

20-MAY-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.

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

Here we don't have parameter so, the information is partial, if we pass parameter then we will get complete information.

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

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

```
int data; Field  
  
public void m1(int y)  
{  
    int x = 10;  
}
```

Flavors of Java :

We have total 4 types of flavors :

- 1) JSE (Java Standard Edition) [Core Java]
- 2) JEE (Java Enterprise Edition) [Advanced Java]
- 3) JME (Java Micro Edition) [Android Application]
- 4) JavaFX (It is used to design or develop GUI Applications) [Outdated]

GUI = Graphical User interface

What is the difference between Stand-alone program and web-related Program ?

If the creation, compilation and execution of the program, everything is done in

a single system then it is called Stand-alone Programs OR Desktop Application OR Software.

By using JSE we can develop stand-alone application.

On the other hand, if the creation of the program, compilation of the program and execution of the program, everything is done in different system in different places then it is called web-related application OR Websites.

By using JEE we can develop websites.

-----  
Why java become so popular in the IT Industry ?

-----  
The role of Java compiler :

- a) Compiler are used to check the syntax.
- b) It also checks 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 purpose (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 enhance the readability of the program. It is ignored by the compiler.

In java we have 3 types of comments

- 1) // Single line Comment
- 2) /\* Multiple line  
comment  
\*/
- 3) /\*\*  
Documentation comment  
\*/

Example :

```
/**  
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.

```
-----  
public class Welcome  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello Batch 34!!!!");  
    }  
}
```

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.

-----  
23-05-2024

-----  
static :

-----  
Our main method must be declared as static so JVM need not to create the object to call the main method.

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

Note :- For static member (static variable + static methods) Object is not required, on the other hand for non static member(instance variable + instance method) object is required.

-----  
void :-

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

Eg:

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

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

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

-----  
main() method :

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

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

-----  
Q) Can we write multiple method with same name ?

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

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

```
public class Test { public static void main(String[] args) { System.out.println("Hello Batch 34"); } public static void main(String args) { System.out.println(args); } }
```

-----  
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 multiple values)

In the command line Argument in we have String args, Here String is predefined class available in java.lang package and args is an array variable of type String.

-----  
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 final, static 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.

Actually, It creates HAS-A relation with System and PrintStream class.

-----  
24-05-2024  
-----

Difference between print() and println() method :

print() and println() both are predefined method of PrintStream class available in java.io package.

print() method will print the data in keep the cursor in the same line where as println() will print data and move the cursor to the next line.

Test.java

-----  
public class Test  
{  
 public static void main(String[] args)  
 {  
 System.out.print("Hello India ");  
 System.out.println("Hello Hyderabad");  
 }  
}

-----  
WAP to add two numbers :

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

-----  
How to provide user-friendly message to end user:

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

//Addition of two numbers

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

-----  
WAP to add two numbers without using 3rd variable :

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

-----  
IQ :

```
-----  
public class IQ
{
    public static void main(String[] args)
    {
        String str = 40 + 40 +" NIT " + 90 + 90;
        System.out.println(str); //80 NIT 90 90
    }
}
```

-----  
Command Line Argument :

```
-----  
Whenever we pass an argument to the main method then it is 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 time execution.

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

```
javac Command.java
java Commamnd Scott Smith -> It will print Scott
```

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

```

        {
            System.out.println(args[1]);
        }
    }

javac Command.java
java Commamnd Scott Smith -> It will print Smith

Note :- Whenever we modify the source code we need to re-compile
       the code to generate a new .class file.
-----
Using command line argument we can pass multiple values (String array) and
different types of values (type is String)

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

javac Command.java
java Commamnd 100 200 -> It will print 100

avac Command.java
java Commamnd 78.78 12.90 -> It will print 78.78
-----
WAP to print full name of client using command Line Argument
-----
public class PrintFullNameUsingCommand
{
    public static void main(String[] name)
    {
        System.out.println(name[0]);
    }
}

javac PrintFullNameUsingCommand.java
java PrintFullNameUsingCommand "Virat Kohli"

It will print Virat Kohli because Virat Kohli is in 0th position
-----
WAP to add two numbers using command Line Argument :
-----
public class CommandAdd
{
    public static void main(String[] x)
    {
        System.out.println(x[0]+x[1]);
    }
}

javac CommandAdd.java
java CommandAdd 100 200

It will print 100200 ['+' will work as a concatenation operator]
-----
How to convert String value into integer ?
-----
In order to convert String to integer, Java software people has provided a
predefined class called Integer available in java.lang package.

```

This Integer class contains a predefined static method called `parseInt(String x)` which is accepting String as a parameter and convert this String into int value. The return type of this `parseInt(String x)` is int.

```
public class CommandAdd
{
    public static void main(String[] x)
    {
        int num1 = Integer.parseInt(x[0]);
        int num2 = Integer.parseInt(x[1]);

        System.out.println("Sum is :" +(num1 + num2));
    }
}
```

-----  
25-05-2024

-----  
WAP in java to find out the cube of a number by using command line argument :

```
public class CubeByCommand
{
    public static void main(String[] args)
    {
        //Converting String to integer

        int num = Integer.parseInt(args[0]);

        System.out.println("Cube of "+num+" is :" +(num*num*num));
    }
}
```

-----  
Write a program to show that How `Integer.parseInt()` is working :

```
class Calculate
{
    public static int doSum(int x, int y)
    {
        return (x+y);
    }
}
public class ParseIntDemo
{
    public static void main(String[] args)
    {
        int result = Calculate.doSum(10,20);
        System.out.println("The Sum is :" +result);
    }
}
```

-----  
What is ECLIPSE IDE ?

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

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

-----  
Writing first program using Eclipse IDE :

-----  
Steps to develop program in eclipse IDE :

-----  
File -> new -> project -> java project -> Name of the project (Basic) -> Finish

-> no

Expand the project name (Basic) -> right click on src (source) folder -> new -> select class -> provide the name of the class like Test -> automatically Test.java file will be created.

-----  
What is a package ?

-----  
A package is nothing but folder in windows. The purpose of package to arrange the classes inside a package(folder) so we can reuse these classes from the entire application.

If the arrange the classes based on the packages then fast searching will become possible.

Program that describes how to create a package using command ?

-----  
package com;  
  
public class Test  
{  
  
}

Test.java file contains a package keyword to arrange this Test.class inside com package. In order to compile this code we should use the following command :

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

Note : After compiling the above program, automatically com folder will be created (nothing but package) and Test.class file will be placed inside this com folder.

Package are divided into two types :

-----  
1) Predefined package :

-----  
The packages which are developed by java software people for arranging the classes according to use and nature, as for example all the classes which are performing input and output operation are available in a predefined package called java.io.

Example of predefined packages :

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

-----  
2) Userdefined Packages :

-----  
The packages which are created by user according to nature of the classes to arrange the classes for fast searching are called user-defined packages.

Example of user-defined packages :

com.ravi.calculation, com.ravi.database\_connection and so on

-----  
WAP in java to add two numbers using eclipse IDE :

-----  
package com.ravi.calculation;

-----  
public class Addition {  
  
    public static void main(String[] args)  
    {

```

        int x = 100;
        int y = 200;
        int z = x + y;
        System.out.println("Sum is :" + z);

    }

}

-----  

WAP to find out square of the number using command line argument :  

-----
package com.ravi.command_argument;

public class FindingSquareOfTheNumber {

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

    }

}

In order to execute command line Argument program, right click on the program -> run as -> run configuration -> verify project name and main class -> In the Argument tab, pass appropriate value -> click on the run button  

-----
WAP to find out Area of rectangle :  

-----
package com.ravi.command_argument;

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

        double area = length * breadth;
        System.out.println("Area of Rectangle is :" + area);

    }

}

How to find out the length of an array variable ?  

-----
While working with array we have a property/variable called length through which we can find out the length of an array.

package com.ravi.command_argument;

public class ArrayLength {

    public static void main(String[] args)
    {
        Object []arr = {12, 67.90, "NIT", true, 'A'};
        System.out.println("The length of array is :" + arr.length);

    }

}

```

-----  
WAP using command line argument to get the length value of an array for different test cases

```
package com.ravi.command_argument;

public class ArrayLengthWithCases {

    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("No value from command line args");
        }
        else if(args.length == 1)
        {
            System.out.println(args[0]);
        }
        else if(args.length == 2)
        {
            int x = Integer.parseInt(args[0]);
            int y = Integer.parseInt(args[1]);
            System.out.println("Sum is :" +(x+y));
        }
    }
}
```

-----  
27-05-2024

-----  
Naming Convention in java :

-----  
While writing the java code we should follow the naming convention provided by java language :

1) How to write a class in java :

-----  
While writing the class we should follow pascal naming convention, According to this convention each letter of first word must be capital and there should no be any space between the word. In java a class represents noun.

Example :

```
-----  
ThisIsExampleOfClass  
String  
System  
Integer  
DataInputStream  
BufferedRedaer  
ArrayIndexOutOfBoundsException
```

2) How to write method in java :

-----  
While writing a method we should follow camel case naming convention, According to this convention first word will be small, second word onwards, first letter of each word must be capital. In java a method represents verb.

Example :

```
-----  
thisIsExampleOfMethod()  
parseInt()  
readLine()
```

```
charAt()  
toUpperCase()  
toLowerCase()
```

### 3) How to write a field in java :

While writing a field we should follow camel case naming convention, It is similar to method, only difference is method contains () symbol.

Example :

```
thisIsExampleOfFiled  
rollNumber  
customerBill  
customerSalary  
employeeNumber
```

### 4) How to write a final filed OR variable in java :

While writing the final variable (constant variable) we should write all the characters in capital, if multiple words are there then provide \_ symbol for each word separation.

Example :

```
final double PI = 3.14;  
        SIZE;  
        MAX_VALUE;  
        MIN_VALUE;
```

### 5) How to write a package in java :

While writing the packages we should write all the characters in small letters as we as It will be reverse of company name.

Example :

```
com.tcs.basic;  
com.ravi.inheritance;  
com.nit.oop;
```

What is a Token ?

A token is the smallest unit of the program which is identified by the compiler.  
A token is divided into 5 types.

- 1) Keywords
- 2) Identifiers
- 3) Literals
- 4) Punctuators (Separators)
- 5) Operator

Keyword :-

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

In java all the keywords must be in lowercase only.

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

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

**Identifiers :**

A name in java program by default considered as identifiers.

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

We can't use keyword as an identifier.

Ex:-

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

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

**Rules for defining an identifier :**

- 1) Can consist of uppercase(A-Z), lowercase(a-z), digits(0-9), \$ sign, and underscore (\_)
- 2) Begins with letter, \$, and \_
- 3) It is case sensitive
- 4) Cannot be a keyword
- 5) No limitation of length

**Literals :-**

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.

