as a parameter.  
This method is available in java.lang.Iterable interface

---

```
package com.nit.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("Banana");
        fruits.add("Gauva");

        System.out.println("USING TOSTRING METHOD :");
        System.out.println(fruits.toString());

        System.out.println("USING ORDINARY FOR LOOP :");
        for(int i=0; i<fruits.size(); i++)
        {
            System.out.println(fruits.get(i));
        }

        System.out.println("USING FOR EACH LOOP :");

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

        System.out.println("BY USING ENUMERATION INTERFACE :");

        Enumeration<String> ele = fruits.elements();

        while(ele.hasMoreElements())
    }
}
```

```

{
    System.out.println(ele.nextElement());
}

System.out.println("BY USING ITERATOR INTERFACE :");

Iterator<String> itr = fruits.iterator();
while(itr.hasNext())
{
    System.out.println(itr.next());
}

System.out.println("BY USING LISTITERATOR INTERFACE :");

ListIterator<String> lstr = fruits.listIterator();
System.out.println("IN FORWARD DIRECTION :");
while(lstr.hasNext())
{
    System.out.println(lstr.next());
}

System.out.println("IN BACKWARD DIRECTION :");
while(lstr.hasPrevious())
{
    System.out.println(lstr.previous());
}

System.out.println("BY USING SPLITITERATOR :");

Spliterator<String> splitr = fruits.spliterator();
splitr.forEachRemaining(fruit -> System.out.println(fruit));

System.out.println("BY USING FOR EACH METHOD :");
fruits.forEach(fruit -> System.out.println(fruit.toUpperCase()));

System.out.println("BY USING METHOD REFERENCE :");
fruits.forEach(System.out::println);

}

```

18-06-2024

---

How forEach(Consumer<T> cons) method is working internally ?

---

forEach(Consumer<T> cons) method internally uses for-each loop as well as it is accepting Consumer as a parameter.

Case 1 :

---

```

package com.ravi.collection;

import java.util.Vector;

```

```

import java.util.function.Consumer;

public class ForEachInternal {

    public static void main(String[] args)
    {
        Vector<String> fruits = new Vector<>();
        fruits.add("Orange");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Banana");
        fruits.add("Gauva");

        //Anonymous inner class
        Consumer<String> cons = new Consumer<String>()
        {
            @Override
            public void accept(String t)
            {
                System.out.println(t.toUpperCase());
            }
        };
        fruits.forEach(cons);
    }
}

```

Case 2 :

---

```

package com.ravi.collection;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachInternal {

    public static void main(String[] args)
    {
        Vector<String> fruits = new Vector<>();
        fruits.add("Orange");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Banana");
        fruits.add("Gauva");

        //Anonymous inner class
        Consumer<String> cons = fruit -> System.out.println(fruit.toUpperCase());

        fruits.forEach(cons);
    }
}

```

Case 3 :

---

```
-----  
package com.ravi.collection;  
  
import java.util.Vector;  
import java.util.function.Consumer;  
  
public class ForEachInternal {  
  
    public static void main(String[] args)  
    {  
        Vector<String> fruits = new Vector<>();  
        fruits.add("Orange");  
        fruits.add("Apple");  
        fruits.add("Mango");  
        fruits.add("Banana");  
        fruits.add("Gauva");  
  
        fruits.forEach(fruit -> System.out.println(fruit.toUpperCase()));  
  
        fruits.forEach(System.out::println);  
    }  
}
```

Vector :

Vector :

-----  
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.

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 retrieval operation.

It also stores the elements on index basis. It is dynamically growable with initial capacity 10. The next capacity will be 20 i.e double of the first capacity.

new capacity = current capacity \* 2;

It implements List, Serializable, Clonable, RandomAccess interfaces.

Constructors in Vector :

-----

We have 4 types of Constructor in Vector

1) Vector v1 = new Vector();

It will create the vector object with default capacity is 10

2) Vector v2 = new Vector(int initialCapacity);

Will create the vector object with user specified capacity.

3) Vector v3 = new Vector(int initialCapacity, int capacityIncrement);

Eg :- Vector v = new Vector(1000,5);

Initially It will create the Vector Object with initial capacity 1000 and then when the capacity will be full then increment by 5 so the next capacity would be 1005, 1010 and so on.

4) Vector v4 = new Vector(Collection c);

We can achieve loose coupling.

---

Programs on Vector :

---

Program that describes that default capacity of Vector is 10.

```
package com.nit.vector_demo;  
  
import java.util.Vector;  
  
public class VectorDemo1 {  
  
    public static void main(String[] args)  
    {  
        Vector<Integer> v1 = new Vector<>();  
        System.out.println(v1.capacity());  
    }  
}
```

---

//Program on capacityIncrement

```
package com.nit.vector_demo;  
  
import java.util.Vector;  
  
public class VectorDemo2 {  
  
    public static void main(String[] args)  
    {  
        Vector<Integer> v1 = new Vector<>(100,5);  
  
        for(int i=0; i<=99; i++)  
        {  
            v1.add(i);  
        }  
        System.out.println("After adding 100 elements capacity is :" +v1.capacity());  
  
        v1.add(100);  
        System.out.println("After adding 101th element capacity is :" +v1.capacity());
```

```
    }

}

-----
package com.nit.vector_demo;

import java.util.Collections;
import java.util.Vector;

public class VectorDemo3 {

    public static void main(String[] args)
    {
        Vector<String> fruits = new Vector<>();
        fruits.add("Orange");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Banana");
        fruits.add("Gauva");

        Collections.sort(fruits);

        System.out.println("Elements are :" + fruits);

        Collections.reverse(fruits);

        System.out.println("Elements are :" + fruits);

        System.out.println("Size is :" + fruits.size());

        System.out.println("Is list is empty :" + fruits.isEmpty());

        fruits.clear();

        System.out.println("Is list is empty :" + fruits.isEmpty());
    }
}
```

```
}

-----
//Collections class method :
package com.nit.vector_demo;

import java.util.Collections;
import java.util.Vector;

public class VectorDemo4 {

    public static void main(String[] args)
    {

        Vector<Integer> v1 = new Vector<>();
        v1.add(78);
        v1.add(19);
    }
}
```

```

v1.add(38);
v1.add(45);
v1.add(12);
v1.add(90);

v1.remove(0); //based on index
v1.remove(Integer.valueOf(90)); //based on Object

System.out.println("Minimum :" + Collections.min(v1));
System.out.println("Maximum :" + Collections.max(v1));

System.out.println(v1);

}

}

-----//How to work with Custom Object ?-----
```

---

```

package com.nit.vector_demo;

import java.util.Vector;

record Product(Integer productId, String productName, Double productPrice)
{
```

---

```

}

public class VectorDemo5 {

    public static void main(String[] args)
    {
        Vector<Product> listOfProducts = new Vector<>();
        listOfProducts.add(new Product(333, "Laptop", 89000.89));
        listOfProducts.add(new Product(222, "Camera", 12000.89));
        listOfProducts.add(new Product(111, "Mobile", 29000.89));

        listOfProducts.forEach(System.out::println);

    }
}
```

---

```

}

-----package com.ravi.vector;

import java.util.Scanner;
import java.util.Vector;

public class VectorDemo2
{
    public static void main(String[] args)
    {
        Vector<String> toDoList = new Vector<>();
```

```
Scanner scanner = new Scanner(System.in);

int choice;
do
{
    System.out.println("ToDo 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("ToDo 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();
            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();
    }
}
```

---

19-06-2024

---

Stack :

---

```
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 `EmptyStackException`, if Stack is empty and we want to fetch the element.

It has only one constructor as shown below

```
Stack s = new Stack();
```

---

Methods :

---

`E push(Object o)` :- To insert an element in the bottom of the Stack.

`E pop()` :- To remove and return the element from the top of the Stack.

`E peek()` :- Will fetch the element from top of the Stack without removing.

`boolean empty()` :- Verifies whether the stack is empty or not (return type is boolean)

`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.isEmpty());
    }

    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)
    {
        Stack<String> stk= new Stack<>();
        stk.push("Apple");
        stk.push("Grapes");
        stk.push("Mango");
        System.out.println("Offset Position is : " + stk.search("Mango")); //1

        System.out.println("Offser Position is : " + stk.search("Banana")); //-1
        System.out.println("Is stack empty ? "+stk.empty()); //false

        System.out.println("Index Position is : " + stk.indexOf("Mango")); //2
    }
}

```

-----ArrayList :

```

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 elements and null values.

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

$$\text{new capacity} = (\text{current capacity} * 3)/2 + 1$$

\*All the methods declared inside an ArrayList is not synchronized so multiple thread can access the method of ArrayList.

\*It is highly suitable for fetching or retrieving operation when duplicates are allowed and Thread-safety is not required.

It implements List, Serializable, Clonable, RandomAccess interfaces

Constructor of ArrayList :

In ArrayList we have 3 types of Constructor:

Constructor of ArrayList :

We have 3 types of Constructor in 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  
(Copying one Collection data to another)

Program that describes, Performance wise ArrayList is better than Vector:

```
package com.nit.collection;

import java.util.ArrayList;
import java.util.Vector;

public class PerformanceComparison {

    public static void main(String[] args)
    {
        long startTime = System.currentTimeMillis(); //2ms

        ArrayList<Integer> al = new ArrayList<>();

        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");

        startTime = System.currentTimeMillis();

        Vector<Integer> v1 = new Vector<>();

        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 :

-----  
From the above program, It is clear that performance wise ArrayList is more better than Vector.

System is a predefined class available in java.lang package and it contains a predefined static method currentTimeMillis() , the return type of this method is long, actually it returns the current time of the system in ms.

```
public static native long currentTimeMillis()

-----
package com.ravi.arraylist;

import java.util.*;
public class ArrayListDemo
{
    public static void main(String... a)
    {
        ArrayList<String> arl = new ArrayList<>();//Generic type
        arl.add("Apple");
        arl.add("Orange");
        arl.add("Grapes");
        arl.add("Mango");
        arl.add("Guava");
        arl.add("Mango");

        Collections.sort(arl);
        System.out.println("In Ascending Order");
        arl.forEach(System.out::println);

        Collections.reverse(arl);
        System.out.println("In reverse Order");
        arl.forEach(System.out::println);

    }
}

import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        ArrayList<Employee> arl = new ArrayList<>();
        arl.add(new Employee(222));
        arl.add(new Employee(111));
        arl.add(new Employee(333));
```

```

        arl.add(new Employee(100));

        Collections.sort(arl);

        System.out.println(arl);
    }

}

class Employee implements Comparable<Employee>
{
    private int eid;
    public Employee(int eid)
    {
        this.eid = eid;
    }

    public int compareTo(Employee e2)
    {
        return this.eid - e2.eid;
    }

    public String toString()
    {
        return ""+this.eid;
    }
}

-----
import java.util.*;
public class ArrayListDemo
{
    public static void main(String... a)
    {
        ArrayList<String> arl = new ArrayList<>();//Generic type
        arl.add("Apple");
        arl.add("Orange");
        arl.add("Grapes");
        arl.add("Mango");
        arl.add("Guava");
        arl.add("Mango");

        System.out.println("Contents :" + arl); //toString() [Apple,....]

        arl.remove(2); //based on the index position [List]
        arl.remove("Guava"); //based on the Object [Collection]

        System.out.println("Contents After Removing :" + arl);
        System.out.println("Size of the ArrayList:" + arl.size());

        Collections.sort(arl);

        arl.forEach(System.out::println);
    }
}
-----
```

20-06-2024

---

### Working with Custom object using ArrayList :

---

The main purpose of Collection to work with custom object so we can hold multiple custom objects in a single collection variable.

```
package com.ravi.arraylist;

import java.util.ArrayList;

record Customer(Integer customerId, String customerName, Double custBill)
{
}

public class ArrayListDemo1
{
    public static void main(String[] args)
    {
        ArrayList<Customer> al = new ArrayList<>();
        al.add(new Customer(333, "Rohan", 29789.90));
        al.add(new Customer(111, "Satish", 23789.90));
        al.add(new Customer(222, "Aryan", 25789.90));
        al.add(new Customer(444, "Zuber", 27789.90));

        //filter all the customer records from the Collection
        // where Customer bill > 25000
        al.stream().filter(cust -> cust.custBill()>25000).forEach(System.out::println);
    }
}
```