**Integral Literal :**

If a numeric literal does not contain decimal or fraction then it is called Integral Literal.

**Example :**

12, 90, 67, 45

In integral literal we have 4 data types

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

In Java, An integral literal we can represent in 4 different ways :

- a) Decimal Literal (Base 10)
- b) Octal Literal (Base 8)
- c) Hexadecimal Literal (Base 16) [0 - 9 and A - F]
- d) Binary Literal (Base 2)

a) Decimal Literal :

-----  
By default every integral literal is decimal literal. Here the base is 10 so it accepts 10 digits i.e. 0 to 9.

Example :

```
int x = 12;  
int y = 15;  
int z = 999;
```

b) Octal Literal :

-----  
If any integral literal starts with 0 (zero) then it is called Octal literal. Here base is 8 so, It accepts 8 digits 0 to 7.

Example :

```
-----  
int x = 015; //Valid  
int x = 017; //Valid  
int x = 018; //Invalid [Digit 8 is out of the range]
```

c) Hexadecimal Literal :

-----  
If any integral literal starts with 0X (zero capital X) or 0x (zero small x) then it is called Hexadecimal literal. Here base is 16 so, It accepts 16 digits 0 to 9 and A - F.

Example :

```
-----  
int x = 0Xadd; //Valid  
int x = 0xface; //Valid  
int x = 0Xage; //Invalid [Here g is out of the range]
```

Binary Literal :

-----  
It came from java 1.7 onwards. If any integral literal starts with 0B (zero capital B) or 0b (Zero small b) then it is called Binary literal. Here base is 2 so, It accepts 2 digits 0 and 1.

Example :

```
-----  
int x = 0B101; //valid  
int y = 0b111; //Valid  
int z = 0B121; //Invalid
```

Note : In java, Programmer has flexibility to represent an integral literal in four different forms i.e decimal, octal, hexadecimal and binary but JVM always produce the result in decimal format so JVM internally converts these number system into decimal to get the output.

-----  
Octal Literal Program :

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

```
        int x = 015;
        System.out.println(x);

    }

-----
public class Test
{
    public static void main(String [] args)
    {
        int x = 017;
        System.out.println(x);

    }
}
```

```
}
```

---

```
Hexadecimal Literal Program :
```

---

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

    }
}
```

---

```
28-05-2024
```

---

```
//Program on Binary Literal
public class BinaryLiteral
{
    public static void main(String[] args)
    {
        int x = 0b101;
        System.out.println(x);

        int y = 0B111;
        System.out.println(y);

    }
}
```

---

```
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.

---

```
/* By default every integral literal is of type int only*/
public class Test4
```

```

{
public static void main(String[] args)
{
    byte b = 128;
    System.out.println(b);

    short s = 32768;
    System.out.println(s);
}
-----
Type casting :

-----  

Converting one data type to another data type.  

It is of two types

1) Implicit OR Automatic OR Widening
-----
Assigning smaller data type to bigger data type.  

Example :

byte b = 12;
short s = b;

2) Explicit OR Manual OR Narrowing
-----
Assigning bigger data type value to smaller data type. User need  

to perform explicitly, here there is chance of loss of data.

short s = 127;
byte b = (byte) s;
-----
//Program on Automatic Type Casting
//Assigning smaller data type value to bigger data type
public class Test5
{
public static void main(String[] args)
{
    byte b = 125; //byte b =(byte) 125;
    short s = b; //Automatic OR Implicit OR Widening
    System.out.println(s);
}
-----
//Converting bigger type to smaller type
public class Test6
{
public static void main(String[] args)
{
    short s = 127;
    byte b = (byte) s; //Explicit OR Narrowing OR manual
    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);

    }
}

-----
Java is a pure object oriented language or not ?
-----
Java is not a pure object oriented language because it is accepting primary data
types (byte, short, int, long and so on)

```

Any language which is supporting primary data types is not treated as a pure object oriented language.

The languages which are accepting directly objects are called pure object oriented language.

Example : Ruby, smalltalk and so on

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

From java 1.5 onwards, Java software people has provided the following two concepts to convert primary data type into corresponding object and vice versa.

### 1) Autoboxing : Converting Primitive into Warpper object

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

### 2) Unboxing : Converting Wrapper object back to primitive data type

Wrapper object -	Primitive data type
Byte	- byte
Short	- short
Integer	- int
Long	- long
Float	- float
Double	- double
Character	- char
Boolean	- boolean

```

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

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

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

```

```

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

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

All these Wrapper classes (Byte, Short, Integer and Long) has provided the following final and static variables to find out minimum and maximum value as well as range of data type.

final and static variables :

MIN\_VALUE : Will provide minimum value

MAX\_VALUE : Will provide maximum value

SIZE : Will provide the size of the data type

Example :

```

Byte.MIN_VALUE    -> -128
Byte.MAX_VALUE   -> 127
Byte.SIZE         -> 8 bits (1 byte)
-----
```

//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);
    }
}
```

From java 1.7 onwards we can provide \_ symbol while working with integral literal to enhance the readability of the number

```

//We can provide _ in integral literal
public class Test10
{
    public static void main(String[] args)
    {
        long mobile = 98_1234_5678L;
```

```

        System.out.println("Mobile Number is :" + mobile);
    }

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

```

Here x is final so we can assign to byte but within the range.

29-05-2024

```

// Converting from decimal to another number system
public class Test12
{
    public static void main(String[] argv)
    {
        //decimal to Binary
        System.out.println(Integer.toBinaryString(7)); //101

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

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

```

Note :- Integer class has provided the following static methods to convert decimal number to binary, octal and hexadecimal

- 1) Integer.toBinaryString(int x) : Convert decimal to binary.
- 2) Integer.toOctalString(int x) : Convert decimal to octal.
- 3) Integer.toHexString(int x) : Convert decimal to hexadecimal.

var keyword in java :

It is introduced from JDK 10v.

It is providing the flexibility to assign different kinds of value (literal and object) to the variable declared with var keyword.

We need to assign the value at the time of declaration otherwise we will get compilation error (We can't initialize later) because the type of variable declared with var keyword decided at time of initialization(statically typed language)

We can use var keyword for local variable only.

```

//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 :
```

---

```
If a numeric literal contains decimal or fraction then it is called floating point literal.
```

Example : 12.78, 78.67, 1.34

In floating point literal we have 2 data types

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

In floating point literal, It is by default of type double only so, the following statement will generate compilation error

```
float f = 1.2; //error
```

Now we have the following 3 solutions

```
float f1 = 1.2f;  
float f2 = 12.89F;  
float f3 = (float) 12.56;
```

Even though, every floating point literal is of type double only but still compiler has provided the following two flavours to represent the double value explicitly hence the readability of the code will increase.

```
double d1 = 12.89d;  
double d2 = 34.78D;
```

We can also represent floating point literal in exponent form.  
double d = 15e3; [15 X 10 to the power of 3]

While working with integral literal, we can represent an integral literal in 4 different forms like decimal, octal, hexadecimal and binary but while working with floating point literal we can represent in only one form i.e decimal.

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

---

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


---



```
public class Test1  
{  
    public static void main(String[] args)  
    {  
        float b = 15.29F;  
        float c = 15.25f;  
        float d = (float) 15.30;  
  
        System.out.println(b+" : "+c+" : "+d);  
    }  
}
```


```

```
-----  
public class Test2  
{  
    public static void main(String[] args)  
    {  
        double d = 15.15;  
        double e = 15d;  
        double f = 15.15D;  
  
        System.out.println(d+" , "+e+" , "+f);  
    }  
}  
-----  
public class Test3  
{  
    public static void main(String[] args)  
    {  
        double x = 0129.89;  
  
        double y = 0167;  
  
        double z = 0178; //error  
  
        System.out.println(x+" , "+y+" , "+z);  
    }  
}  
-----  
class Test4  
{  
    public static void main(String[] args)  
    {  
        double x = 0X29;  
  
        double y = 0X9.15;  
  
        System.out.println(x+" , "+y);  
    }  
}  
-----  
public class Test5  
{  
    public static void main(String[] args)  
    {  
        double d1 = 15e-3;  
        System.out.println("d1 value is :" + d1);  
  
        double d2 = 15e3;  
        System.out.println("d2 value is :" + d2);  
    }  
}  
-----  
public class Test6  
{  
    public static void main(String[] args)  
    {  
        double a = 0791; //error  
  
        double b = 0791.0;  
  
        double c = 0777;  
  
        double d = 0Xdead;  
  
        double e = 0Xdead.0; //error  
    }  
}
```

```

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

        System.out.println("\n Double range:");
        System.out.println(" min: " + Double.MIN_VALUE);
        System.out.println(" max: " + Double.MAX_VALUE);
        System.out.println(" size :" +Double.SIZE);
    }
}
-----
boolean literal :
-----
boolean literal is used to represent two states i.e true or false
```

We have only one data type i.e boolean data type which accepts 1 bit of memory OR depnds upon JVM implementation.

```

    boolean isEmpty = true;
    boolean isValid = false;
```

Unlike C and C++, we can't assign numeric value to boolean variable, if we assign we will get compilation error.

```
boolean isValid = 0; [Valid in C but invalid in java]
```

We cannot assign String literal to boolean type.

```

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

        System.out.println(isValid);
        System.out.printlnisEmpty);
```

```

    }
}

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);
    }
}

```

30-05-2024

Character Literal :

It is also known as char literal.

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

char literal we can represent in different ways :

a) Single Character enclosed with single quotes.

```
char ch = 'A';
```

b) char literal we can assign to integral literal to know the UNICODE value of the character.

```
int x = 'A';
```

c) In older language like C language which supports ASCII format and here the range is 0 to 255 but Java support UNICODE where range is 0 - 65535 (Max value of hexadecimal)

```
char ch = 65535; //Valid within the range
char ch1 = 65536; //Invalid out of the range
```

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

```
'\uXXXX'
```

Here u represents UNICODE  
X represents the digit

The range is : '\u0000' to '\uffff'

e) All the escape sequences we can represent as char literal

```

        char ch = '\n';
-----
public class Test1
{
    public static void main(String[] args)
    {
        char ch1 = 'a';
        System.out.println("ch1 value is :" + ch1);

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

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

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

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

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

```

Note :- We will get the Output ? we already know the character ? Unicode value is 63 but here we are getting 63 because to represent the character 47000 and 2782 we don't have equivalent language translator support.

```

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

```

```

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

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

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

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

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

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

        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("Hello");
            System.out.println(ch);
        }
    }

-----  

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

-----  

//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)
            {
                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 :-

We can create String in java by using 3 ways :

- 1) By using String Literal (Most popular)

```
String str = "India";
```

- 2) By using new keyword

```
String str1 = new String("Hyderabad");
```

- 3) By using Character Array (Old technique)

```
char ch[] = {'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);
    }
}

```

31-05-2024

#### 4) Punctuators

It is also called as separators.