---

```
package com.ravi.arraylist;

//Program to merge and retain of two collection
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(x -> System.out.println(x.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));
}
}

```

---

#### Immutable List :

---

We can make our list as a immutable (unchanged) list by using the following two techniques :

- 1) Arrays.asList(T ...x) : Here asList() is a predefined static method of Arrays class. It will make our array fixed of length so further we can't perform modification, otherwise we will get java.lang.UnsupportedOperationException
- 2) List.of(E1... x) : of(E1...x) is a overloaded static of List interface available from JDK 9v which will create immutable list so further we can't perform any modification, otherwise we will get java.lang.UnsupportedOperationException

```

package comn.ravi.collection;

import java.util.Arrays;
import java.util.List;

public class ImmutableList {

    public static void main(String[] args)
    {
        List<String> asList = Arrays.asList("A","B","C","D");
        asList.add("E"); //UnsupportedOperationException

        List<Integer> listOfNumbers = List.of(1,2,3,4,5,6);
        listOfNumbers.add(7); //UnsupportedOperationException

    }
}

```

```
}
```

---

```
//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");

        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 De-serialization on ArrayList object :

---

```
//Serialization and De-serialization on ArrayList Object
package com.ravi.arraylist;
```

```
import java.io.EOFException;
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
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws IOException, ClassNotFoundException
```

```

{
    ArrayList<String> al = new ArrayList<>();
    al.add("Hyderabad");
    al.add("Pune");
    al.add("Delhi");
    al.add("Chennai");

    var fout = new FileOutputStream("C:\\new\\City.txt");
    var oos = new ObjectOutputStream(fout);

    try(oos; fout)
    {
        oos.writeObject(al);
        System.out.println("Object stored Successfully");
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }

    //De-serialization
    var fin = new FileInputStream("C:\\new\\City.txt");
    var ois = new ObjectInputStream(fin);

    try(fin;ois)
    {
        ArrayList<String> listOfCity = (ArrayList<String>)ois.readObject();
        System.out.println(listOfCity);
    }
    catch(EOFException e)
    {
        System.err.println("End of File reached");
    }
}

//Serialization and De-serialization on Product object
-----
package comn.ravi.collection;

import java.io.EOFException;
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 Product(Integer pid, String pName) implements Serializable
{

```

```

}

public class SerializationDeserialization {

    public static void main(String[] args) throws IOException, ClassNotFoundException
    {
        ArrayList<Product> listOfProduct = new ArrayList<>();
        listOfProduct.add(new Product(111, "Mobile"));
        listOfProduct.add(new Product(222, "Camera"));
        listOfProduct.add(new Product(333, "HeadPhone"));

        var fout = new FileOutputStream("C:\\new\\Product.txt");
        var oos = new ObjectOutputStream(fout);

        try(oos; fout)
        {
            oos.writeObject(listOfProduct);
            System.out.println("Product Object stored Successfully");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        //De-serialization
        var fin = new FileInputStream("C:\\new\\Product.txt");
        var ois = new ObjectInputStream(fin);

        try(fin;ois)
        {
            ArrayList<String> productList = (ArrayList<String>)ois.readObject();
            System.out.println(productList);
        }
        catch(EOFException e)
        {
            System.err.println("End of File reached");
        }
    }
}

```

---

```
public void ensureCapacity(int minimumCapacity) :
```

---

As we know we don't have capacity() method with ArrayList class.

In ArrayList, once we create the object then the default capacity is 10. After object creation, if we want to resize our arraylist capacity then we can use ensureCapacity(int minimumCapacity) method of ArrayList class.

This method will ensure that the ArrayList must hold the minimum capacity element which is specified as a parameter to ensureCapacity()

Even after resizing, It is dynamically Growable.

```
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.LinkedList;

public class ArrayListDemo5
{
    public static void main(String[] args)
    {
        ArrayList<String> city= new ArrayList<>();

        city.ensureCapacity(3); //resize the capacity of ArrayList
        city.add("Hyderabad");
        city.add("Mumbai");
        city.add("Delhi");

        city.add("Kolkata");
        System.out.println("ArrayList: " + city);
    }
}
```

---

```
-----
```

```
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"); //add(int index, Object o)method of List interface
        al.add(1,"Naresh");
        al.add(null);
        al.add(11);
        System.out.println(al); //12 Naresh Ravi 12 Hyderabad
    }
}
```

---

```
-----
```

```
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.List;

class Professor
{
```

```
private String name;
private String specialization;

public Professor(String name, String specialization)
{
    this.name = name;
    this.specialization = specialization;
}

public String getName()
{
    return name;
}

public String getSpecialization()
{
    return specialization;
}
}

class Department
{
    private String name;
    private final List<Professor> professors; // Department "HAS-A" relationship with Professor

    public Department(String name)
    {
        this.name = name;
        this.professors = new ArrayList<>();
    }

    public void addProfessor(Professor prof)
    {
        professors.add(prof);
    }

    public String getName()
    {
        return name;
    }

    public List<Professor> getProfessors()
    {
        return professors;
    }
}

public class ArrayListDemo7
{
    public static void main(String[] args)
    {
        Professor prof1 = new Professor("Scott", "Java");
        Professor prof2 = new Professor("Rahul", "Python");
        Professor prof3 = new Professor("Samir", ".Net");
    }
}
```

```
Department csd = new Department("Computer Science");
csd.addProfessor(prof1);
csd.addProfessor(prof2);
csd.addProfessor(prof3);

// Accessing properties through the "HAS-A" relationship
System.out.println("Department Name: " + csd.getName());

System.out.println("Professors in " + csd.getName() + ":");

csd.getProfessors().forEach(prof -> System.out.println("-> " + prof.getName() + " (" +
prof.getSpecialization() + ")"));

}

-----
```

21-06-2024

Limitation of ArrayList :

Time Complexity of ArrayList :

The time complexity of ArrayList to insert and delete an element from the middle would be O(n) because 'n' number of elements will be re-located so it is not a good choice to perform insertion and deletion operation in the middle of the List.

On the other hand time complexity of ArrayList to retrieve an element from the List would be O(1) because by using get(int index) method we can retrieve the element randomly from the list.

ArrayList class

implements RandomAccess marker interface which provides the facility to fetch the elements Randomly.

[21-JUN-24]

To avoid this we introduced LinkedList.

LinkedList :

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable
```

It is a predefined class available in java.util package under List interface.

It is ordered by index position like ArrayList except the elements (nodes) are doubly linked to one another. This linkage provide us new method for adding and removing the elements from the middle of LinkedList.

\*The important thing is, LikedList may iterate more slowly 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.

ArrayList is using Array data structure but LinkedList class is using LinkedList data structure.

Constructor:

-----  
It has 2 constructors

- 1) `LinkedList list1 = new LinkedList();`  
It will create a LinkedList object with 0 capacity.
- 2) `LinkedList list2 = new LinkedList(Collection c);`  
Interconversion between the collection

Methods of LinkedList class:

- 
- 1) `void addFirst(Object o)`
  - 2) `void addLast(Object o)`
  - 3) `Object getFirst()`
  - 4) `Object getLast()`
  - 5) `Object removeFirst()`
  - 6) `Object removeLast()`

Note :- It stores the elements in non-contiguous memory location.

The time complexity for insertion and deletion is  $O(1)$

The time complexity for searching  $O(n)$

-----

```
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[])
    {
        List<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));
    }
}
```

```
-----
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");
        list.add("Rahul");
        list.addLast("Anand");

        System.out.println(list.getFirst());
        System.out.println(list.getLast());

        list.removeFirst();
        list.removeLast();

        System.out.println(list);
    }
}
```

---

```
package com.ravi.linked_list;
//ListIterator methods
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);
    }
}
```

}

Note :- By using ListIterator, we can also add, remove and set the elements.

---

```
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 elementToDelete = scanner.nextInt();
                        boolean remove = linkedList.remove(Integer.valueOf(elementToDelete));

                        if(remove)
                        {
                            System.out.println("Element "+elementToDelete+" is deleted Successfully" );
                        }
                        else
                        {
                            System.out.println(elementToDelete+" not available in the LinkedList");
                        }
                    }
            }
        }
    }
}
```

```

        }
    }
    break;
    case 3:
        System.out.println("Elements in the linked list.");
        System.out.println(linkedList); // []
        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.HashSet;
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);
        System.out.println(".....");

    }
}

```

---

22-06-2024

---

```

import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

class Dog
{
    private String name;
    Dog(String n)
    {
        name = n;
    }

    public String getName()

```

```

        {
            return this.name;
        }

    public String toString()
    {
        return this.name;
    }
}
public class LinkedListDemo5
{
    public static void main(String[] args)
    {
        List<Dog> d = new LinkedList<>();
        Dog dog = new Dog("Tiger");
        d.add(dog);
        d.add(new Dog("Tommy"));
        d.add(new Dog("Rocky"));

        Iterator<Dog> i3 = d.iterator();
        i3.forEachRemaining(x -> System.out.println(x.getName().toUpperCase())); //java
8

```

```

        System.out.println("size " + d.size());
        System.out.println("Get 1st Position Object " + d.get(1).getName());

    }
}
```

---

```

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<>();

        // Adding elements to the front of the deque
        deque.addFirst("Ravi");
        deque.addFirst("Raj"); //Raj Ravi Pallavi Sweta

        // Adding elements to the back of the deque
        deque.addLast("Pallavi");
        deque.addLast("Sweta");
    }
}
```

```
System.out.println("Deque: " + deque); // Ravi P
```

```
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<E> interface :

---

Set(I)

---

Set interface is the sub interface of Collection(I) which came from JDK 1.2v

Set interface never accept duplicate elements, Here internally equals(Object obj) method is working to verify two object are identical or not, if identical then it will accept only 1.

Set interface does not use any indexing technique.

ListIterator interface does not work with Set interface.

Set interface uses methods of Collection interface as well as It has more methods from java 9.

---

Set interface Hierarchy :

---

Available in the paint diagram (22-JUNE)

---

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 structure, Hashtable data structure internally uses Bucket data structure.

Here elements are inserted by using hashing algorithm so the time complexity to insert, delete and search an element would be O(1)

---

HashSet (UNSORTED, UNORDERED , NO DUPLICATES)

---

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Clonable, 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 heterogeneous 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 null value.

HashSet methods are not synchronized.

HashSet is used for fast searching operation.

It contains 4 types of constructors

1) HashSet hs1 = new HashSet();

It will create the HashSet Object with default capacity is 16. The default load fator or Fill Ratio is 0.75 (75% of HashSet is filled up then new HashSet Object will be created having double capacity)

2) HashSet hs2 = new HashSet(int initialCapacity);

will create the HashSet object with user specified capacity

3) HashSet hs3 = new HashSet(int initialCapacity, float loadFactor);

we can specify our own initialCapacity and loadFactor(by default load factor is 0.75)

4) HashSet hs = new HashSet(Collection c);

Interconversion of Collection

---

```
//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.forEach(str-> System.out.println(str));
    }
}
```

---

24-06-2024

---

```
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));
    }
}
```

By default hashCode() value of null is 0, because we can't call hashCode() method on null, otherwise we will get NPE.

---

hashCode() and equals(Object obj) both are overridden in String class so, if we compare two String based on the equals(Object obj) methods and if returns true then hashCode of both the String must be same.

```
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));
    }
}
```

---

Program to calculate custom hashCode for String object :

---

```
package com.ravi.custom_hash_code;

public class CustomStringHashCode
{
    public static int customHashCode(String str)
    {
        if (str == null)
        {
            return 0; // default value for null is 0
        }

        int hashCode = 0;

        for (int i = 0; i < str.length(); i++)
        {
            char charValue = str.charAt(i);
            hashCode = 31 * hashCode + charValue;
        }

        return hashCode;
    }
}
```

```
public static void main(String[] args)
{
    String exampleString = "Ravi";

    // Using the built-in hashCode method
    int hashCodeBuiltIn = exampleString.hashCode();
    System.out.println("Built-in hashCode: " + hashCodeBuiltIn);

    // Using the customHashCode method
    int customHashCode = customHashCode(exampleString);
    System.out.println("Custom hashCode: " + customHashCode);
}
```

---

```
package com.ravi.set_demo;
```

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

public class HashSetDemo2
{
    public static void main(String[] args)
    {
        Boolean br[] = new Boolean[5];

        Set<Object> set = new HashSet<>();
        br[0] = set.add(12);
        br[1] = set.add("NIT");
        br[2] = set.add(new String("NIT"));
        br[3] = set.add(12);
        br[4] = set.add(true);

        System.out.println(Arrays.toString(br));

        set.forEach(val -> System.out.println(val));

        if(set.contains(12))
        {
            System.out.println("Object 12 is available");
        }
        else
        {
            System.err.println("Object 12 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();
    }
}
-----
```

LinkedHashSet :

-----

LinkedHashSet :

-----

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 ordered 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 heterogeneous 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.LinkedHashSet;
```

```
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 we can't perform sorting operation on HashSet and LinkedHashSet object so, to provide sorting facility, java software people has provided SortedSet interface from JDK 1.2v

By using SortedSet interface we can provide default natural sorting order that means If we are sorting String then by default sorting order would be in ascending order OR alphabetical order, on the other hand if we are sorting numbers then sorting order would be ascending order.

In order to perform sorting we have two interfaces which are as follows

- 1) java.lang.Comparable<T> interface
- 2) java.util.Comparator<T> interface

\*\*\* What is the difference between Comparable<T> and Comparator<T> interface?

Difference is available in the paint diagram (24-JUNE)

---

**Program on Comparable interface :**

---

2 files :

---

Employee.java(R)

---

package com.ravi.comparable\_demo;

```

public record Employee(Integer employeeId, String employeeName, Double empSalary)
implements Comparable<Employee>
{
    //Sorting the Employee record based on the Name
    @Override
    public int compareTo(Employee e2)
    {
        return this.employeeName.compareTo(e2.employeeName);
    }

}

```

### EmployeeComparable.java(C)

```

-----
package com.ravi.comparable_demo;

import java.util.ArrayList;
import java.util.Collections;

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

        ArrayList<Employee> al = new ArrayList<>();
        al.add(new Employee(333, "Aryan", 29455D));
        al.add(new Employee(222, "Raj", 23455D));
        al.add(new Employee(111, "Zuber", 27455D));

        Collections.sort(al);

        System.out.println("Sorting based on the Name :");
        al.forEach(System.out::println);

    }
}

```

Note : In the above program we are sorting the Employee object using employeeName, In order to compare two Strings we have a predefined method in String class called compareTo(String str) which compares two String character by character based on the UNICODE value known as Lexicographical Comparsion.

```
public int compareTo(String str);
```

---

Limitation of Comparable<T> interface :

---

We have 2 limitations :

- 1) We need to modify the original source code (BLC code) to implement Comparable interface.
- 2) We can provide only one sorting logic.

To avoid the above said drawback, java software people has provided Comparator<T> interface.

25-06-2024

-----  
Program on Comparator<T> interface :

-----  
2 files :

-----  
Product.java(R)

-----  
package com.ravi.comparator;

```
public record Product(Integer productId, String productName, Double productPrice)
{
}
```

-----  
ProductComparator.java(C)

-----  
package com.ravi.comparator;

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
```

```
public class ProductComparator
{
```

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

```
        ArrayList<Product> al = new ArrayList<>();
        al.add(new Product(333, "Camera", 34890.89));
        al.add(new Product(111, "Mobile", 35890.89));
        al.add(new Product(222, "Laptop", 84890.89));
```

```
//Anonymous inner class
```

```
    Comparator<Product> comID = new Comparator<Product>()
    {
```

```
        @Override
```

```
        public int compare(Product p1, Product p2)
        {
```

```
            return p1.productId() - p2.productId();
        }
    };

```

```
Collections.sort(al, comID);
```

```
System.out.println("Sorting Based on the ID :");
```

```
al.forEach(System.out::println);
```

```
//Sorting Based on the Product name :
```

```

        Comparator<Product> cmpName = (p1,p2)->
p1.productName().compareTo(p2.productName()));

        Collections.sort(al, cmpName);
        System.out.println("Sorting Based on the Name :");
        al.forEach(System.out::println);
    }
}

-----
```

//Program that describes, how to sort Integer object in descending order by using Comparator

```

package com.ravi.comparator;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class DescendingInteger {

    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<>();
        al.add(45);
        al.add(90);
        al.add(12);
        al.add(10);

        Comparator<Integer> comDes = (i1,i2)-> i2.compareTo(i1);

        Collections.sort(al, comDes);

        al.forEach(System.out::println);
    }
}
```

TreeSet<E> :

TreeSet :

```

-----
```

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, TreeMap and PriorityQueue are the three sorted collection in the entire Collection Framework so these classes never accepting non comparable objects.

It will sort the elements in natural sorting order i.e ascending order in case of number , and alphabetical order or Dictionary order in the case of String. In order to sort the elements according to user choice, It uses Comparable/Comparator interface.

It does not accept duplicate and null value (java.lang.NullPointerException) as well as duplicate value.

It does not accept non comparable type of data if we try to insert it will throw a runtime exception i.e java.lang.ClassCastException

TreeSet implements NavigableSet.

NavigableSet extends SortedSet.

It contains 4 types of constructors :

- 
- 1) TreeSet t1 = new TreeSet();  
create an empty TreeSet object, elements will be inserted in a natural sorting order.
  - 2) TreeSet t2 = new TreeSet(Comparator c);  
Customized sorting order
  - 3) TreeSet t3 = new TreeSet(Collection c);
  - 4) TreeSet t4 = new TreeSet(SortedSet s);
- 

```
package com.ravi.comparator;

import java.util.TreeSet;

class Dog
{
    private String name;

    public Dog(String name)
    {
        super();
        this.name = name;
    }

    @Override
    public String toString() {
        return "Dog [name=" + name + "]";
    }

    public String getName() {
        return name;
    }
}

public class DescendingInteger {

    public static void main(String[] args)
    {
        TreeSet<Dog> ts = new TreeSet<>((d1,d2)->
d2.getName().compareTo(d1.getName()) );
        ts.add(new Dog("Tiger"));
    }
}
```

```
ts.add(new Dog("Tommy"));

System.out.println(ts);

}

-----  
//program that describes 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);
    }
}
```

Note : Integer and String both classes are implementing java.lang.Comparable so internally compareTo() is invoked to compare the Integer and String Object.

---

```
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.

```

public Iterator descendingIterator()

-----
import java.util.*;
public class TreeSetDemo2
{
    public static void main(String[] args)
    {
        Set<String> t = new TreeSet<>();
        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));

        //From 1.8 to replace hasNext() and next() method
    }
}

-----
import java.util.*;

public class TreeSetDemo3
{
    public static void main(String[] args)
    {
        Set<Character> t = new TreeSet<>();
        t.add('A');
        t.add('C');
        t.add('B');
        t.add('E');
        t.add('D');
        Iterator<Character> iterator = t.iterator();
        iterator.forEachRemaining(x -> System.out.println(x));
    }
}

-----
package com.ravi.comparator;

import java.util.TreeSet;

```

```

record Employee(int empNo, String empName, int empAge)
{
}

public class EmployeeSortingOrder {
    public static void main(String[] args)
    {
        TreeSet<Employee> ts1 = new TreeSet<>((e1,e2)-> e1.empNo()-e2.empNo());

        ts1.add(new Employee(401, "Bobby", 23));
        ts1.add(new Employee(101, "Zaheer", 24));
        ts1.add(new Employee(201, "Aryan", 27));
        ts1.add(new Employee(301, "Pooja", 26));
        System.out.println("Sorting Based on the ID in ascending order");
        ts1.forEach(i -> System.out.println(i));

        TreeSet<Employee> ts2 = new TreeSet<>((e1,e2)-> e2.empNo()-e1.empNo());

        ts2.add(new Employee(401, "Bobby", 23));
        ts2.add(new Employee(101, "Zaheer", 24));
        ts2.add(new Employee(201, "Aryan", 27));
        ts2.add(new Employee(301, "Pooja", 26));
        System.out.println("Sorting Based on the name in descending order");
        ts2.forEach(i -> System.out.println(i));

        TreeSet<Employee> ts3 = new TreeSet<>((e1,e2)->
e1.empName().compareTo(e2.empName()) );
        ts3.add(new Employee(401, "Bobby", 23));
        ts3.add(new Employee(101, "Zaheer", 24));
        ts3.add(new Employee(201, "Aryan", 27));
        ts3.add(new Employee(301, "Pooja", 26));
        System.out.println("Sorting Based on the ID in ascending order");
        ts3.forEach(i -> System.out.println(i));

        TreeSet<Employee> ts4 = new TreeSet<>((e1,e2)->
e2.empName().compareTo(e1.empName()) );
        ts4.add(new Employee(401, "Bobby", 23));
        ts4.add(new Employee(101, "Zaheer", 24));
        ts4.add(new Employee(201, "Aryan", 27));
        ts4.add(new Employee(301, "Pooja", 26));
        System.out.println("Sorting Based on the name in descending order");
        ts4.forEach(i -> System.out.println(i));
    }
}

-----
package com.ravi.comparator;

```

```
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo {

    public static void main(String[] args)
    {
        SortedSet<Integer> ss = new TreeSet<>();
        ss.add(12);
        ss.add(45);
        ss.add(90);

        TreeSet<Integer> ts1 = new TreeSet<>(ss);
        ts1.add(100);
        ts1.add(200);

        System.out.println(ts1);

    }
}
```

---

26-06-2024

---

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(I) :

---

By using SortedSet interface we can find out the range of values but we have one limitation i.e navigation is not possible among the elements.

In order to provide navigation among the elements, java software people has introduced a separate interface called NavigableSet.

NavigableSet extends SortedSet

---

```

import java.util.*;

public class NavigableSetDemo
{
    public static void main(String[] args)
    {
        NavigableSet<Integer> ns = new TreeSet<>();
        ns.add(1);
        ns.add(2);
        ns.add(3);
        ns.add(4);
        ns.add(5);
        ns.add(6);

        System.out.println("lower(3): " + ns.lower(3));//Just below than the specified element or null
        System.out.println("floor(3): " + ns.floor(3)); //Equal less or null
        System.out.println("higher(3): " + ns.higher(3));//Just greater than specified element or null
        System.out.println("ceiling(3): " + ns.ceiling(3));//Equal or greater or null
    }
}

```

---

Map interface :

---

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 interface is not the part the Collection.

Before Map interface We had Dictionary(abstract class) class and it is extended by Hashtable class in JDK 1.0V

Map interface works with key and value pair introduced from 1.2V.

Here key and value both are objects.

Here key must be unique and value may be duplicate.

Each key and value pair is creating one Entry.(Entry is nothing but the combination of key and value pair)

```
interface Map<K,V>
{
    interface Entry<K,V>
    {
        //key and value
    }
}
```

How to represent this entry interface (Map.Entry in .java) [Map\$Entry in .class]

In Map interface whenever we have a duplicate key then the old key value will be replaced by new key(duplicate key) value.

forEach(BiConsumer cons) method is available in Map interface.

Iterator and ListIterator we can't use directly using Map.

---

Map interface Hierarchy :

---

The hierarchy is available in paint diagram 26-JUNE

---

Methods of Map Interface :

---

1) Object put(Object key, Object value) :- To insert one entry in the Map collection. It will return the old object key value if the key is already available(Duplicate key), If key is not available then it will return null.