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

Example :- (), {}, [], ; , ... (var args)

#### 5) Operator :

An operator is a symbol that describes how the calculation will be performed on the operands.

Example :- a + b; //Here a and b both are operands where as '+' is operator

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 Operator (`&`    `|`)
  - 7) Bitwise Operator (`~`)
  - 8) Ternary Operator
  - \*9) new Operator
  - \*10) Dot Operator (`.` [Member Access Operator])
  - \*11) instanceof operator
- 

What is BLC and ELC class ?

---

BLC : It stands for Business Logic class. It is the class which is meant for writing the business logic and we should not write main method in the BLC class.

ELC : It stands for Executable Logic class. It is class through which we can execute our program so we should write main method inside the ELC class.

2 files :

---

Calculation.java (BLC)

---

package com.ravi.blc\_elc;

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

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

Main.java (ELC)

---

package com.ravi.blc\_elc;

```
//ELC
public class Main
{
    public static void main(String[] args)
    {
        Calculation c = new Calculation();
        c.doSum(12, 12);
        c.getSquare(10);
    }
}
```

---

How to read the value from the user :

---

In order to read the value from the user we are using command line Argument but with command line argument the limitation is we can't ask any message from our end user.

```
package com.ravi.blc_elc;  
  
public class ReadName {  
  
    public static void main(String[] args)  
    {  
        System.out.println("Enter your Name :");  
        System.out.println("Your Name is :" + args[0]);  
    }  
}
```

-----  
How to take the input from the user :

-----  
In order to take the input from the user, Java software people has provided a predefined class called Scanner available in java.util package. It is available from java 1.5v.

static variable of System class :

-----  
As we know System is a predefined class available in java.lang package which provided the following static variables :

- a) System.out :- It is used to print normal message.
- b) System.err :- It is used to print error message. (Red Colour)
- c) System.in :- It is used to take the input from the source.

How to create an object for Scanner class :

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

Methods of Scanner class :

- 1) public String next() : Will read single word.
- 2) public String nextLine() : Will read complete line. (Multiple Words)
- 3) public byte nextByte() : Will read byte value.
- 4) public short nextShort() : Will read short value.
- 5) public int nextInt() : Will read int value.
- 6) public long nextLong() : Will read long value.
- 7) public float nextFloat() : Will read float value.
- 8) public double nextDouble() : Will read double value.
- 9) public boolean nextBoolean() : Will read boolean value.
- 10) public char next().charAt(0) : Will read a character.

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

```
import java.util.Scanner;  
  
public class ReadName
```

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

-----  
WAP to read gender from the keyboard.

How to read a character :

-----  
As we know we can read a single word by using next() method so from this single word we can also retrieve a character by using charAt() method of String class, this charAt(int indexPosition) will retrieve the character based on the index position.

ReadCharacter.java

```
-----  
package com.ravi.blc_elc;  
  
import java.util.Scanner;  
  
public class ReadCharacter  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter your gender:");  
  
        char gender = sc.next().charAt(0);  
        System.out.println("Your gender is :" + gender);  
        sc.close();  
    }  
}
```

-----  
WAP to read employeeNumber and employeeName from Scanner class :

```
-----  
package com.ravi.blc_elc;  
  
import java.util.Scanner;  
  
public class ReadEmployeeData {  
  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
  
        System.out.print("Enter employee Number :");  
        int employeeNumber = sc.nextInt(); // 78  
  
        System.out.print("Enter Employee Name :");  
        String employeeName = sc.nextLine(); // Buffer Problem  
        employeeName = sc.nextLine(); // Actual Data  
  
        System.out.println("Employee Number is :" + employeeNumber);  
        System.out.println("Employee Name is :" + employeeName);  
        sc.close();  
    }  
}
```

```
}
```

```
-----  
01-06-2024
```

```
-----  
What is local and parameter variable ?
```

```
-----  
Local Variable :
```

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

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

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

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

```
Example :
```

```
public void accept()  
{  
    int x = 100; //Local variable [only final modifier]  
}
```

```
-----  
Why we can't use local variable outside of the method ?
```

```
-----  
All the methods in java are executed in a special memory called  
STACK memory.
```

```
STACK memory works on LIFO (Last In First Out) basis.
```

```
In Java, whenever we call a method, everytime one new stack frame will be  
created and this stack frame contains 3 parts :
```

- 1) Local Variable
- 2) Frame Data
- 3) Operand Stack

```
//Program on local variable execution :
```

```
-----  
package com.ravi.stack_demo;  
  
public class MethodExecution  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main method started");  
        m1();  
        System.out.println("Main method ended");  
    }  
  
    public static void m1()  
    {  
        System.out.println("m1 method started");  
        m2();  
        System.out.println("m1 method ended");  
    }  
  
    public static void m2()  
    {
```

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

-----  
Parameter variable :

-----  
If we declare a variable inside the method parameter (not inside the method body) then it is called parameter variable. It is used to receive the value from outer world so we can use the outside value in the body of the method.

Example :

```
-----  
public void doSum(int x , int y) //x and y both are parameter
{                               variable
    System.out.println(x+y);
}
```

-----  
Promotion of type with Operator :

-----  
While working with Arithmetic operator and Unary minus operator, after expression calculation the result will be promoted to int type in other words to store the result minimum 32 bits i.e int type is required.

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

    }
}
```

```
-----  
public class Test
{
    public static void main(String [] args)
    {
        short s = 12;
        short t = 90;
        short u = s * t; //error

    }
}
```

```
-----  
public class Test
{
    public static void main(String [] args)
    {
        byte b = 12;
        byte c = 24;
        c += b;
        System.out.println(c);

    }
}
```

Here the code will compile and execute because it is short hand operator.

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

```
{  
    public static void main(String [] args)  
    {  
        byte b = 1;  
        byte c = -b; //error (Unary minus opertor)  
    }  
}
```

---

#### Boolean Operators :

---

Boolean Operators work with boolean values that is true and false. It is used to perform boolean logic upon two boolean expressions.

It is also known as non short circuit. There are two non short circuit logical operators.

&   boolean AND operator   (All condions must be true but if first expression is false still it will check all right side expressions)

|   boolean OR operator   (At least one condition must be true but if the first condition is true still it will check all right side expression )

//Program on Boolean AND

```
public class Test  
{  
    public static void main(String [] args)  
    {  
        int z = 5;  
  
        if(++z > 6 & ++z > 8)  
        {  
            ++z;  
        }  
        System.out.println(z); //7  
    }  
}
```

//Program on Boolean OR

```
public class Test  
{  
    public static void main(String [] args)  
    {  
        int z = 5;  
        if(++z > 5 | ++z > 6) //Boolean OR  
        {  
            z++;  
        }  
        System.out.println(z); //8  
    }  
}
```

---

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

```
    }
}
-----
Bitwise complement operator (~) :
-----
It does not work with boolean.

public class Test
{
    public static void main(String [] args)
    {
        //System.out.println(~ true);
        System.out.println(~ -6); //5
        System.out.println(~ 6);//-7
    }
}
```

03-06-2024

Control Statements :

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 dpendng 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.

The data type/ reference type we can pass in switch case :

Allwoed Data type : byte, short, int and char

Data type/ reference which are not allowed : long, float, double and boolean

Strings are allowed from JDK 1.7v

Enums are allowed from JDK 1.5v.

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

        short b = 12;

        switch(b)
        {
            case 12 :
                System.out.println("12");
            case 13 :
                System.out.println("13");
        }
    }
}
```

```
}
```

---

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

        byte b = 121;

        switch(b)
        {
            case 121 :
                System.out.println("121");
                break;
            case 128 : //error 128 is int
                System.out.println("133");
        }
    }
}
```

---

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

        int x = 12;
        final int y = 12;

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

        }
    }
}
```

In the switch label, we can take a variable but that variable must be final.

---

```
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.next().toLowerCase();

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

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

-----  

Loops in java :
-----
A loop is nothing but repeatation of statement based on some condition.

```

In java we have 4 types of loop :

- 1) do-while loop (exit control loop)
- 2) while loop (entry control loop)
- 3) for loop
- 4) for each loop

Program on do - while loop :

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

Here 'x' variable is a block level variable so we can't use x variable outside of the do block, it's scope is limited to do block only.

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

Program on While loop :

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

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

Program on for loop :

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

```
        for(int i=1; i<=10; i++)
    {
        System.out.println(i);
    }
}
```

-----  
For-Each loop in java :

-----  
for-each loop in java is introduced from JDK 1.5v

It is also known as enhanced for loop.

It is mainly used to retrieve the values from the collection one by one so it is known as for each loop.

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

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

Note :- Arrays is a predefined class available in java.util package which contains a predefined static method sort() through which we can sort the data in ascending order.

```
-----  
package com.ravi.forEach;

import java.util.Arrays;

public class ForEachDemo2
{
    public static void main(String[] args)
    {
        String []fruits = {"Orange", "Mango", "Apple", "Kiwi"};

        Arrays.sort(fruits);

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

Here also sort(Object []) method is used to sort the String data based on the alphabetical OR Dictionary order.

-----  
04-06-2024

-----  
Working with static method with different types :

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

```
public class Test1
```

```

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

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

```

Note :- from the above program it is clear that a static method we can directly call with another static method from the same class.

-----  
Some important points to remember :

- 1) If any method return type is void then we can't call/invoke the method from System.out.println() because that method is not returning any value as well as println() method is not getting any return value.

Example :

```

-----
public class Test2
{
    public static void main(String[] args)
    {
        System.out.println(doSum(12,90));
    }

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

```

- 2) We can't declare multiple public classes in a single .java file.  
We should declare only one public class per .java file.

If we don't declare our java classes as public then it can't be reusable from another package.

2 files :

-----  
GetSquare.java

```

-----
package com.ravi.pack2;

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

```

Test2.java

```

-----
package com.ravi.pack2;

import java.util.Scanner;

```

```

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

```

Note :- GetSquare is a public class so we can reuse this class from any another package.

For the Re-usability purpose we should always separate classes in BLC and ELC.

---

2 files :

---

FindSquare.java

---

```

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

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

```

Test3.java

---

```

package com.ravi.pack3;

import java.util.Scanner;

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

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

```

---

2 files :

---

```
/*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  
*/
```

Calculate.java

```
-----  
package com.ravi.pack4;  
  
//BLC  
public class Calculate  
{  
    public static int getSquareAndCube(int num)  
    {  
        if(num <= 0)  
        {  
            return -1;  
        }  
        else if(num%2==0)  
        {  
            return (num*num);  
        }  
        else  
        {  
            return (num*num*num);  
        }  
    }  
}
```

Test4.java