2) putIfAbsent(Object key, Object value) :- It will insert an entry if and only if key is not present , if the key is already available then it will not insert the Entry to the Map Collection

3) Object get(Object key) :- It will return corresponding value of key, if the key is not present then it will return null.

4) Object getOrDefault(Object key, Object defaultValue) :- To avoid null value this method has been introduced, 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
- 

27-06-2024

---

Methods of Map interface to convert the Map into Collection :

---

- 1) public Set keySet() : It will provide all the keys which are available in the Map collection.
  - 2) public Collection values() : It will provide all the values of the map collection.
  - 3) public Set<Map.Entry> entrySet() : It will provide key and value pair both in a single object.
- 

\*\*\*\* 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 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.

Note :- equals(Object obj) method is invoked only when two keys are having same hashCode as well as their references are different.

- e) Actually by calling hashCode method we are not comparing the objects, we are just storing the objects in a group so the currently adding key object will be compared with its SAME HASHCODE GROUP objects, but not with all the keys which are available in the Map.

- f) The main purpose of storing objects into the corresponding group to decrease the number of comparison so the efficiency of the program will increase.
- g) To insert an entry in the HashMap, HashMap internally uses Hashtable data structure
- h) Now, for storing same hashCode object into a single group, hash table data structure internally uses one more data structure called Bucket.
- i) The Hashtable data structure internally uses Node class array object.
- j) The bucket data structure internally uses LinkedList data structure, It is a single linked list again implemented by Node class only.
- \*k) A bucket is group of entries of same hash code keys.

l) Performance wise LinkedList is not good to serach, so from java 8 onwards LinkedList is changed to Binary tree to decrease the number of comparison within the same bucket hashCode if the number of entries are greater than 8.

\* equals() and hashCode() method contract :

---

Both the methods are working together to find out the duplicate objects in the Map.

\*If equals() method invoked on two objects and it returns true then hashCode of both the objects must be same.

Note : IF TWO OBJECTS ARE HAVING SAME HASH CODE THEN IT MAY BE SAME OR DIFFERENT BUT if we call equals() methdo then hashCode of both the objects must be same.

```
package com.ravi.map;

import java.util.HashMap;

public class MapDemo {

    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 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);

    }

}
```

---

28-06-2024

---

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 1.2.

It gives us unsorted and Unordered map. when we need a map and we don't care about the order while iterating the elements through it then we should use HashMap.

It inserts the element based on the hashCode of the Object key using hashing technique [hasing alogorithhm]

It does not accept duplicate keys but value may be duplicate.

It accepts only one null key(because duplicate keys are not allowed) but multiple null values are allowed.

HashMap is not synchronized.

Time complexcity of search, insert and delete will be O(1)

We should use HashMap to perform fast searching opeartion.

It contains 4 types of constructor

1) HashMap hm1 = new HashMap();

It will create the HashMap Object with default capacity is 16. The default load fator or Fill Ratio is 0.75 (75% of HashMap is filled up then new HashMap Object will be created having double capacity)

2) HashMap hm2 = new HashMap(int initialCapacity);

will create the HashMap object with user specified capacity

3) HashMap hm3 = new HashMap(int initialCapacity, float loadFactor);

we can specify our own initialCapacity and loadFactor(by default load factor is 0.75%)

4)HashMap hm4 = new HashMap(Map m);

Interconversion of Map Collection

---

//Program that shows HashMap is unordered

import java.util.\*;

public class HashMapDemo

```

{
    public static void main(String[] a)
    {
        Map<String, String> map = new HashMap<>();
        map.put("Ravi", "12345");
        map.put("Rahul", "12345");
        map.put("Aswin", "5678");
        map.put(null, "6390");
        map.put("Ravi", "1529");
        map.put("Aamir", "890");

        System.out.println(map); //

        System.out.println(map.get(null));
        System.out.println(map.get("Virat"));

        map.forEach((k,v)-> System.out.println("Key is :" + k + " value is " + v));

    }
}

-----//Program to search a particular key and
value in the Map collection
import java.util.*;
public class HashMapDemo1
{
    public static void main(String args[])
    {
        HashMap<Integer, String> hm = new HashMap<>();
        hm.put(1, "JSE");
        hm.put(2, "JEE");
        hm.put(3, "JME");
        hm.put(4, "JavaFX");
        hm.put(5, null);
        hm.put(6, null);

        System.out.println("Initial map elements: " + hm);
        System.out.println("key 2 is present or not :" + hm.containsKey(2));

        System.out.println("JME is present or not :" + hm.containsValue("JME"));

        System.out.println("Size of Map : " + hm.size());
        hm.clear();
        System.out.println("Map elements after clear: " + hm);
    }
}

-----//Collection view methods [keySet(), values(), Set<Map.Entry> entrySet()]
import java.util.*;
public class HashMapDemo2
{
    public static void main(String args[])
    {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "C");
        map.put(2, "C++");
    }
}

```

```

        map.put(3, "Java");
        map.put(4, ".net");

        map.forEach((k,v)->System.out.println("Key :" + k + " Value :" + v));

        System.out.println("Return Old Object value :" + map.put(4, "Python"));

        Set keys = map.keySet();
        System.out.println("All keys are :" + keys);

        Collection values = map.values();
        System.out.println("All values are :" + values);

        for(Map.Entry m : map.entrySet())
        {
            System.out.println(m.getKey() + " = " + m.getValue());
        }

        System.out.println("Retrieving using Iterator :");

        Iterator itr= map.entrySet().iterator();

        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
    }

-----
import java.util.*;
public class HashMapDemo3
{
    public static void main(String args[])
    {
        HashMap<Integer,String> map = new HashMap<>(26,0.95f);
        map.put(1, "Java");
        map.put(2, "is");
        map.put(3, "best");
        map.remove(3); //will remove the complete Entry
        String val=(String)map.get(3);
        System.out.println("Value for key 3 is: " + val);
        map.forEach((k,v)->System.out.println(k + " : " + v));
    }
}

-----
//To merge two Map Collection (putAll)
import java.util.*;
public class HashMapDemo4
{
    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);

    newmap2.forEach((k,v)->System.out.println(k+" : "+v));
}
}
```

---

```
import java.util.*;
public class HashMapDemo5
{
    public static void main(String[] argv)
    {
        Map<String,String> map = new HashMap<>(9, 0.85f);
        map.put("key", "value");
        map.put("key2", "value2");
        map.put("key3", "value3");
        map.put("key7","value7");

        Set keys = map.keySet(); //keySet return type is Set
        System.out.println(keys ); //[]

        Collection val = map.values(); //values return type is collection
        System.out.println(val);

        map.forEach((k,v)-> System.out.println(k+" : "+v));

    }
}

//getOrDefault() method
import java.util.*;
public class HashMapDemo6
{
    public static void main(String[] args)
    {
        Map<String, String> map = new HashMap<>();
        map.put("A", "1");
        map.put("B", "2");
        map.put("C", "3");
        //if the key is not present, it will return default value .It is used to avoid null
        String value = map.getOrDefault("E","Key is not available");
        System.out.println(value);
        System.out.println(map);
    }
}
```

```

//interconversion of two HashMap
import java.util.*;
public class HashMapDemo7
{
    public static void main(String args[])
    {
        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);
    }
}

```

---

```

import java.util.*;
class Employee
{
    int eid;
    String ename;

    Employee(int eid, String ename)
    {
        this.eid = eid;
        this.ename = ename;
    }

    @Override
    public boolean equals(Object obj) //obj = e2
    {
        if(obj instanceof Employee)
        {
            Employee e2 = (Employee) obj; //downcasting

            if(this.eid == e2.eid && this.ename.equals(e2.ename))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            System.out.println("Comparison is not possible");
            return false;
        }
    }
}

```

```

public String toString()
{
    return "+eid+" "+ename;
}
}

public class HashMapDemo8
{
    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));
    }
}

```

Note : Why String and Wrapper classes are immutable :

---

In order to Work with HashMap 2 very important things are required

- 1) Key must be immutable (String and Wrapper classes are immutable)
- 2) Overriding of hashCode() and equals(Object obj) method is compulsory [All the Wrapper classes and String class has overridden hashCode() and equals(Object obj)]

---

**LinkedHashMap<K,V>**

---

**LinkedHashMap :**

---

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 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(int 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", "1456");  
        map.put("Samir", "1239");  
  
        map.forEach((k,v)->System.out.println(k+ " : "+v));  
    }  
}
```

Hashtable<K,V>

```
-----  
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Clonable,  
Serializable
```

It is predefined class available in java.util package under Map interface from JDK 1.0.

Like Vector, Hashtable is also form the birth of java so called legacy class.

It is the sub class of Dictionary class which is an abstract class.

\*The major difference between HashMap and Hashtable is, HashMap methods are not synchronized where as Hashtable 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);
}
}

```

---

29-06-2024

---

**WeakHashMap :**

---

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 and still it is 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 contains 4 types of Constructor :

---

1) WeakHashMap wm1 = new WeakHashMap();

Creates an empty WeakHashMap object with default capacity is 16 and load factor 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);

```
-----  
import java.util.*;  
public class WeakHashMapDemo  
{  
    public static void main(String args[]) throws Exception  
    {  
        WeakHashMap<Product, String> map = new WeakHashMap<>();  
  
        Product p = new Product();  
        map.put(p, "Laptop");  
  
        System.out.println(map); // {Product Properties Details = Laptop}  
  
        p = null;  
  
        System.gc();  
  
        Thread.sleep(5000);  
  
        System.out.println(map); // {}  
    }  
}  
  
class Product  
{  
    @Override  
    public String toString()  
    {  
        return "Product Properties Details";  
    }  
  
    @Override  
    public void finalize()  
    {  
        System.out.println("finalize method is called so Product object is eligible for GC");  
    }  
}
```

---

System generated hashCode for the Object :

---

System class has provided a predefined static method `identityHashCode(Object obj)` through which we can find out system generated hashCode value.

```
public static native int identityHashCode(Object obj);
```

```
public class Demo  
{  
    public static void main(String [] args)  
    {  
        String str = "nit";  
        System.out.println(str.hashCode());  
        System.out.println(System.identityHashCode(str));  
    }  
}
```

IdentityHashMap<K,V> :

---

```
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 search the bucket location and insert the entry there only.

So We should use IdentityHashMap where we need to check the reference or memory address instead of logical equality.

HashMap uses hashCode of the "Object key" to find out the bucket location 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.

It has three constructors, It does not contain loadFactor specific constructor.

---

```
import java.util.*;
public class IdentityHashMapDemo
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> hm = new HashMap<>();
        IdentityHashMap<String, Integer> ihm = new IdentityHashMap<>();

        hm.put("Ravi", 23);
        hm.put(new String("Ravi"), 24);

        ihm.put("Ravi", 23);
        ihm.put(new String("Ravi"), 27); //compares based on == operator

        System.out.println("HashMap size :" + hm.size());
        System.out.println(hm);
        System.out.println(".....");
        System.out.println("IdentityHashMap size :" + ihm.size());
        System.out.println(ihm);

    }
}
```

---

SortedMap<K,V>

---

It is a predefined interface available in java.util package under Map interface.

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.

---

## TreeMap<K,V>

---

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>,  
Clonable, Serializable
```

It is a predefined class available 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.

It does not accept null key but null value allowed.

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 sorting logic
  - 3) TreeMap tm3 = new TreeMap(Map m);
  - 4) TreeMap tm4 = new TreeMap(SortedMap m);
- 

```
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);  
  
        System.out.println(t);  
    }  
}
```

---

```
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 map = new TreeMap();
        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.collection;

import java.util.TreeMap;

public class TreeMapDemo {

    public static void main(String[] args)
    {

        TreeMap<Employee, String> tm1 = new TreeMap<Employee, String>((e1,e2)->e1.name().compareTo(e2.name()));

        tm1.put(new Employee(101, "Zaheer", 24), "Hyderabad");
        tm1.put(new Employee(201, "Aryan", 27), "Jamshedpur");
        tm1.put(new Employee(301, "Pooja", 26), "Mumbai");

        tm1.forEach((k,v)-> System.out.println(k+": "+v));
    }
}

```

```
 }  
}
```

```
record Employee(Integer id, String name, Integer age)  
{  
}  
}
```

---

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 :

---

01-07-2024

---

Properties class:

---

Properties class is used to maintain the data in the key-value form. It takes both key and value as a string.

Properties class is a subclass of Hashtable. It provides the methods to store properties in a properties file and to get the properties from the properties file, we should use load method of Properties class.

---

```
FileReader reader = new FileReader("db.properties");
```

```
Properties p = new Properties();
p.load(reader);
p.getProperty(String key); //read the corresponding key value
```

getProperty() is a non static method of Property class

---

System.getProperties() returns the all system properties.

Here we need to create a file with extension .properties

Example :

---

```
driver = oracle.jdbc.driver.OracleDriver
user = system
password = tiger
```

```
FileReader fr = new FileReader("db.properties");
```

```
Properties p = new Properties();
p.load(fr);
```

---

Here we need to create db.properties file

db.properties

---

```
driver = oracle.jdbc.driver.OracleDriver
user = scott
password = tiger
```

---

```
import java.util.*;
import java.io.*;
public class PropertiesExample1
{
    public static void main(String[] args) throws Exception
    {
        FileReader reader = new FileReader("db.properties");
    }
}
```

```
Properties p = new Properties();
p.load(reader);

System.out.println(p.getProperty("user"));
System.out.println(p.getProperty("password"));
System.out.println(p.getProperty("driver"));
}
```

Reading the data from the properties file dynamically.

---

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

```
import java.io.*;
public class PropertiesExample2
{
public static void main(String[] args) throws Exception
{
    Properties p=System.getProperties();
    Set set=p.entrySet();

    Iterator itr=set.iterator();
    while(itr.hasNext())
    {
        Map.Entry entry=(Map.Entry)itr.next();
        System.out.println(entry.getKey()+" = "+entry.getValue());
    }
}
```

---

#### Queue interface :-

---

- 1) It is sub interface of Collection(I)
- 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) From jdk 1.5 onwards LinkedList class implements Queue interface to handle the basic queue operations.

#### PriorityQueue :

---

```
public class PriorityQueue extends AbstractQueue implements Serializable
```

It is a predefined class in java.util package, available from Jdk 1.5 onwards.

It inserts the elements based on the priority HEAP (Using Binary tree).

The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

A priority queue does not permit null elements as well as it uses Binary tree to insert the elements.

It provides natural sorting order so we can't take non-comparable objects(heterogeneous types of Object)

The initial capacity of PriorityQueue is 11.

#### Constructor :

---

- 1) PriorityQueue pq1 = new PriorityQueue();
- 2) PriorityQueue pq2 = new PriorityQueue(int initialCapacity);

3) PriorityQueue pq3 = new PriorityQueue(int initialCapacity, Comparator cmp);

4) PriorityQueue pq3 = new PriorityQueue(Comparator cmp);

5) PriorityQueue pq4 = new PriorityQueue(Collection c);

Methods :-

-----  
add() / offer() :- Used to add an element in the Queue

poll() :- It is used to fetch the elements from top of the queue, after fetching it will delete the element.

peek() :- It is also used to fetch the elements from top of the queue, Unlike poll it will only fetch but not delete the element.

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<String> pq = new PriorityQueue<>();  
        pq.add("Orange");  
        pq.add("Apple");  
        pq.add("Mango");  
        pq.add("Guava");  
        pq.add("Grapes");  
  
        System.out.println(pq);  
    }  
}

-----  
import java.util.PriorityQueue;

public class PriorityQueueDemo3

{  
    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 PriorityQueueDemo1

{

```

public static void main(String[] argv) //7 3 5 6
{
    PriorityQueue<String> pq = new PriorityQueue<>();
    pq.add("9");
    pq.add("8");
        pq.add("7");
    System.out.print(pq.peek() + " "); // 6 7 8 9
    pq.offer("6");
        pq.offer("5");
    pq.add("3");

    pq.remove("1");
    System.out.print(pq.poll() + " ");
    if (pq.remove("2"))
        System.out.print(pq.poll() + " ");
    System.out.println(pq.poll() + " " + pq.peek());
        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"); // 6 9
        System.out.print(pq.peek() + " "); //2 2 3 4 4
        pq.offer("1");
            pq.offer("9");
        pq.add("3");

        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());
    }
}

```

---

Generics :

-----

Why generic came into picture :

-----

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 type casting as shown below.

```

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());
        }
    }
}

```

---

Even after type casting there is no guarantee that the things which are coming from ArrayList Object is Integer only because we can add anything in the Collection as a result java.lang.ClassCastException as shown in the program below.

```

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);
    }
}

```

---

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 out.

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 :-

---

- a) Type safe Object (No compilation warning)
  - b) Strict compile time checking (Type erasure)
  - c) No need of type casting
- 

```

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]

```

import java.util.*;
public class Test4
{
    public static void main(String [] args)
    {
        Dog d1 = new Dog();
    }
}

```

```

        Dog d2 = d1.getDogList().get(0);
        System.out.println(d2);
    }
}

class Dog
{
    public List<Dog> getDogList()
    {
        ArrayList<Dog> d = new ArrayList<>();
        d.add(new Dog());
        d.add(new Dog());
        d.add(new Dog());
        return d;
    }
}

```

Note :- In the above program the compiler will stop us from returning anything which is not compatible 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 A COMPILE TIME PROTECTION.

---

02-07-2024

---

//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 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()};

        Test8 t = new Test8();

        t.checkAnimals(dogs);
        t.checkAnimals(cats);
        t.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());

        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.

Eg:-

```

Parent [] arr = new Child[5]; //valid
Object [] arr = new String[5]; //valid

```

But in generics the same type is not valid

```

List<Object> list = new ArrayList<Integer>(); //Invalid
List<Parent> mylist = new ArrayList<Child>(); //Invalid

-----
import java.util.*;
public class Test10
{
    public static void main(String [] args)
    {
        /*ArrayList<Number> al = new ArrayList<Integer>(); [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 :- It 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 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 checking)

---

Wild card character(?) :

---

<?>	-: Many possibilities	
<Animal>	-: Only <Animal> can assign, but not Dog	or sub type of animal
<? super Dog>	-: Dog, Animal, Object can assign (Compiler has surity)	
<? extends Animal> -: Below of Animal(Child of Animal) means, sub classes of Animal (But the compiler does not have surity because you can have many sub classes of Animal in the future, so chances of wrong collections)		

---

```

import java.util.*;
class Parent
{
}
class Child extends Parent
{
}

public class Test11
{
    public static void main(String [] args)
    {

```

```
        ArrayList<?> lp = new ArrayList<Child>();
        ArrayList<Parent> lp1 = new ArrayList<Parent>();
        ArrayList<Child> lp2 = new ArrayList<>();
        System.out.println("Success");
    }
}

-----
```

```
//program on wild-card character
import java.util.*;
class Parent
{
}

class Child extends Parent
{
}
public class Test12
{
    public static void main(String [] args)
    {
        List<?> lp = new ArrayList<Parent>();
        System.out.println("Wild card....");
    }
}
```

```
-----
```

```
import java.util.*;
public class Test13
{
    public static void main(String[] args)
    {
        List<? extends Number> list1 = new ArrayList<Byte>();

        List<? super String> list2 = new ArrayList<Object>();

        List<? super Beta> list3 = new ArrayList<Alpha>();

        List list4 = new ArrayList();

        System.out.println("yes");
    }
}
```

```
-----
```

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

```

class Test<R,A>
{
    private R r;
    public void set(A a)
    {
        r = a;
    }

    public R get()
    {
        return r;
    }
}
public class Test14
{
    public static void main(String[] args)
    {
        Test<String,String> test = new Test();
        test.set("Info");
        System.out.println(test.get());
    }
}

-----
class MyClass<T>
{
    T obj;
    public MyClass(T obj)      //Student obj = new Student(
    {
        this.obj=obj;
    }

    T getObj()
    {
        return obj;
    }
}
public class Test15
{
    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)
{
}

-----//E stands for Element type
class Fruit
{
}
class Apple extends Fruit //Fruit is super, Apple is sub class
{
    @Override
    public String toString()
    {
        return "Apple";
    }
}

class Basket<E> // E is Fruit type
{
    private E element;
    public void setElement(E element) //Fruit element = new Apple();
    {
        this.element = element;
    }

    public E getElement() // public Fruit getElement{}
    {
        return this.element;
    }
}

public class Test16
{
    public static void main(String[] args)
    {
        Basket<Fruit> b = new Basket<>();
        b.setElement(new Apple());

        Apple x = (Apple) b.getElement();
        System.out.println(x);

        Basket<Fruit> b1 = new Basket<>();
        b1.setElement(new Mango());
        Mango y = (Mango)b1.getElement();
        System.out.println(y);
    }
}

```

```
    }
}

class Mango extends Fruit
{
    @Override
    public String toString()
    {
        return "Mango";
    }
}
```

---

## Concurrent collections in java

---

Concurrent Collections are introduced from JDK 1.5 onwards to enhance the performance of multithreaded application.

These are threadsafe collection and available in `java.util.concurrent` sub package.

### Limitation of Traditional Collection :

---

1) In the Collection framework most of the Collection classes are not thread-safe because those are non-synchronized like `ArrayList`, `LinkedList`, `HashSet`, `HashMap` is non-synchronized in nature, So If multiple threads will perform any operation on the collection object simultaneously then we will get some wrong data this is known as Data race or Race condition.

2) Some Collection classes are synchronized like `Vector`, `Hashtable` but performance wise these classes are slow in nature.

`Collections` class has provided static methods to make our `List`, `Set` and `Map` interface classes as a synchronized.

- a) `public static List synchronizedList(List list)`
- b) `public static Set synchronizedSet(Set set)`
- c) `public static Map synchronizedMap(Map map)`

3) Traditional Collection works with fail fast iterator that means while iterating the element, if there is a change in structure then we will get `ConcurrentModificationException`,

On the other hand concurrent collection works with fail safe iterator where even though there is a change in structure but we will not get `ConcurrentModificationException`.

---

```
import java.util.*;
public class Collection1
{
    public static void main(String args[])
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        System.out.println("ArrayList Elements : "+al);
        Set s = new HashSet(al);
```

```

        System.out.println("Set Elements are: "+s);
    }
}

//Collections.synchronizedList(List list);
import java.util.*;
public class Collection2
{
    public static void main(String[] args)
    {
        ArrayList<String> arl = new ArrayList<>();
        arl.add("Apple");
        arl.add("Orange");
        arl.add("Grapes");
        arl.add("Mango");
        arl.add("Guava");
        arl.add("Mango");

        List<String> syncCollection = Collections.synchronizedList(arl);

        List<String> upperList = new ArrayList<>(); //New List

        Runnable listOperations = () ->
        {
            synchronized (syncCollection)
            {
                syncCollection.forEach(str -> upperList.add(str.toUpperCase()));
            }
        };
        Thread t1 = new Thread(listOperations);
        t1.start();

        upperList.forEach(x -> System.out.println(x));
    }
}

//Collections.synchronizedSet(Set set);
import java.util.*;
public class Collection3
{
    public static void main(String[] args)
    {
        Set<String> set = Collections.synchronizedSet(new HashSet<>());
        set.add("Apple");
        set.add("Orange");
        set.add("Grapes");
        set.add("Mango");
        set.add("Guava");
        set.add("Mango");

        System.out.println("Set after Synchronization :");
        synchronized (set)
        {
            Spliterator<String> itr = setspliterator();
            itr.forEachRemaining(str -> System.out.println(str));
        }
    }
}

```

```

        }
    }

//Collections.synchronizedMap(Map map);
import java.util.*;
public class Collection4
{
    public static void main(String[] args)
    {
        Map<String, String> map = new HashMap<String, String>();
        map.put("1", "Ravi");
        map.put("4", "Elina");
        map.put("3", "Aryan");
        Map<String, String> synmap = Collections.synchronizedMap(map);
        System.out.println("Synchronized map is :" + synmap);
    }
}

package com.ravi.concurrent;

import java.util.ArrayList;
import java.util.Iterator;

class ConcurrentModification extends Thread
{
    ArrayList<String> al = null;

    public ConcurrentModification(ArrayList<String> al)
    {
        this.al= al;
    }

    @Override
    public void run()
    {
        try
        {
            Thread.sleep(2000);
        }
        catch(InterruptedException e)
        {

        }

        al.add("KIWI");
    }
}

public class Collection5 {

    public static void main(String[] args) throws InterruptedException
    {
        ArrayList<String> arl = new ArrayList<>();

```

```

        arl.add("Apple");
        arl.add("Orange");
        arl.add("Grapes");
        arl.add("Mango");
        arl.add("Guava");

        ConcurrentModification cm = new ConcurrentModification(arl);
        cm.start();

        Iterator<String> itr = arl.iterator();

        while(itr.hasNext())
        {
            System.out.println(itr.next());
            Thread.sleep(500);
        }

    }
}

```

Note :- In the above program we will get `java.util.ConcurrentModificationException` because Iterator is fail fast iterator.