```
-----  
package com.ravi.pack4;  
  
import java.util.Scanner;  
  
public class Test4  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter a number :");  
        int num = sc.nextInt();  
  
        int result = Calculate.getSquareAndCube(num);  
        System.out.println("Result is :" + result);  
        sc.close();  
    }  
}
```

-----  
2 files :

-----  
Rectangle.java(c)

```
-----  
package com.ravi.pack5;  
  
//BLC  
public class Rectangle
```

```
{  
    public static double getAreaOfRectangle(double length, double breadth)  
    {  
        return (length * breadth);  
    }  
}
```

Test5.java

```
-----  
package com.ravi.pack5;  
  
import java.util.Scanner;  
  
public class Test5  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter the length of the Rect :");  
        double length = sc.nextDouble();  
        System.out.print("Enter the breadth of the Rect :");  
        double breadth = sc.nextDouble();  
  
        double areaOfRectangle = Rectangle.getAreaOfRectangle(length, breadth);  
        System.out.println("Area of Rectangle is :" + areaOfRectangle);  
        sc.close();  
    }  
}
```

-----  
05-06-2024

-----  
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 isEven = EvenOrOdd.isEven(11);  
        System.out.println("number is Even ?: " + isEven);  
  
        isEven = EvenOrOdd.isEven(12);  
        System.out.println("number is Even ?: " + isEven);  
    }  
}
```

```
}
```

```
-----  
How to provide userdefined format for decimal numbers ?
```

```
-----  
java.text package has provided a predefined class called DecimalFormat  
which is accepting format as a String parameter as shown below.
```

```
DecimalFormat df = new DecimalFormat("00.00"); // "00.00" is the format which  
is available in String type (String pattern)
```

```
Now this DecimalFormat class contains non static method format() which accepts  
double as a parameter.
```

```
DecimalFormat df = new DecimalFormat("00.00");  
df.format(double value);
```

```
-----  
Circle.java
```

```
-----  
//Area of Circle  
//If the radius is 0 or Negative then return -1.  
  
package com.ravi.pack7;  
public class Circle  
{  
    public static String getAreaOfCircle(double rad)  
    {  
        if(rad <=0)  
        {  
            return ""+(-1);  
        }  
        else  
        {  
            final double PI = 3.14;  
            double areaOfCircle = PI * rad * rad;  
            return ""+areaOfCircle;  
        }  
    }  
}
```

```
Test7.java
```

```
-----  
package com.ravi.pack7;  
  
import java.text.DecimalFormat;  
import java.util.Scanner;  
  
public class Test7  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter the radius of the Circle :");  
        double radius = sc.nextDouble();  
  
        String areaOfCircle = Circle.getAreaOfCircle(radius);  
  
        //Converting String to double  
        double area = Double.parseDouble(areaOfCircle);  
  
        DecimalFormat df = new DecimalFormat("000.000");  
        System.out.println("Area of Circle is :" + df.format(area));  
    }  
}
```

```

        sc.close();
    }
}
-----
2 files :
-----
Student.java
-----
package com.ravi.pack8;

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

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

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

public class Test8
{
    public static void main(String[] args)
    {
        String studentDetails = Student.getStudentDetails(101, "Scott",
14000);
        System.out.println(studentDetails);
    }
}
-----
2 files :
-----
Table.java
-----
package com.ravi.pack9;

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

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

//ELC

```

```
public class Test9
{
    public static void main(String[] args)
    {
        for(int i=1; i<=9; i++)
        {
            Table.printTable(i);
        }
    }
}
```

-----  
Object Oriented Programming (OOPs) :

-----  
What is an Object?

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

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

An Object is having 3 characteristics :

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

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

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

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

Advantages of OOP :

- 1) Modularity (Dividing the bigger task into smaller task)
- 2) Reusability (We can reuse the component so many times)
- 3) Flexibility (Easy to maintain)

Features of OOP :

- 1) Class
- 2) Object
- 3) Abstraction
- 4) Encapsulation
- 5) Inheritance
- 6) Polymorphism

-----  
What is a class?

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

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

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

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

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

-----  
First Object Oriented Program :

-----  
Steps to develop first Object Oriented Program

- 1) Based on the class create the object in ELC class.
- 2) Based on thinking and imagination, write the properties of object in BLC class.
- 3) Write the behavior of Object in the BLC class.
- 4) Initialize all the object properties through object reference in ELC class.
- 5) Call the behavior of the object.

2 files :

-----  
Student.java(BLC)

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

```
//BLC
public class Student
{
    //Object Properties
    int rollNumber;
    String studentName;
    double studentHeight;

    //Object Behavior
    public void talk()
    {
        System.out.println("My roll Number is :" + rollNumber);
        System.out.println("My name is :" + studentName);
        System.out.println("and My Height is :" + studentHeight);
    }

    public void writeExam()
    {
        System.out.println("I am " + studentName + ". I write exam weekly");
    }
}
```

-----  
StudentDemo.java(ELC)

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

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

        //Initializing the object properties
        raj.rollNumber = 111;
        raj.studentName = "Raj Gourav";
        raj.studentHeight = 5.9;

        //call the behavior
        raj.talk();
        raj.writeExam();
    }
}
```

```

        System.out.println(".....");
        Student priya = new Student();
        //Initializing the Object Properties
        priya.rollNumber = 222;
        priya.studentName = "Priya";
        priya.studentHeight = 5.8;

        priya.talk();
        priya.writeExam();
    }
}

```

#### Types of variable in Java :

In Java based on data type, we have only 2 types of variables

- 1) Primitive type (byte, short, int , long and so on)

Example :- int x = 10; [Primitive type will always hold the value]

- 2) Reference Variable

Example :- Employee e1 = new Employee(); [Reference variable will hold the address]

//Here e1 is a reference variable

Now further based on the Declaration position variable is divided into 4 types :

- 1) Class variable OR Static field
- 2) Instance variable OR Non-static field
- 3) Local/Stack/Automatic/Temporary variable
- 4) Parameter Varaiable.

#### PrimitiveVariable.java

```

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

public class PrimitiveVariable
{
    int x = 100; //Instance Variable OR Non static Field

    static int y = 200; //Class Variable OR Static Field

    public static void main(String[] args)
    {
        System.out.println("Class Variable :"+PrimitiveVariable.y);
        PrimitiveVariable p = new PrimitiveVariable();
        System.out.println("Instance Variable :"+p.x);
        accept(300);
    }

    public static void accept(int z)
    {

        int local = 400;
        System.out.println("Parameter Variable :"+z);
        System.out.println("Local Variable :"+local);
    }
}

```

```

        }
    }

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

import java.util.Scanner;

class Employee
{
    public void getSalary()
    {
        System.out.println("Employee Salary");
    }
}

public class ReferenceVariable
{
    Employee e1 = new Employee(); //Reference + Instance
    static Scanner sc = new Scanner(System.in); //Reference + Static

    public static void main(String[] args)
    {
        Employee e2 = new Employee(); //Reference + Local
        accept(e2);
    }

    public static void accept(Employee emp) //Reference + Parameter
    {
        emp.getSalary();
    }
}
-----
```

07-06-2024

-----  
Instance variable :

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

```

public class Example
{
    int x = 100; //Primitive Instance Variable
    Example e = new Example(); //Reference Instance Variable
}
```

The scope of instance variable is within the class as well as outside of the class but depends upon the access modifier we have applied on instance variable.

The life of an instance variable starts with Object creation that means without object we can't think about instance variable.

```

public class Test
{
    int x = 100;

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

```
        System.out.println(x); //error
    }
}
```

-----

Initializing the object property through method :

-----

We can also initialize the object properties through methods so whenever we call the method, properties will initialize with our user-defined values as shown in the program.

2 files :

-----

Employee.java

-----

```
package com.nit.oop;

public class Employee
{
    int employeeNumber;
    String employeeName;

    //Initializing the object properties through methods
    public void setEmployeeData()
    {
        employeeNumber = 111;
        employeeName = "Scott";
    }

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

EmployeeDemo.java

-----

```
package com.nit.oop;

public class EmployeeDemo
{
    public static void main(String[] args)
    {
        Employee scott = new Employee();
        scott.setEmployeeData();
        scott.getEmployeeData();

    }
}
```

-----

What is a constructor (Introduction Part)

-----

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

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

Default constructor added by the compiler :

In java, whenever we write a class and if we don't write any type of constructor in our class then automatically one default constructor will be added by the compiler in the class.

```
public class Customer
{
    //user has not written any type of constructor
}

javac Customer.java (Compilation of Java Program)

public class Customer
{
    public Customer() //default constructor added by the compiler.
    {
    }
}
```

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

The access modifier of default constructor depends upon the class access modifier that means if class is public then the default constructor added by the compiler is also public.

-----

Why compiler is adding default constructor to our class :

-----

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

- 1) Without default constructor, Object creation is not possible in java.
- 2) default constructor will initialize all the instance variables with default values.

Data type - Default value

byte - 0

short - 0

int - 0

long - 0

float - 0.0

double - 0.0

char - (space) '\u0000'

boolean - false

String - null

Object - null (For any class i.e reference variable the default value is null)

-----

//Program that describes default values will be provided by the default constructor.

```
package com.nit.oop;

public class Customer
{
    int customerId;
    String customerName;

    public void show()
    {
        System.out.println("Customer id is :" +customerId);
        System.out.println("Customer Name is :" +customerName);
    }
}
```

```
public static void main(String[] args)
{
    Customer c1 = new Customer();
    c1.show();
}
-----
```

08-06-2024

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

-----  
The default values provided by default constructor like 0, 0.0, false, null are not useful for the user so, user can provide own user-defined values by using method setManagerData() as shown in the program below.

2 files :

-----  
Manager.java

-----  
package com.nit.oop;

//BLC  
public class Manager  
{

int managerId;  
 String managerName;

public void setManagerData()

{  
 managerId = 111;  
 managerName = "Dr. Smith";  
 }

public void getManagerData()

{  
 System.out.println("Manager Id is :" +managerId);  
 System.out.println("Manager Name is :" +managerName);  
 }

}

ManagerDemo.java

-----  
package com.nit.oop;

public class ManagerDemo

{

public static void main(String[] args)  
{

Manager smith = new Manager();  
 smith.getManagerData();

smith.setManagerData(); //Initializing the properties through Methods  
 smith.getManagerData();

}

}

-----  
2 files :

-----  
//Program on instance variable, initializing through Scanner class.  
Student.java

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

import java.util.Scanner;

public class Student
{
    int studentId;
    String studentName;

    public void setStudentData()
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Student Id :");
        studentId = sc.nextInt();
        System.out.print("Enter Student Name :");
        studentName = sc.nextLine();
        studentName = sc.nextLine();
        sc.close();
    }

    public void getStudentData()
    {
        System.out.println("Student Id is :" + studentId);
        System.out.println("Student Name is :" + studentName);
    }
}
```