---

03-07-2024

---

`CopyOnWriteArrayList` in java :

---

`public class CopyOnWriteArrayList implements List, Cloneable, Serializable, RandomAccess`

A `CopyOnWriteArrayList` is similar to an `ArrayList` but it has some additional features like thread-safe. This class is existing in `java.util.concurrent` sub package.

`ArrayList` is not thread-safe. We can't use `ArrayList` in the multi-threaded environment because it creates a problem in `ArrayList` values (Data inconsistency).

\*The `CopyOnWriteArrayList` is an enhanced version of `ArrayList`. If we are making any modifications(add, remove, etc.) in `CopyOnWriteArrayList` then JVM creates a new copy by use of Cloning.

The `CopyOnWriteArrayList` is costly, if we want to perform update operations, because whenever we make any changes the JVM creates a cloned copy of the array and add/update element to it.

It is a thread-safe version of `ArrayList` as well as here Iterator is fail safe iterator. Multiple threads can read the data but only one thread can write the data at one time.

\*`CopyOnWriteArrayList` is the best choice if we want to perform read operation frequently in multithreaded environment.

The `CopyOnWriteArrayList` is a replacement of a synchronized List, because it offers better concurrency.

Constructors of `CopyOnWriteArrayList` in java :

---

We have 3 constructors :

1) CopyOnWriteArrayList c = new CopyOnWriteArrayList();  
It creates an empty list in memory. This constructor is useful when we want to create a list without any value.

2) CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection c);  
Interconversion of collections.

3) CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] obj) ;  
It Creates a list that containing all the elements that is specified Array. This constructor is useful when we want to create a CopyOnWriteArrayList from Array.

Note : All the immutable objects are thread-safe because, On immutable objects if we perform any operation then another object will be created in a new memory location so at a time multiple threads can work.

All String Object, Wrapper classes object, Concurrent collection classes are Thread-safe

---

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListExample1
{
    public static void main(String[] args)
    {
        List<String> list = Arrays.asList("Apple", "Orange", "Mango", "Kiwi", "Grapes");

        CopyOnWriteArrayList<String> copyOnWriteList = new CopyOnWriteArrayList<String>(list);

        System.out.println("Without modification = "+copyOnWriteList);

        //Iterator1
        Iterator<String> iterator1 = copyOnWriteList.iterator();

        //Add one element and verify list is updated
        copyOnWriteList.add("Guava");

        System.out.println("After modification = "+copyOnWriteList);

        //Iterator2
        Iterator<String> iterator2 = copyOnWriteList.iterator();

        System.out.println("Element from first Iterator:");
        iterator1.forEachRemaining(System.out::println);

        System.out.println("Element from Second Iterator:");
        iterator2.forEachRemaining(System.out::println);
    }
}

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

```

class ConcurrentModification extends Thread
{
    CopyOnWriteArrayList<String> al = null;
    public ConcurrentModification(CopyOnWriteArrayList<String> al)
    {
        this.al = al;
    }
    @Override
    public void run()
    {
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
        }
        al.add("KIWI");
    }
}
public class CopyOnWriteArrayListExample2
{
    public static void main(String[] args) throws InterruptedException
    {
        CopyOnWriteArrayList<String> arl = new CopyOnWriteArrayList<>();
        arl.add("Apple");
        arl.add("Orange");
        arl.add("Grapes");
        arl.add("Mango");
        arl.add("Guava");
        ConcurrentModification cm = new ConcurrentModification(arl);
        cm.start();

        Iterator<String> itr = arl.iterator();
        while(itr.hasNext())
        {
            String str = itr.next();
            System.out.println(str);
            Thread.sleep(1500);
        }

        System.out.println(".....");
        Spliterator<String> spl = arl.spliterator();
        spl.forEachRemaining(x -> System.out.println(x));
    }
}

```

---

CopyOnWriteArrayList :

---

public class CopyOnWriteArrayList extends AbstractSet implements Serializable

A CopyOnWriteArrayList is a thread-safe version of HashSet in Java and it works like CopyOnWriteArrayList in java.

The CopyOnWriteArrayList internally used CopyOnWriteArrayList to perform all type of operation. It means the CopyOnWriteArrayList internally creates an object of CopyOnWriteArrayList and perform operation on it.

Whenever we perform add, set, and remove operation on CopyOnWriteArrayList, it internally creates a new object of CopyOnWriteArrayList and copies all the data to the new object. So, when it is used in by multiple threads, it doesn't create a problem, but it is well suited if we have small size collection and want to perform only read operation by multiple threads.

The CopyOnWriteArrayList is the replacement of synchronizedSet and offers better concurrency.

It creates a new copy of the array every time iterator is created, so performance is slower than HashSet.

Constructors :

---

It has two constructors

1) CopyOnWriteArrayList set1 = new CopyOnWriteArrayList();  
It will create an empty Set

2) CopyOnWriteArrayList set1 = new CopyOnWriteArrayList(Collection c); Interconversion of collection

---

```
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListExample1
{
    public static void main(String[] args)
    {
        CopyOnWriteArrayList<String> set = new CopyOnWriteArrayList<>();

        set.add("Java");
        set.add("Python");
        set.add("C++");
        set.add("Java");

        Iterator itr = set.iterator();
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }

        // Adding a new element
        set.add("JavaScript");
        System.out.println(".....");
        for (String language : set)
        {
            System.out.println(language);
        }
    }
}
```

```
import java.util.concurrent.CopyOnWriteArraySet;

public class CopyOnWriteArraySetExample2
{
    public static void main(String[] args)
    {
        CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<Integer>();
        set.add(1);
        set.add(2);
        set.add(3);
        set.add(4);
        set.add(5);

        System.out.println("Is element contains: "+set.contains(1));

        System.out.println("Is set empty: "+set.isEmpty());

        System.out.println("remove element from set: "+set.remove(3));

        System.out.println("Element from Set: "+ set);
    }
}
```

\*\*\* ConcurrentHashMap :

```
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements
ConcurrentMap<K,V>, Serializable
```

Like HashMap, ConcurrentHashMap provides similar functionality except that it has internally maintained concurrency.

It is the concurrent version of the HashMap. It internally maintains a Hashtable that is divided into segments(Buckets).

The number of segments depends upon the level of concurrency required the ConcurrentHashMap. By default, it divides into 16 segments and each Segment behaves independently. It doesn't lock the whole HashMap as done in Hashtables/synchronizedMap, it only locks the particular segment(Bucket) of HashMap. [Bucket level locking]

ConcurrentHashMap allows multiple threads can perform read operation without locking the ConcurrentHashMap object.

It does not allow null as a key or even null as a value.

[Note :- TreeSet, TreeMap, Hashtable, PriroityQueue, ConcurrentHashMap , These 5 classes never containing null key or null element)

It contains 5 types of constructor :

- 1) ConcurrentHashMap chm1 = new ConcurrentHashMap();
- 2) ConcurrentHashMap chm2 = new ConcurrentHashMap(int initialCapacity);
- 3) ConcurrentHashMap chm3 = new ConcurrentHashMap(int initialCapacity, float loadFactor);

- 4) ConcurrentHashMap chm4 = new ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);
- 5) ConcurrentHashMap chm5 = new ConcurrentHashMap(ConcurrentMap m);

Internal Working of ConcurrentHashMap :

Like HashMap and Hashtable, the ConcurrentHashMap is also used Hashtable data structure. But it is using the segment locking strategy to handle the multiple threads.

A segment(bucket) is a portion of ConcurrentHashMap and ConcurrentHashMap uses a separate lock for each thread. Unlike Hashtable or synchronized HashMap, it doesn't synchronize the whole HashMap or Hashtable for one thread.

As we have seen in the internal implementation of the HashMap, the default size of HashMap is 16 and it means there are 16 buckets. The ConcurrentHashMap uses the same concept is used in ConcurrentHashMap. It uses the 16 separate locks for 16 buckets by default because the default concurrency level is 16. It means a ConcurrentHashMap can be used by 16 threads at same time. If one thread is reading from one bucket(Segment), then the second bucket doesn't affect it.

Why we need ConcurrentHashMap in java?

As we know Hashtable and HashMap works based on key-value pairs. But why we are introducing another Map? As we know HashMap is not thread safe, but we can make it thread-safe by using Collections.synchronizedMap() method and Hashtable is thread-safe by default.

But a synchronized HashMap or Hashtable is accessible only by one thread at a time because the object get the lock for the whole HashMap or Hashtable. Even multiple threads can't perform read operations at the same time. It is the main disadvantage of Synchronized HashMap or Hashtable, which creates performance issues. So ConcurrentHashMap provides better performance than Synchronized HashMap or Hashtable.

```
//HashMap for ConcurrentHashMap
import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample1
{
    public static void main(String args[])
    {

        HashMap<Integer, String> hashMap = new HashMap<Integer, String>();
        hashMap.put(1, "Ravi");
        hashMap.put(2, "Ankit");
        hashMap.put(3, "Prashant");
        hashMap.put(4, "Pallavi");

        ConcurrentHashMap<Integer, String> concurrentHashMap = new
        ConcurrentHashMap<>(hashMap);
        System.out.println("Object from ConcurrentHashMap: "+ concurrentHashMap);

    }
}
```

```
import java.util.Iterator;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample2
{
    public static void main(String args[])
    {
        // Creating ConcurrentHashMap
        Map<String, String> cityTemperatureMap = new ConcurrentHashMap<>();

        cityTemperatureMap.put("Delhi", "30");
        cityTemperatureMap.put("Mumbai", "32");
        cityTemperatureMap.put("Chennai", "35");
        cityTemperatureMap.put("Bangalore", "22");

        Iterator<String> iterator = cityTemperatureMap.keySet().iterator();

        while (iterator.hasNext())
        {
            System.out.println(cityTemperatureMap.get(iterator.next()));
            // adding new value, it won't throw error
            cityTemperatureMap.put("Hyderabad", "28");
        }
    }
}
```

we can easily add an entry in the concurrenthashmap, even at the time of fetching the data using iterator, It is working because internally it uses bucket level locking.

04-07-2024

-----  
Streams in java :

-----  
It is introduced from Java 8 onwards, the Stream API is used to process the collection objects.

It contains classes for processing sequence of elements over Collection object and array.

Stream is a predefined interface available in `java.util.stream` sub package.

Package Information :

-----  
`java.util` -> Base package  
`java.util.function` -> Functional interfaces  
`java.util.concurrent` -> Multithreaded support  
`java.util.stream` -> Processing of Collection Object

forEach() method in java :

-----  
The Java `forEach()` method is a technique to iterate over a collection such as (list, set or map) and stream. It is used to perform a given action on each of the element of the collection.

The `forEach()` method has been added in following places:

Iterable interface – This makes Iterable.forEach() method available to all collection classes.  
Iterable interface is the super interface of Collection interface

Map interface – This makes forEach() operation available to all map classes.

Stream interface – This makes forEach() operations available to all types of stream.

Creation of Streams to process the data :

We can create Stream from collection or array with the help of stream() and of() methods:

A stream() method is added to the Collection interface and allows creating a Stream<T> using any collection object as a source

```
public java.util.Stream<E> stream();
```

The return type of this method is Stream interafce available in java.util.stream sub package.

Eg:-

```
List<String> items = new ArrayList<String>();
    items.add("Apple");
    items.add("Orange");
    items.add("Mango");
    Stream<String> stream = items.stream();
```

```
package com.ravi.basic;
import java.util.*; //Base package
import java.util.stream.*; //Sub package
public class StreamDemo1
{
    public static void main(String[] args)
    {
        List<String> items = new ArrayList<>();
        items.add("Apple");
        items.add("Orange");
        items.add("Mango");

        //Collections Object to Stream
        Stream<String> strm = items.stream();
        strm.forEach(p -> System.out.println(p));
    }
}
```

Stream.of()

```
public static java.util.stream.Stream<T> of(T ...x)
```

It is a static method of Stream interface through which we can create Stream of arrays and Stream of Collection. The return type of this method is Stream interface

```
//Stream.of()
package com.ravi.basic;
import java.util.stream.*;
```

```

public class StreamDemo2
{
    public static void main(String[] args)
    {
        //Stream of Collection
        Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
        stream.forEach(p -> System.out.println(p));

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

        //Anonymous Array Object (Stream of Arrays)
        Stream<Integer> strm = Stream.of( new Integer[]{15,29,45,8,16} );
        strm.forEach(p -> System.out.println(p));
    }
}

```

---

**Operation on Stream interface :**

---

We have 2 types of operation :

- 1) Intermediate Operation
- 2) Terminal Operation

**Intermediate Operations:**

---

The following methods are available to perform intermediate operation.

**filter(Predicate<T> predicate):** Returns a new stream which contains filtered elements based on the boolean expression using Predicate.

**map(Function<T, R> mapper):** Transforms elements in the stream using the provided mapping function.

**flatMap(Function<T, Stream<R>> mapper):** Flattens a stream of streams into a single stream.

**distinct():** Returns a stream with distinct elements (based on their equals method).

**sorted():** Returns a stream with elements sorted in their natural order.

**sorted(Comparator<T> comparator):** Returns a stream with elements sorted using the specified comparator.

**peek(Consumer<T> action):** Allows us to perform an action on each element in the stream without modifying the stream.

**limit(long maxSize):** Limits the number of elements in the stream to a specified maximum size.

**skip(long n):** Skips the first n elements in the stream.

**takeWhile(Predicate<T> predicate):** Returns a stream of elements from the beginning until the first element that does not satisfy the predicate.

**dropWhile(Predicate<T> predicate):** Returns a stream of elements after skipping elements at the beginning that satisfy the predicate.

---



---

```
public abstract Stream<T> filter(Predicate<T> p) :
```

It is a predefined method of Stream interface. It is used to select/filter elements as per the Predicate passed as an argument. It is basically used to filter the elements based on boolean condition.

```
public abstract <T> collect(java.util.stream.Collectors c)
```

It is a predefined method of Stream interface. It is used to return the result of the intermediate operations performed on the stream.

It is a terminal operation. It is used to collect the data after filtration and convert the data to the Collection(List/Set).

Collectors is a predefined final class available in java.util.stream sub package which contains static method toList() and toSet() to convert the data as a List/Set i.e Collection object. The return type of this method is List/Set interface.

```
//Filter all the even numbers from Collection  
package com.ravi.basic;
```

```
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.Set;  
import java.util.stream.Collectors;
```

```
public class StreamDemo3  
{  
    public static void main(String[] args)  
    {  
        //With Traditional Collection approach  
        List<Integer> listOfNumbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
        ArrayList<Integer> listOfEven = new ArrayList<>();  
  
        for(Integer i : listOfNumbers)  
        {  
            if(i%2==0)  
            {  
                listOfEven.add(i);  
            }  
        }  
        System.out.println("All Even numbers using Collection");  
        listOfEven.forEach(System.out::println);  
  
        System.out.println("All the even numbers using Stream API");  
  
        listOfNumbers.stream().filter(num -> num%2==0).forEach(System.out::println);  
  
        System.out.println(".....");  
  
        listOfNumbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,3,5);  
        //Fetch all the unique odd numbers from the given list
```

```

        Set<Integer> collect = listOfNumbers.stream().filter(num ->
num%2==1).collect(Collectors.toSet());
        collect.forEach(System.out::println);

    }

-----



//Filtering the name
package com.ravi.basic;
import java.util.*;
import java.util.stream.*;
public class StreamDemo4
{
    public static void main(String[] args)
    {
        List<String> listOfNames = Arrays.asList("Ravi", "Rahul", "Akshar",
"Roshan","Raj","Ankit","Ravi");

        listOfNames.stream().filter(str -> str.startsWith("R")).sorted((s1,s2)->s2.compareTo(s1)).distinct().forEach(System.out::println);
    }
}

-----



//Sorting the data
package com.ravi.basic;
import java.util.*;
import java.util.stream.*;
public class StreamDemo5
{
    public static void main(String[] args)
    {
        List<String> names = Arrays.asList("Zaheer","Rahul","Aryan","Sailesh","Zaheer");

        List<String> sortedName =
names.stream().sorted((s1,s2)->s1.compareTo(s2)).collect(Collectors.toList());

        System.out.println(sortedName);
    }
}

-----



package com.ravi.basic;

import java.util.ArrayList;

//Fetch all the Employees name whose salary is greater then 50k

public class StreamDemo6
{
    public static void main(String[] args)

```

```
{  
    ArrayList<Employee> listOfEmployees = new ArrayList<>();  
    listOfEmployees.add(new Employee(1, "Scott", 57000.89));  
    listOfEmployees.add(new Employee(2, "Smith", 51000.89));  
    listOfEmployees.add(new Employee(3, "Alex", 97000.89));  
    listOfEmployees.add(new Employee(4, "John", 41000.89));  
  
    listOfEmployees.stream().filter(emp -> emp.empSalary()>=50000).forEach(e1 ->  
        System.out.println(e1.empName()));  
}  
}
```

```
record Employee(Integer empld, String empName, Double empSalary)  
{
```

```
}
```

---

05-07-2024

---

public Stream map(Function<T,R> mapper) :

---

It is a predefined method of Stream interface.

It takes Function (Predefined functional interface ) as a parameter.

It performs intermediate operation and consumes single element from input Stream and produces single element to output Stream. (1:1 transformation)

Here mapper function is functional interface which takes one input and provides one output.

---

```
package com.ravi.map_demo;  
  
import java.util.List;  
import java.util.stream.Collectors;  
import java.util.stream.Stream;  
  
public class MapMethod {  
  
    public static void main(String[] args)  
    {  
        //add constant value 10 to each odd number and convert into  
        //list  
  
        List<Integer> collect = Stream.of(1,2,3,4,5,6,7,8,9,10).filter(n -> n %2==1).map(num  
-> num + 10).collect(Collectors.toList());  
  
        collect.forEach(System.out::println);  
    }  
}  
  
-----  
  
package com.ravi.basic;
```

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo7
{
    public static void main(String[] args)
    {
        //add value 10 to each and every number
        List<Integer> asList = Arrays.asList(1,2,3,4,5,6,7,8,9);
        asList.stream().map(num -> num + 10).forEach(System.out::println);

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

        //Find even numbers in stream and collect the cubes
        List<Integer> numbers = List.of(1,2,3,4,5,6,7,8,9,10,11,12);
        List<Integer> collect = numbers.stream().filter(n -> n % 2 == 0).map(num ->
        num*num*num).collect(Collectors.toList());
        collect.forEach(System.out::println);

        System.out.println(".....");
        //Find the length of the name
        Stream.of("Raj","Scott","subramanyam","Rahul").map(str->
        str.length()).forEach(System.out::println);
    }
}

-----//Program on map(Function<T,R> mapped)
package com.ravi.basic;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamDemo8
{
    public static void main(String args[])
    {
        List<Player> listOfPlayers = createPlayerList();
        //Player object to name of the player
        Set<String> collect = listOfPlayers.stream().map(obj -> obj.name()).collect(Collectors.toSet());
        collect.forEach(System.out::println);
    }

    public static List<Player> createPlayerList()
    {
        ArrayList<Player> al = new ArrayList<>();
        al.add(new Player(18, "Virat"));
        al.add(new Player(45, "Rohit"));
        al.add(new Player(7, "Dhoni"));
        al.add(new Player(18, "Virat"));
    }
}

```

```
        al.add(new Player(90, "Bumrah"));
        al.add(new Player(67, "Hrdik"));

        return al;
    }
}
```

```
record Player(Integer playerId, String name)
{
}
```

---

```
public Stream flatMap(Function<? super T,>? extends Stream<? extends R>> mapper)
```

It is a predefined method of Stream interface.

The map() method produces one output value for each input value in the stream. So if there are n elements in the stream, map() operation will produce a stream of n output elements.

flatMap() is two step process i.e. map() + Flattening. It helps in converting Collection<Collection<T>> to Collection<T> [to make flat i.e converting Collections of collection into single collection or merging of all the collection]

---

```
//flatMap()
//map + Flattening [Converting Collections of collection into single collection]
package com.ravi.basic;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamDemo9
{
    public static void main(String[] args)
    {
        List<String> list1 = Arrays.asList("A","B","C");
        List<String> list2 = Arrays.asList("D","E","F");
        List<String> list3 = Arrays.asList("G","H","I");

        List<List<String>> listOfList = Arrays.asList(list1, list2, list3);
        System.out.println(listOfList);
        System.out.println(".....");
        List<String> collect = listOfList.stream().flatMap(list ->
list.stream()).collect(Collectors.toList());
        System.out.println(collect);
    }
}
```

---

```
//Flattening of prime, even and odd number
package com.ravi.basic.flat_map;
```

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
```

```
import java.util.stream.Stream;

public class FlatMapDemo1
{
    public static void main(String[] args)
    {
        List<Integer> primeNumbers = Arrays.asList(5,7,11);
        List<Integer> evenNumbers = Arrays.asList(2,4,6);
        List<Integer> oddNumbers = Arrays.asList(1,3,5);

        List<List<Integer>> listOfCollection = Arrays.asList(primeNumbers, evenNumbers,
oddNumbers);
        System.out.println(listOfCollection);

        List<Integer> collect = listOfCollection.stream().flatMap(num ->
num.stream()).collect(Collectors.toList());

        System.out.println(collect);

    }
}
```

---

```
//Fetching first character using flatMap()
package com.ravi.basic.flat_map;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class FlatMapDemo2
{
    public static void main(String[] args)
    {
        List<String> listOfNames = Arrays.asList("Jyoti","Aryan","Virat","Aman");

        List<Character> collect = listOfNames.stream().flatMap(str ->
Stream.of(str.charAt(0))).collect(Collectors.toList());
        System.out.println(collect);

    }
}
```

---

```
package com.ravi.basic.flat_map;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

class Product
{
```

```

private Integer productId;
private List<String> listOfProducts;

public Product(Integer productId, List<String> listOfProducts)
{
    super();
    this.productId = productId;
    this.listOfProducts = listOfProducts;
}

public Integer getProductId() {
    return productId;
}

public List<String> getListOfProducts() {
    return listOfProducts;
}
}

public class FlatMapDemo3
{
    public static void main(String[] args)
    {

        List<Product> listOfProduct = Arrays.asList(
            new Product(1, Arrays.asList("Camera", "Mobile", "Laptop")),
            new Product(2, Arrays.asList("Bat", "Ball", "Wicket")),
            new Product(3, Arrays.asList("Chair", "Table", "Lamp")),
            new Product(4, Arrays.asList("Cycle", "Bike", "Car"))
        );

        List<String> collect = listOfProduct.stream().flatMap(p ->
            p.getListOfProducts().stream()
                .collect(Collectors.toList()));
        System.out.println(collect);
    }
}

```

---

\*\*Difference between map() and flatMap()

---

map() method transforms each element into another single element.

flatMap() transforms each element into a stream of elements and then flattens those streams into a single stream.