StudentDemo.java

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

public class StudentDemo {

    public static void main(String[] args)
    {
        Student raj = new Student();
        raj.setStudentData();
        raj.getStudentData();

    }
}
```

-----  
How to initialize the instance variable through parameter variable :

-----  
As we know we can receive the outer world value from parameter variable so, with this parameter variable we can initialize the instance variable as shown in the program

2 files :

-----  
Player.java

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

public class Player
{
    String playerName;
    double basePrice;

    public void setPlayerData(String name, double price)
    {
```

```
    playerName = name;
    basePrice = price;
}

public void getPlayerData()
{
    System.out.println("Player Name is :" + playerName);
    System.out.println("Player base price is :" + basePrice + "Cr");
}

}
```

PlayerDemo.java

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

public class PlayerDemo
{
    public static void main(String[] args)
    {
        Player p1 = new Player();
        p1.setPlayerData("Rohit", 3.4);
        p1.getPlayerData();
    }
}
```

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

2 files :

-----  
Employee.java

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

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

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

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

    public void calculateEmployeeGrade()
    {
        if(employeeSalary >= 100000)
```

```
        {
            employeeGrade = 'A';
        }
    else if(employeeSalary >= 75000)
    {
        employeeGrade = 'B';
    }
    else if(employeeSalary >= 50000)
    {
        employeeGrade = 'C';
    }
    else
    {
        employeeGrade = 'D';
    }
}
}
```

EmployeeDemo.java

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

public class EmployeeDemo {

    public static void main(String[] args)
    {
        Employee scott = new Employee();
        scott.setEmployeeData(1, "Scott", 155000);
        scott.calculateEmployeeGrade();
        scott.getEmployeeData();

    }
}
```

What is variable shadowing ?

If the name of instance variable and local/Parameter variable both are same then inside the method the local/Parameter variable will hide instance variable that is known as Variable Shadow so, always local and parameter variables are having more priority than instance variable inside the method body as shown in the program below.

Sample.java

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

public class Sample
{
    int id = 101;
    String name = "Ravi";
    double salary = 20000;

    public void showData(double salary)
    {
        int id = 201;
        String name = "Raj";

        System.out.println("Id is :" + id);
        System.out.println("Name is :" + name);
        System.out.println("Salary is :" + salary);
    }
}
```

```
public static void main(String[] args)
{
    Sample s1 = new Sample();
    s1.showData(35000);
}

}
```

In the above program we will get the output of local/parameter variable because instance variables are hidden by local/parameter variables.

-----  
10-06-2024  
-----

this keyword in java :

Whenever 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 instance variable and which one is parameter variable (Due to variable shadow). Inside the method body always local and parameter variable is higher priority.

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

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

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

-----  
2 files :

Customer.java

```
package com.ravi.oop;

public class Customer
{
    int customerId ;
    String customerName;

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

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

CustomerDemo.java

```
package com.ravi.oop;

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

```
        Customer raj = new Customer();
        raj.setCustomerData(111, "Raj Gourav");
        raj.getCustomerData();
    }
}
```

-----  
11-06-2024

-----  
Role of instance variable while creating the Object :

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

```
public class Student
{
    int rollNumber;
}

Student s1 = new Student(); //rollNumber copy will be created
Student s2 = new Student(); //rollNumber copy will be created
Student s3 = new Student(); //rollNumber copy will be created
-----  
Test.java
-----
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
    }
}
```

-----  
Role of static variable while creating the object :

-----  
In java, whenever we create an object then all the objects will share a single copy of static variable hence if we modify the static variable by using any object then its value will be reflected to all the objects.

```
package com.nit.oop;

public class Demo
{
    static int x = 10;

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

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

```

        System.out.println(d1.x); //12
        System.out.println(d2.x); //12
    }
}

```

So the conclusion is :

Instance variable = Multiple Copies with multiple objects  
 Static Variable = Single copy with all the objects

-----  
 As a developer when we should declare a variable as an instance variable and when we should declare a variable as a static variable ?

If the value of the variable is different with respect to object then we should use instance variable where as on the other hand, If the value of the variable is common for all the objects then we should use static variable.

```

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

```

Note :- static variables are mainly used to save the memory.

-----  
 //Program on instance and static variable :

-----  
 2 files :

-----  
 Student.java

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

```

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

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

```

```

    public void getStudentData()
    {
        System.out.println("Student Roll number is :" +this.rollNumber);
        System.out.println("Student Name is :" +this.studentName);
        System.out.println("Student 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;

public class StudentDemo {
    public static void main(String[] args)
    {
        Student raj = new Student();
        raj.setStudentData(1, "Raj", "Indore");
        raj.getStudentData();

        Student scott = new Student();
        scott.setStudentData(2, "Scott", "Hyderabad");
        scott.getStudentData();
    }
}
-----
```

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 :

```

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

public class Manager
{
    int managerId;
    String managerName;
    double managerSalary;

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

    @Override
    public String toString()
    {
        return "Manager [managerId=" + this.managerId + ", managerName=" +
this.managerName + ", managerSalary=" + this.managerSalary
                + "]";
    }
}
```

```
    }
}

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

public class ManagerDemo
{
    public static void main(String[] args)
    {
        Manager smith = new Manager();
        smith.setManagerData(1, "Smith", 115000);

        System.out.println(smith); //calling toString() method of Manager
    }
}
```

#### Data Hiding :

It is a concept through which we can protect our data from outer world so no one can access our the data directly.

In order to achieve this we should declare our data with private access modifier.

Our data must be accessible via methods only so we can validate the outer world data for acceptance.

```
2 files :
-----
Customer.java
-----
package com.ravi.oop;

public class Customer
{
    private double balance = 10000;

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

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

```

package com.ravi.oop;

public class Customer
{
    private double balance = 10000;

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

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

```

#### BankApplication.java

```

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

public class BankApplication
{
    public static void main(String[] args)
    {
        Customer raj = new Customer();
        raj.deposit(5000);
        raj.withdraw(3000);

    }
}

```

So the conclusion is our data must not be visible directly from the Outer world so declare them private.

-----  
12-06-2024

-----  
Abstraction [Hiding the complexity]

-----  
Showing the essential details without showing the background details (Implementation Details) is called Abstraction.

In java, we can achieve abstraction by using following two techniques :

1) Abstract class and abstract method

(By using abstract class and abstract method we can achieve 0 to 100 % abstraction so, partial abstraction)

Example :

```
-----  
public abstract class Lift  
{  
    public abstract void keyOne();  
    public abstract void keyTwo();  
    public abstract void keyThree();  
    public abstract void keyFour();  
    public abstract void keyFive();  
  
    public void getLiftInformation()  
    {  
        System.out.println("NIT Lift");  
    }  
}
```

2) By using interface

(By using interface we can achieve 100% abstraction)

```
public interface ATMMachine  
{  
    void deposit();  
    void withdraw();  
    void pinChange();  
    void others();  
}
```

Constructor :

What is the advantage of writing constructor in our class ?

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

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

[Diagram 12-jun-24]

Note :- We should not use method to initialize our object properties, It is not a recommended way, instead of method we should use constructor so we will get two benefits :

- a) Variable Initialization and Variable re-initialization both are done in the same line at the time of creating the object
- b) Object properties are initialized at the time of object creation only so we will not get default values for instance variable.

The following program explains that we will get default values even we are writing our own constructor.

```
package com.nit.oop;  
  
public class Sample  
{  
    int x,y;  
  
    public Sample() //User written constructor  
    {  
        System.out.println(x); //0
```

```

        System.out.println(y); //0
    }

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

}
-----
```

Constructor key points :

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

The main purpose of constructor to initialize the object properties, nothing but instance variable of the class.

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

Whenever we create an object in java by using new keyword then at-least one constructor must be invoked.

A constructor never contain any return type including void also, if we put return type then it will become method as shown below.

```

package com.nit.oop;

public class Sample
{
    int x,y;

    public void Sample() //It is a method, We need to call explicitly
    {
        System.out.println(x);
        System.out.println(y);
    }

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

By default constructor never containing any return type but implicitly it returns current class obejct (this keyword) as shown below

```

package com.nit.oop;

public class Sample
{
    public void m1()
    {
        System.out.println("Non static method m1");
    }

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

```
//Nameless object OR Anonymous object  
new Sample().m1();  
}  
}
```

Inside a constructor we can use return keyword only but not return keyword with value.

```
package com.nit.oop;  
  
public class Sample  
{  
  
    public Sample()  
    {  
        System.out.println("User Constructor");  
        return ; //valid  
    }  
  
    public static void main(String[] args)  
    {  
        new Sample(); //Nameless OR Anonymous object  
    }  
}
```

A constructor is automatically called and executed at the time of creating the object. [We need not to call explicitly]

Everytime we create the object, constructor will be invoked.

```
package com.nit.oop;  
  
public class Sample  
{  
  
    public Sample()  
    {  
        System.out.println("User Constructor");  
        return ;  
    }  
  
    public static void main(String[] args)  
    {  
        new Sample(); //Nameless OR Anonymous object  
        new Sample();  
    }  
}
```

---

13-06-2024

-----  
Types of constructor in java :

-----  
Java supports 3 types of constructor :

- 1) Default constructor
- 2) No Argument OR Parameterless OR Non-parameterized Constructor
- 3) Parameterized Constructor

Default constructor :

The constructor which is added by compiler in our class is called as default constructor. The access modifier of default constructor depends upon class access modifier.

Example :

```
public class Test
{
}

javac Test.java (Compilation)

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

-----  
2) No Argument OR Parameter less OR non parameterized OR Zero Argument constructor

If a constructor is written by user without parameter then it is called no-argument constructor.

Example :

```
public class Student
{
    int rollNumber;

    public Student() //No Argument constructor
    {
        rollNumber = 111;
    }
}
```

Note : No argument and default constructor both are parameter-less constructor, the only difference is default constructor is added by compiler and no-argument is written by user.

By using no-argument constructor all the object properties will initialize with same value so, it is not a recommended way as shown in the program below.

2 files :

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

```
package com.ravi.constructor;

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

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

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

```

+ "]";
    }
}

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

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

        Person raj = new Person();
        System.out.println(raj);
    }
}

```

Note : In the above program both the objects, raj and scott are initializing with same value so it is not a recommended way hence we should use Parameterized constructor.

Parameterized Constructor :

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

It is used to initialized all the objects with different values.