We should use map() when you want a one-to-one transformation, and we should use flatMap() when dealing with nested structures or when you need to produce multiple output elements for each input element.

---

public Stream sorted() :

-----  
It is a predefined method of Stream interface.

It provides default natural sorting order.

The return type of this method is Stream.

public Stream distinct() :

-----  
It is a predefined method of Stream interface.

If we want to return stream from another stream by removing all the duplicates then we should use distinct() method.

```
-----  
package com.ravi.basic;  
import java.util.stream.Stream;  
public class StreamDemo10  
{  
    public static void main(String[] args)  
    {  
        Stream<String> s = Stream.of("Virat", "Rohit", "Dhoni", "Virat",  
"Rohit", "Aswin", "Bumrah");  
        s.distinct().sorted((s1,s2)-> s2.compareTo(s1)).forEach(System.out::println);  
    }  
}
```

-----  
public Stream distinct() :

-----  
It is a predefined method of Stream interface.

If we want to return stream from another stream by removing all the duplicates then we should use distinct() method.

public Stream<T> limit(long maxSize) :

-----  
It is a predefined method of Stream interface to work with sequence of elements.

The limit() method is used to limit the number of elements in a stream by providing maximum size.

It creates a new Stream by taking the data from original Stream.

Elements which are not in the range or beyond the range of specified limit will be ignored.

```
-----  
package com.ravi.basic;  
import java.util.stream.Stream;  
public class StreamDemo11  
{  
    public static void main(String[] args)  
    {  
        Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
  
        Stream<Integer> limitedStream = numbers.limit(6);  
  
        limitedStream.forEach(System.out::println);  
    }  
}
```

```
}
```

---

```
public Stream<T> skip(long n) :
```

---

It is a predefined method of Stream interface which is used to skip the elements from beginning of the Stream.

It returns a new stream that contains the remaining elements after skipping the specified number of elements which is passed as a parameter.

```
package com.ravi.basic;
import java.util.stream.Stream;
public class StreamDemo12
{
    public static void main(String[] args)
    {
        Stream<String> s = Stream.of("Virat", "Rohit", "Dhoni", "Zaheer",
"Raina", "Sahwag", "Sachin", "Bumrah");
        s.skip(3).limit(5).forEach(System.out::println);
    }
}
```

---

```
public Stream<T> peek(Consumer<? super T> action) :
```

---

It is a predefined method of Stream interface which is used to perform a side-effect operation on each element in the stream while the stream remains unchanged.

It is an intermediate operation that allows us to perform operation on each element of Stream without modifying original.

The peek() method takes a Consumer as an argument, and this function is applied to each element in the stream. The method returns a new stream with the same elements as the original stream.

```
package com.ravi.basic;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo13
{
    public static void main(String[] args)
    {
        Stream<String> listOfFruits =
Stream.of("Apple", "Mango", "Grapes", "Kiwi", "Pomegranate");

        List<Integer> doubledNumbers = listOfFruits
            .peek(str -> System.out.println("Peeking from Original: " + str.toUpperCase()))
            .map(fruit -> fruit.length())
            .collect(Collectors.toList());
        System.out.println("-----");
        System.out.println(doubledNumbers);

    }
}
```

```
}
```

---

```
public Stream takeWhile(Predicate<t> p) :
```

---

```
public Stream<T> takeWhile(Predicate<T> predicate) :
```

It is a predefined method of Stream interface introduced from java 9 which is used to perform a side-effect operation on each element in the stream while the stream remains unchanged.

\*It is used to create a new stream that includes elements from the original stream only as long as they satisfy a given predicate.

```
package com.ravi.basic;
```

```
import java.util.stream.Stream;
```

```
public class StreamDemo14
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
    Stream<Integer> numbers = Stream.of(10,11,9,13,2,1,100);
```

```
    numbers.takeWhile(n -> n > 9).forEach(System.out::println);
```

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

```
    numbers = Stream.of(12,2,3,4,5,6,7,8,9);
```

```
    numbers.takeWhile(n -> n%2==0).forEach(System.out::println);
```

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

```
    numbers = Stream.of(1,2,3,4,5,6,7,8,9);
```

```
    numbers.takeWhile(n -> n < 9).forEach(System.out::println);
```

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

```
    numbers = Stream.of(11,2,3,4,5,6,7,8,9);
```

```
    numbers.takeWhile(n -> n > 9).forEach(System.out::println);
```

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

```
    Stream<String> stream = Stream.of("Ravi", "Ankit", "Rohan", "Aman", "Ravish");
```

```
    stream.takeWhile(str -> str.charAt(0)=='R').forEach(System.out::println);
```

```
}
```

---

```
public Stream<T> dropWhile(Predicate<T> predicate) :
```

---

It is a predefined method of Stream interface introduced from java 9 which is used to create a new stream by excluding elements from the original stream as long as they satisfy a given predicate.

```
package com.ravi.basic;

import java.util.stream.Stream;

public class StreamDemo15 {

    public static void main(String[] args)
    {
        Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        numbers.dropWhile(num -> num < 5).forEach(System.out::println);

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

        numbers = Stream.of(15, 8, 7, 9, 5, 6, 7, 8, 9, 10);

        numbers.dropWhile(num -> num > 5).forEach(System.out::println);

    }
}
```

---

Method Reference in java :

Method Reference :

It is a new feature introduced from java 1.8 onwards.

It is mainly used to write concise coding.

By using method reference we can refer an existing method which is available at API level or Project level.

We can use this technique in the body of Lambda expression just to call method definition.

The entire method body will be automatically placed into Lambda Expression.

It is used to enhance the code reusability.

It uses :: (Double Colon Operator)

While working with Lambda expression we need to write the Lambda Method Body but while working with Method reference we can refer an existing method which is already available in the package or Project.

There are 4 types of method reference

- 1) Static Method Reference(ClassName::methodName)
  - 2) Instance Method Reference(objectReference::methodName)
  - 3) Constructor Reference (ClassName::new)
  - 4) Arbitrary Referenec (ClassName::instanceMethodName)
- 

Basic Example of Method Reference :

---

3 files :

---

Worker.java(I)

---

```
package com.ravi.method_ref;
```

```
@FunctionalInterface  
public interface Worker  
{  
    void work();  
}
```

Employee.java(C)

---

```
package com.ravi.method_ref;
```

```
public class Employee  
{  
    public void work()  
    {  
        System.out.println("Employee is working");  
    }  
}
```

MethodReferenceDemo1.java

---

```
package com.ravi.method_ref;  
  
public class MethodReferenceDemo1  
{  
    public static void main(String[] args)  
    {  
        //Writing Lambda Expression  
        Worker w = ()-> System.out.println("Worker is working");  
        w.work();  
  
        //Writing Method Reference  
        Worker w1 = new Employee()::work;  
        w1.work();  
    }  
}
```

---

Method reference can be called by Functional Interface . Here Functional interface method argument and method return type must be same otherwise we will get compilation error as shown in the program

3 files :

---

Worker.java(I)

```
-----  
package com.ravi.method_ref;
```

```
@FunctionalInterface  
public interface Worker  
{  
    void work();  
}
```

Employee.java(C)

```
-----  
package com.ravi.method_ref;  
  
public class Employee  
{  
    public static void salary()  
    {  
        System.out.println("Employee is working for salary");  
    }  
}
```

MethodReferenceDemo2.java(C)

```
-----  
package com.ravi.method_ref;  
  
public class MethodReferenceDemo1  
{  
    public static void main(String[] args)  
    {  
        Worker w1 = Employee::salary;  
        w1.work();  
  
    }  
}
```

3 files :

-----  
Worker.java(I)

```
-----  
package com.ravi.method_ref;  
  
@FunctionalInterface  
public interface Worker  
{  
    void work(double salary);  
}
```

Employee.java(C)

```
-----  
package com.ravi.method_ref;  
  
public class Employee
```

```
{  
    public static void salary(double salary)  
    {  
        System.out.println("Employee Salary is :" + salary);  
    }  
}
```

MethodReferenceDemo3.java

```
-----  
package com.ravi.method_ref;  
  
public class MethodReferenceDemo3  
{  
    public static void main(String[] args)  
    {  
        Worker w1 = Employee::salary;  
        w1.work(40000);  
  
    }  
}
```

-----  
Program on Static Method Reference :

```
-----  
package com.ravi.static_method_reference;  
  
import java.util.Vector;  
  
class EvenOrOdd  
{  
    public static void isEven(int number)  
    {  
        if (number % 2 == 0)  
        {  
            System.out.println(number + " is even");  
        }  
        else  
        {  
            System.out.println(number + " is odd");  
        }  
    }  
}  
public class StaticMethodReferenceDemo1  
{  
    public static void main(String[] args)  
    {  
        Vector<Integer> numbers = new Vector<>();  
        numbers.add(5);  
        numbers.add(2);  
        numbers.add(9);  
        numbers.add(12);  
        System.out.println("By using Lambda....");  
        numbers.forEach(x -> EvenOrOdd.isEven(x));  
  
        System.out.println("_____");  
    }  
}
```

```
        System.out.println("By using static method reference");
        numbers.forEach(EvenOrOdd::isEven);
    }
}
```

---

Program that describes how to work with instance method :

---

```
package com.ravi.instance_method_reference;

interface Trainer
{
    void getTraining(String name, int experience);
}

class InstanceMethod
{
    public void getTraining(String name, int experience)
    {
        System.out.println("Trainer name is :" + name + " having " + experience + " years of
experience.");
    }
}

public class InstanceMethodReferenceDemo
{
    public static void main(String[] args)
    {
        //Using Lambda Expression
        Trainer t1 = (name, exp) -> System.out.println("Trainer name is :" + name + " and total
experience is :" + exp);
        t1.getTraining("Smith", 5);

        //By using Method reference
        Trainer t2 = new InstanceMethod()::getTraining;
        t2.getTraining("Scott", 10);
    }
}
```

---

By using constructor :

---

```
package com.ravi.constructor_reference;

@FunctionalInterface
interface A
{
    Test createObject();
}

class Test
{
    public Test()
    {
```

```

        System.out.println("Test class Constructor invoked");
    }
}
public class ConstructorReferenceDemo1
{
    public static void main(String[] args)
    {
        //By Using Lambda
        A obj = ()-> new Test();
        obj.createObject();

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

        //By Using Constructor Reference
        A obj1 = Test::new;
        obj1.createObject();
    }
}

```

---

```
package com.ravi.constructor_reference;
```

```
import java.util.function.Function;
```

```

class MyClass
{
    private int value;

    public MyClass(int value)
    {
        this.value = value;
    }

    public int getValue()
    {
        return this.value;
    }
}
```

```
public class ConstructorReferenceDemo2
```

```
{
    public static void main(String[] args)
    {
        Function<Integer,MyClass> consRef = MyClass::new;
        MyClass obj = consRef.apply(9);
        System.out.println(obj.getValue());
    }
}
```

---

```
//How to work with array object i.e creating array using constructor :
```

---

```
package com.ravi.constructor_reference;
```

```
import java.util.Scanner;
import java.util.function.Function;
```

```
class Person
{
    private String name;

    public Person(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}

public class ConstructorReferenceDemo3
{
    public static void main(String[] args)
    {
        Function<Integer,Person[]> consRef = Person[]::new;
        Person[] persons = consRef.apply(5); //5 is size of array

        Scanner sc = new Scanner(System.in);

        for(int i=0; i<persons.length; i++)
        {
            System.out.println("Enter the name at "+i+" position :");
            String name = sc.nextLine();
            persons[i] = new Person(name);
        }

        System.out.println("Getting the name of the person :");
        for(Person person : persons)
        {
            System.out.println(person.getName());
        }
    }
}
```

---