Example :

```

-----
public class Student
{
    int rollNumber;

    public Student(int rollNumber) //Parameterized constructor
    {
        this.rollNumber = rollNumber;
    }
}

```

Program on parameterized constructor :

2 files :

Dog.java

```

-----
package com.ravi.constructor;

public class Dog
{
    private String dogName;
    private double dogHeight;
    private String dogColor;

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

```

```

        this.dogHeight = dogHeight;
        this.dogColor = dogColor;
    }

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

```

ParameterizedConstructor.java

```

-----
package com.ravi.constructor;

public class ParameterizedConstructor {

    public static void main(String[] args)
    {
        Dog tommy = new Dog("Tommy", 2.2, "Grey");
        System.out.println(tommy);

        Dog tiger = new Dog("Tiger", 3.5, "Black");
        System.out.println(tiger);
    }
}
-----
```

How to write and use setter and getter in java application ?

setter : Used to modify the existing object data.

getter : used to read private data value outside of the BLC class.

Example :

```

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

    //Writing setter for employeeId
    public void setEmployeeId(int employeeId)
    {
        this.employeeId = employeeId;
    }

    //Writing setter for employeeName
    public void setEmployeeName(string employeeName)
    {
        this.employeeName = employeeName;
    }

    //Writing getter for employeeId
    public int getEmployeeId()
    {
        return this.employeeId;
    }

    //Writing getter for employeeName

```

```

public String getEmployeeName()
{
    return this.employeeName;
}
-----
//Program on setter and getter

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

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

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

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

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

    public String getEmployeeName()
    {
        return this.employeeName;
    }

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

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

public class EmployeeDemo {

    public static void main(String[] args)
    {
        Employee raj = new Employee(111, "Raj", 40000);
        raj.setEmployeeSalary(50000); //setter to modify the data
        System.out.println(raj);

        double salary = raj.getEmployeeSalary();
    }
}

```

```
        String name = raj.getEmployeeName();

        if(salary >=50000)
        {
            System.out.println(name +" is a Developer!!!");
        }
        else if(salary >=35000)
        {
            System.out.println(name +" is a designer!!!");
        }
        else
        {
            System.out.println(name +" is a tester!!!");
        }
    }

}
```

so the final conclusion is,

Parameterized Constructor : Used to initialized the object.

setter : Used to modify existing object data

getter : Used to read private data outside of the BLC class.

-----  
14-06-2024

-----  
\*\*\*Encapsulation [private data must be accessible via methods only]

-----  
Binding the data member with its associated function/method in a single unit is called encapsulation.

In other words we can say "Grouping the related things together is called Encapsulation". [Laptop]

In encapsulation data must be tightly coupled with associated function.

It provides us security because we can't access the data directly, data must be accessible via methods only.

We can achieve encapsulation in our program by using following

- a) Declare all the data members as private (Tightly encapsulated class)
- b) Define getters and setters (Or any other kind of public method) for each instance variable to perform read and write operation.

Note :

-----  
If we declare all the instance variables with private access modifier then it is called tightly encapsulated class.

On the other hand If we declare our instance variable other than private access modifier then it is called loosely encapsulated class.

Program on Encapsulation :

-----  
2 files :

-----  
Employee.java

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

```
public class Employee
{
    private int employeeId;

    public void setEmployeeId(int employeeId)
    {
        if(employeeId<=0)
        {
            System.out.println("Invalid Employee ID ");
        }
        else
        {
            this.employeeId = employeeId;
        }
    }

    public int getEmployeeId()
    {
        return this.employeeId;
    }
}
```

EmployeeDemo.java

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

public class EmployeeDemo {

    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        e1.setEmployeeId(222);

        int empId = e1.getEmployeeId();
        System.out.println("Employee Id is :" + empId);

    }
}
```

-----  
How to use class name as a return type of the method ?

-----  
2 files :

-----  
Employee.java

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

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

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

```
public static Employee getEmployeeObject()
{
    Employee e1 = new Employee(111, "Scott", 50000);
    return e1;
}

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

}

EmployeeDemo.java

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

public class EmployeeDemo {

    public static void main(String[] args)
    {
        Employee object = Employee.getEmployeeObject();
        System.out.println(object);
    }
}
```

-----  
2 files :

-----  
Book.java

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

import java.util.Scanner;

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

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

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

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

```
        System.out.print("Enter book price :");
        double price = sc.nextDouble();

        return new Book(title, author, price);
    }

}
```

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

import java.util.Scanner;

public class BookApplication {

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("How many book objects ?");
        int bookObject = sc.nextInt();

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

}
```

Note :- From the above it is clear that from method we can return multiple objects based on the requirement.

-----  
17-06-2024

-----  
What is a factory Method ?

-----  
If a method return type is a class that means the method returns object then it is called Factory Method.

-----  
Pass by Value :

-----  
Java does not support pointeurs so if we pass any value to the method then java works with pass by value only, Here only the copy is passing top the method so it is known as Pass by value.

```
package com.ravi.pass_by_value;

public class Demo1 {

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

    public static void accept(int y)
    {
        y = 150;
    }
}
```

```
}
```

Here we are passing the primitive so the copy will pass and it will not reflect to the original value.

```
-----  
package com.ravi.pass_by_value;  
  
public class Demo2 {  
  
    public static void main(String[] args)  
    {  
        Integer i = 100;  
        accept(i);  
        System.out.println(i); //100  
    }  
  
    public static void accept(Integer j)  
    {  
        j = 200;  
    }  
}  
-----  
package com.ravi.pass_by_value;  
  
public class Demo3 {  
  
    public static void main(String[] args)  
    {  
        String str = "India";  
        accept(str);  
        System.out.println(str); //India  
    }  
    public static void accept(String x)  
    {  
        x = "Hyderabad";  
    }  
}
```

Note :- Integer and String both are immutable (un-changed) classes so there is no modification on the original object.

-----  
Working with Object reference :

-----  
2 files :

-----  
Employee.java

```
-----  
package com.ravi.pass_by_value;  
  
public class Employee  
{  
    private int employeeId = 100;  
  
    public int getEmployeeId() {  
        return employeeId;  
    }  
  
    public void setEmployeeId(int employeeId) {  
        this.employeeId = employeeId;  
    }  
}
```

```
}

Demo4.java
-----
package com.ravi.pass_by_value;

public class Demo4 {

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

        Employee e2 = e1;
        e2.setEmployeeId(500);

        System.out.println(e1.getEmployeeId()); //500
        System.out.println(e2.getEmployeeId()); //500

    }
}

-----
package com.ravi.pass_by_value;

class Customer
{
    double bill = 5000;
}

public class Demo5
{
    public static void main(String[] args)
    {
        Customer c1 = new Customer();
        accept(c1);
        System.out.println(c1.bill); //12000
    }

    public static void accept(Customer cust)
    {
        cust.bill = 12000;
    }
}

-----
package com.ravi.pass_by_value;

class Manager
{
    double salary = 40000;
}

public class Demo6 {

    public static void main(String[] args)
    {
        Manager m1 = new Manager();

        Manager m2 = m1;
```

```
m2.salary = 60000;  
  
System.out.println(m1.salary);  
System.out.println(m2.salary);  
  
}  
}
```

---

#### HEAP and STACK Diagram :

---

In java all the objects and its contents are stored in a very special memory called HEAP Memory.

All the methods (parameter and local variable) are stored in Stack memory.

What is Garbage Collector in java ?

---

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 GC :

---

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

1) Assigning null literal to reference variable :

```
Employee e1 = new Employee(111,"Ravi");  
e1 = null;
```

2) Creating an Object inside a method :

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

Here we are creating Employee object inside the method so, once the method execution is over then e2 will be deleted from the Stack Frame and the employee object will become eligible for GC.

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

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

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

---

18-06-2024

---

```

HEAP and STACK Diagram Programs :
-----
HEAP and STACK Diagram for Customer.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 3000x is eligible for GC]

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

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

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

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

//output 9 5
-----
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 are eligible 1000x, 2000x, 5000x and 6000x]
        System.out.println(s2.i1);
    }
public static Sample modify(Sample s)
{
    s.i1=9;
    s = new Sample();
    s.i1= 20;
    System.out.println(s.i1);
    s=null;
    return s;
}
}

//20 9
-----
```

HEAP and STACK 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;
        t3.t = t4;
        t1.t = t2.t;
        t2.t = t4.t;

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

-----  
19-06-2024

```

-----
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 objects 2000x and 4000x both are eligible for GC]

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

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

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

//Output 400 500 500 500
-----
class Test
{
int x;
int y;

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

```

```

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

t1.m1(t2);

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

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

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

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

20-06-2024

-----  
Passing an object reference to the Constructor :

-----  
We can pass an object reference to the constructor, the main purpose of passing an object reference to the constructor to copy the content of one object into another object.

```

3 files :
-----
Employee.java
-----
package com.ravi.copy_constructor;

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

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

    public int getEmployeeId() {
        return employeeId;
    }

    public String getEmployeeName() {
        return employeeName;
    }
}
```

}

Manager.java

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

public class Manager {
```

```
private int managerId;
private String managerName;

public Manager(Employee emp)
{
    managerId = emp.getEmployeeId();
    managerName = emp.getEmployeeName();
}

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

}
```

Main.java

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

public class Main {

    public static void main(String[] args)
    {
        Employee e1 = new Employee(111, "Scott");

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

Note : In the above program we are initializing the Manager object data by using Employee object data so, Employee object data we can use to initialize Manager class properties.

The following program describes how to copy the same object data to another object.

2 files :

-----
Player.java

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

public class Player
{
    private String name1, name2;

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

    public Player(Player p1) //1000x -> name1 = Rohit     name2 = Virat
    {
        this.name1 = p1.name2;
        this.name2 = p1.name1;
    }

    @Override
```

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

CopyConstructorDemo.java

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

public class CopyConstructorDemo
{
    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);
    }
}
```

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

A class called Customer is given to you.

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

Define the following for the class.

Attributes :

```
    customerName : String, private
    creditPoints: int, private
```

Constructor :

```
    parameterizedConstructor: for both cusotmerName & creditPoints in that
order.
```

Methods :

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

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

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

Attributes :

```
    customer : Customer, private
    cardType : String, private
```

Constructor :

```
    parameterizedConstructor: for customer and cardType attributes in that
order
```

Methods :

```
    Name of the method : toString Override this.
    Return type : String
    Modifier : public
    Task : Return the string in the following format.
```

The Customer 'Rajeev' Is Eligible For 'Gold' Card.

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

Method :

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

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

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

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

4 files :

-----

Customer.java

-----

```
package com.ravi.lab_program;
```

```
public class Customer
```

```
{
```

```
    private String customerName;  
    private int creditPoints;
```

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

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

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

```
}
```

CardType.java

-----

```
package com.ravi.lab_program;
```

```
public class CardType
```

```
{
```

```
    private Customer cust;  
    private String cardType;
```

```
    public CardType(Customer cust, String cardType)
```

```
{  
    super();  
    this.cust = cust;  
    this.cardType = cardType;  
}  
  
@Override  
public String toString()  
{  
    return "The Customer '" +this.cust+ "' Is Eligible For  
'" +this.cardType+ "' Card";  
}
```

```
}
```

```
CardsOnOffer.java
```

```
-----  
package com.ravi.lab_program;  
  
public class CardsOnOffer  
{  
    public static CardType getOfferedCard(Customer obj)  
    {  
        int creditPoint = obj.getCreditPoints();  
  
        if(creditPoint >=100 && creditPoint <=500)  
        {  
            return new CardType(obj, "Silver");  
        }  
        else if(creditPoint >500 && creditPoint <=1000)  
        {  
            return new CardType(obj, "Gold");  
        }  
  
        else if(creditPoint >1000)  
        {  
            return new CardType(obj, "Platinum");  
        }  
        else  
        {  
            return new CardType(obj, "EMI");  
        }  
    }  
}
```

```
ELC.java
```

```
-----  
package com.ravi.lab_program;  
  
public class ELC {  
  
    public static void main(String[] args)  
    {  
        Customer c1 = new Customer("Rajeev", 760);  
  
        CardType card = CardsOnOffer.getOfferedCard(c1);  
  
        System.out.println(card);  
    }  
}
```

21-06-2024

-----  
Modifier on constructor :

-----  
A constructor we can declare with all access modifiers (which is used as accessibility level) like private, default, protected and public.

```
public class Sample
{
    private int x,y;

    protected Sample(int x, int y) //We can declare constructor with
    {                               private, default, protected and
        public
    }

}

-----  
* What is the use of declaring a constructor with private access modifier?
```

We can declare a constructor with private access modifier due to the following 2 reasons :

- 1) If we want to declare only static variable and static method inside a class then we can declare the constructor as a private constructor.
- 2) If we want to develop singleton class that means we can create the object only one time by the same developer, Here outsiders are not allowed to create the object.

```
package com.ravi.copy_constructor;

public class Sample
{
    private int x,y;

    private Sample(int x, int y)
    {
        super();
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "Sample [x=" + x + ", y=" + y + "]";
    }

    public static void main(String[] args)
    {
        Sample s = new Sample(10, 20);
        System.out.println(s);
    }
}
```

Note : A constructor can't be declare as final and static.

-----  
-----

What is an instance block OR instance initializer ?

-----

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.[top to bottom]

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

We can't use return keyword with value inside non static block.

```
-----  
package com.ravi.instance_demo;  
  
class Sample  
{  
    public Sample()  
    {  
        System.out.println("No Argument Constructor!!!");  
    }  
  
    {  
        System.out.println("Instance OR Non static block");  
    }  
}  
  
public class InstanceBlockDemo1 {  
  
    public static void main(String[] args)  
    {  
        new Sample(); //Anonymous OR Nameless object  
        new Sample();  
    }  
}
```

Note : It is clear that instance block will be executed before constructor body execution

```
-----  
package com.ravi.instance_demo;  
  
class Test  
{  
    private int x;  
  
    public Test()  
}
```

```

{
    System.out.println("Inside constructor");
}

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

{
    x = 200;
    System.out.println(x);
}

{
    x = 300;
    System.out.println(x);
}
}

```

```

public class InstanceBlockDemo2
{
    public static void main(String[] args)
    {
        System.out.println("Main");
        new Test();
    }
}

```

Note :- Instance blocks are executed according to the order[Top to bottom]

---

```

package com.ravi.instance_demo;

class Demo
{
    {
        return 0; //return value not allowed inside
                  instance block
}

```

```

public class InstanceBlockDemo3 {

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

```

---

```

package com.ravi.instance_demo;

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

        {
            System.out.println("Instance Block");
        }
    }
}

```

```
        }
    }

public class InstanceBlockDemo3 {
    public static void main(String[] args)
    {
        new Demo();
    }
}
```

-----  
22-06-2024  
-----

Life cycle of instance variable initialization :

```
class Test
{
    int x = 100;

    public Test()
    {
        x = 300;
    }

    {
        x = 200;
    }
}
```

An instance variable, during its life cycle will be initialized in the following places according to the same order.

- 1) Instance variable will be initialized with default values in the Object class constructor using super keyword.
- 2) We will verify that instance variable is initialized at the time of declaration or not
- 3) We will verify instance variable is initialized at the time of execution of non static block or not
- 4) We will verify instance variable is initialized inside the constructor body or not
- 5) We can also initialize inside method body but It is not recommended because It is not the part of object creation and we need to call the method explicitly.

-----  
package com.ravi.instance\_variable;

```
public class Test
{
    int x = 10;

    {
        x = 20;
    }

    public Test()
    {
        x = 30;
    }
}
```

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

    System.out.println(t1.x);

}
```

From the above program it is clear that instance variable initialization order will be as follows :

default value(Object class) -> at the time of variable declaration -> At the time execution of non static block -> at the time execution of constructor body -> inside a method body [not recommended]

-----  
What is blank final field in java ?

-----  
If an instance final variable not initialized at the time of declaration then it is called blank final variable.

Example : final int A; //Blank final variable

\*A blank final variable must be initialized by the user till the execution of constructor body otherwise compilation error will be generated.

A blank final variable can't be initialized by default constructor, It must be initialized by user explicitly.

A user can initialized the blank final variable in the following 2 places only.

- 1) Inside a non static block OR
- 2) Inside the body of the constructor

A blank final variable, We can't be initialized by method.

A blank final variable can also have default values, initialized through object class.

A blank final variable must be initialized by the user in all the constructors available in the class.

-----  
//Program on blank final variable

```
class Test
{
    final int A;
}

public class BlankFinalField1
{
    public static void main(String[] args)
    {
        Test t1 = new Test();

        System.out.println(t1.A);
    }
}
```

Note : A blank final variable can't be initialized by default constructor.

```
-----  
class Demo  
{  
    final int A;  
  
    {  
        A = 100;  
    }  
}  
public class BlankFinalField2  
{  
    public static void main(String[] args)  
    {  
        Demo d1 = new Demo();  
        System.out.println(d1.A);  
    }  
}
```

A blank final variable can be initialized inside non static block.

```
-----  
class Demo  
{  
    final int A;  
  
    public Demo()  
    {  
        A = 900;  
    }  
}  
public class BlankFinalField3  
{  
    public static void main(String[] args)  
    {  
        Demo d1 = new Demo();  
        System.out.println(d1.A);  
    }  
}
```

A blank final variable can be initialized inside constructor body.

```
-----  
class Foo  
{  
    final int B;  
  
    {  
        m1();  
        B = 100;  
        System.out.println("User Value :" + B);  
    }  
  
    public void m1()  
    {  
        System.out.println("Default value :" + B);  
    }  
}  
  
public class BlankFinalField4  
{  
    public static void main(String[] args)  
    {  
        new Foo();  
    }  
}
```

Note : A blank final variable can also have default values.

```
-----  
class Sample  
{  
    final int D;  
  
    public Sample()  
    {  
        D = 100;  
    }  
  
    public Sample(int x)  
    {  
        D = x;  
    }  
}  
  
public class BlankFinalField5  
{  
    public static void main(String[] args)  
    {  
        Sample s1 = new Sample();  
        System.out.println(s1.D);  
  
        Sample s2 = new Sample(200);  
        System.out.println(s2.D);  
    }  
}
```

A blank final variable must be explicitly initialized by the user in all the constructor available in the class that means with each object creation final field must be initialized.

-----  
22-06-2024

-----  
Relationship between the classes :

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

- a) IS-A relation
- b) HAS-A relation

IS-A relation we can achieve by using Inheritance concept :

HAS-A relation we can achieve by using Association concept :

Example of IS-A relation :

```
-----  
class Vehicle  
{  
}  
class Car extends Vehicle  
{  
}
```

Here we have IS-A relation between Vehicle and Car because Car IS-A Vehicle

HAS-A relation :

```
-----  
class Engine  
{  
}
```

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

Here Car HAS-A Engine, because Car is having Engine.

24-06-2024

-----  
Inheritance (IS-A Relation) :

Deriving a new class (child class) from existing class (parent class) in such a way that the new class will acquire all the properties and features (except private) from the existing class is called inheritance.

It is one of the most 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 beginning 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 downward more specialized properties will occur.

-----  
Types of Inheritance :

Java supports 5 types of Inheritance

- 1) Single Level Inheritance
- 2) Multilevel Inheritance
- 3) Hierarchical Inheritance
- 4) Multiple Inheritance (Invalid)
- 5) Hybrid Inheritance (Combination of two)

-----  
25-06-2024

-----  
Program on Single level Inheritance :

-----  
SingleLevelDemo.java [Single File Approach]

```
package com.ravi.inheritance;

class Father
{
    public void house()
    {
        System.out.println("3 BHK House");
    }
}
class Son extends Father
```

```

{
    public void car()
    {
        System.out.println("Audi Car");
    }
}

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

```

Note : In inheritance, we should always create the object for the more specialized class so we can access all the super class properties from the specialized class object reference.

-----  
//Program on Single level inheritance :

-----  
3 files :

-----  
Super.java

-----  
package com.ravi.inheritance;

```

public class Super
{
    private int x,y;

    public int getX()
    {
        return x;
    }

    public void setX(int x)
    {
        this.x = x;
    }

    public int getY()
    {
        return y;
    }

    public void setY(int y)
    {
        this.y = y;
    }
}

```

}

-----  
Sub.java

```

package com.ravi.inheritance;

public class Sub extends Super
{

```

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

SingleInheritanceDemo.java

```
-----
package com.ravi.inheritance;

public class SingleInheritanceDemo {

    public static void main(String[] args)
    {
        Sub s = new Sub();
        s.setX(100); s.setY(200);
        s.displayData();
    }
}
-----
```

Initializing the super class properties :

In order to initialize super class properties through constructor we should use super keyword as shown below :

What is super keyword ?

super keyword is a technique through which we can access the properties of super class using super keyword.

super keyword always refers to the member of immediate super class.

Just like this keyword, we can't use super keyword inside the static area.

super keyword we can use 3 ways in java :

- ```
-----
1) To access the super class variable (Variable Hiding)
2) To access the super class method
3) To access the super class constructor
```

1) To access the super class variable :

Whenever super class variable name and sub class variable name both are same then it is called variable Hiding, Here sub class variable hides super class variable.

In order to access super class variable, we should use super keyword as shown in the program.

```
package com.ravi.super_demo;

class Father
{
    protected double balance = 50000;
}
class Son extends Father
{
    protected double balance = 18000; //Variable Hiding
```

```

        public void displayBalance()
    {
        System.out.println("Son balance is :" +balance);
        System.out.println("Father balance is :" +super.balance);
    }
}

public class SuperVar {
    public static void main(String[] args)
    {
        Son s = new Son();
        s.displayBalance();
    }
}

```

---

### 2) To access the super class method :

---

Whenever super class method name and sub class method name, both are same then sub class method hides super class method so, to accesss super class method we shoulud use super keyword.

Note :- In inheritance tree whenever we search a method, It will search from bottom to top. If method names are same then to access super class method we can use super keyword.

```

package com.ravi.super_demo;

class Alpha
{
    public void show()
    {
        System.out.println("Alpha class show method");
    }
}
class Beta extends Alpha
{
    public void show()
    {
        super.show();
        System.out.println("Beta class show method");
    }
}

public class SuperDemo {
    public static void main(String[] args)
    {
        Beta b = new Beta();
        b.show();
    }
}

```

---

26-06-2024

---

### 3) To access the super class constructor (Constructor Chaining) :

---

Whenever we write a class in java and we don't write any kind of constructor to the class then the java compiler will automatically add one default 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.

We have 4 cases :

-----  
super() : It is automatically added by compiler, Used to call the super class no argument or default constructor.

CallingConstructor.java

```
-----  
package com.ravi.super_ex;  
  
class Super  
{  
    public Super()  
    {  
        super();  
        System.out.println("No Argument constructor of Super class");  
    }  
}  
class Sub extends Super  
{  
    public Sub()  
    {  
        super();  
        System.out.println("No Argument constructor of Sub class");  
    }  
}  
  
public class CallingConstructor {  
    public static void main(String[] args)  
    {  
        Sub s = new Sub();  
    }  
}
```

-----  
WAP that shows default constructor and super keyword added by compiler in our class :

```
class Alpha  
{  
    public Alpha()  
    {  
        System.out.println("No Argument constructor of Alpha class");  
    }  
}  
class Beta extends Alpha  
{
```

```

}

class Gamma extends Beta
{
    public Gamma()
    {
        System.out.println("No Argument constructor of Gamma class");
    }
}

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

Case 2 :

```
super("NIT"); : It must be written by user, used to call parameterized
constructor of super class which accepts one argument
of String type parameter.
```

```

package com.ravi.super_ex;

class Alpha
{
    public Alpha(String message)
    {
        System.out.println("Parameterized Constructor :" + message);
    }
}
class Beta extends Alpha
{
    public Beta()
    {
        super("NIT");
        System.out.println("No Argument constructor :");
    }
}

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

Case 3 :

```
this() : Written by user, used to call no argument constructor of
current class / same class.
```

```

package com.ravi.super_ex;

class Parent
{
    public Parent()
    {
        super();
    }
}
```

```

        System.out.println("No Args of Parent class");
    }

    public Parent(String message)
    {
        this();
        System.out.println("Parameterized of Parent class :" + message);
    }
}
class Child extends Parent
{
    public Child(String msg)
    {
        super(msg);
        System.out.println("No Argument of Child class");
    }
}

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

```

Case 4 :

`this("NIT")` : Written by used, used to call parameterized constructor of current class/ Same class

```

package com.ravi.super_ex;

class Base
{
    public Base()
    {
        this(10);
        System.out.println("No Args of Base :");
    }

    public Base(int x)
    {
        super();
        System.out.println("Parameterized constructor :" + x);
    }
}
class Derived extends Base
{
    public Derived()
    {
        super();
        System.out.println("No Args of Derived class");
    }
}

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

```

```

    }

}

-----
Program on super keyword :
-----
package com.nit.super_demo;

class Shape
{
    protected int x;

    public Shape(int x)
    {
        this.x = x;
    }
}
class Square extends Shape
{
    public Square(int side)
    {
        super(side);
    }

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

public class SuperDemo {

    public static void main(String[] args)
    {
        Square ss = new Square(8);
        ss.getAreaOfSquare();
    }
}

-----
27-06-2024
-----
How to access super class non static variable with sub class static method using
variable hiding concept.

package com.ravi.inheritance;

class A
{
    protected int x = 100;
}
class B extends A
{
    protected int x = 200; //Variable hiding

    public static void show()
    {
        B b1 = new B();
        System.out.println(b1.x);
        A a1 = b1;
    }
}

```

```

        System.out.println(a1.x);
    }
}

public class SuperWithStatic {

    public static void main(String[] args)
    {
        B b1 = new B();
        B.show();
    }
}

-----//Program on Single level Inheritance :
-----
package com.ravi.inheritance;

class Employee
{
    protected int employeeId;
    protected String employeeName;
    protected double employeeSalary;

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

    @Override
    public String toString()
    {
        return "Employee [employeeId=" + employeeId + ", employeeName=" +
employeeName + ", employeeSalary="
                + employeeSalary + "]";
    }
}
class PermanentEmployee extends Employee
{
    protected String departmrnt;
    protected String designation;

    public PermanentEmployee(int employeeId, String employeeName, double
employeeSalary, String departmrnt, String designation) {
        super(employeeId, employeeName, employeeSalary);
        this.departmrnt = departmrnt;
        this.designation = designation;
    }

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

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

```

```

    }
    PermanentEmployee p = new PermanentEmployee(1, "Scott", 120000, "IT",
"Programmer");
    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()
    {
        x = 100;  y = 200;
    }
}

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

Here also, all the objects will be initialized with same value.

B) Second Approach (Method with Parameter)

```
-----  
public class Test  
{  
    int x,y;  
  
    public void setData(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}  
  
Test t1 = new Test();  t1.setData(12,78);  [x = 12  y = 78]  
Test t2 = new Test();  t2.setData(15,29);  [x = 15  y = 29]
```

Here the Drawback is initialization and re-initialization both are done in two different lines so Constructor introduced.

4) By using Constructor

A) First Approach (No Argument Constructor)

```
-----  
public class Test  
{  
    int x,y;  
  
    public Test() //All the objects will be initialized with  
    {                                same value  
        x = 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();

```

---

28-06-2024

---

//Program on Hierarchical Inheritance :

---

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

```
import java.util.Scanner;
```

```

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

}
class Circle extends Shape
{
    protected final double PI = 3.14;

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

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

}
class Rectangle extends Shape
{
    protected int breadth;
    public Rectangle(int length, int breadth) //10 20
    {
        super(length);
        this.breadth = breadth;
    }

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

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

        new Circle(radius).areaOfCircle();

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

```

```
        new Rectangle(length, breadth).areaOfRectangle();  
  
    }  
}  
-----
```

Why java does not support multiple inheritance ?

Multiple Inheritance is a situation where a sub class wants to inherit the properties two or more super classes and by using super keyword we have ambiguity issue. It is also known as Diamond Problem.

Java does not support multiple inheritance because in the sub class, automatically one super keyword will be added by compiler to the first line of constructor now we have ambiguity issue to call default constructor of super class as shown in the diagram (28-JUNE-24)

Multiple inheritance we can't achieve using classes but same is possible by using interface.

01-07-2024

-----  
Program on hierarchical Inheritance:

```
-----  
package com.ravi.inheritance_demo;  
  
class Employee  
{  
    protected double salary;  
  
    public Employee(double salary)  
    {  
        super();  
        this.salary = salary;  
    }  
}  
class Developer extends Employee  
{  
    public Developer(double sal)  
    {  
        super(sal);  
    }  
  
    @Override  
    public String toString()  
    {  
        return "Developer [salary=" + salary + "]";  
    }  
}  
class Designer extends Employee  
{  
    public Designer(double sal)  
    {  
        super(sal);  
    }  
  
    @Override  
    public String toString()  
    {  
        return "Designer [salary=" + salary + "]";  
    }  
}
```

```

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

        Designer des = new Designer(25000);
        System.out.println(des);
    }
}

-----//Program on Multilevel Inheritance :-----
-----package com.ravi.inheritance_demo;

class Student
{
    private int studentNumber;
    private String studentName;
    private String studentAddress;

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

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

class Science extends Student
{
    protected int physics;
    protected int chemistry;

    public Science(int studentNumber, String studentName, String
studentAddress, int physics, int chemistry) {
        super(studentNumber, studentName, studentAddress);
        this.physics = physics;
        this.chemistry = chemistry;
    }

    @Override
    public String toString() {
        return super.toString()+"Science [physics=" + physics + ",
chemistry=" + chemistry + "]";
    }
}

```

```

}
class PCM extends Science
{
    protected int math;

    public PCM(int studentNumber, String studentName, String studentAddress,
int physics, int chemistry, int math)
    {
        super(studentNumber, studentName, studentAddress, physics,
chemistry);
        this.math = math;
    }

    @Override
    public String toString() {
        return super.toString()+"PCM [math=" + math + "]";
    }

    public void calculateTotalMarks()
    {
        double marks = this.physics + this.chemistry + this.math;
        System.out.println("Total Marks is :" +marks);
    }
}

public class MultilevelInheritance
{
    public static void main(String[] args)
    {
        PCM p = new PCM(1, "Raj", "Ameerpet", 78, 70, 90);
        System.out.println(p);
        p.calculateTotalMarks();
    }
}
-----
```

Access modifiers in java :

In order to define the accessibility level of the class as well as member of the class we have 4 access modifiers :

- 1) private (Within the same class)
- 2) default (Within the same package)
- 3) protected (Within the same package Or even from another package by using Inheritance)
- 4) public (No Restriction)

private :

-----  
It is the most restrictive access modifier because the member declared as private can't be accessible from outside of the class.

In Java we can't declare an outer class as a private or protected. Generally we should declare the data member(variables) as private.

In java outer class can be declared as public, abstract, final, sealed and non-sealed only.

default :-

-----  
It is an access modifier which is less restrictive than private. It is such kind of access modifier whose physical existance is not avaialble that means when we don't specify any kind of access modifier before the class name, variable name

or method name then by default it would be default.

As far as its accessibility is concerned, default members are accessible within the same folder(package) only. It is also known as private-package modifier.

protected :

-----  
It is an access modifier which is less restrictive than default because the member declared as protected can be accessible from the outside of the package (folder) too but by using inheritance concept.

This program contains two files in two different packages :

-----  
Access.java(com.nit.m1)

-----  
package com.nit.m1;

public class Access  
{  
 protected int x = 888;  
}

Main.java

-----  
package com.nit.m2;

import com.nit.m1.Access;  
  
public class Main extends Access  
{  
 public static void main(String[] args)  
 {  
 Main m1 = new Main();  
 System.out.println(m1.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.

Note : If a method is used for internal purpose only (like validation) then we can declare that method as private method. It is called Helper method.

-----  
02-07-2024

-----  
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 studentId;
    private String studentName;
    private double totalMarks;
    private String studentAddress;

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

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

    public int getStudentId()
    {
        return studentId;
    }
}
```

Trainer.java  
-----

```
package com.ravi.association;

import java.util.Scanner;

public class Trainer
{
    public static void viewStudentProfile(Student student)
```

```

{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter Student id :");
    int id = sc.nextInt();

    if(id == student.getStudentId())
    {
        System.out.println(student);
    }
    else
    {
        System.err.println("Sorry!! Student is not available with this
id!!!");
    }
    sc.close();
}

}

```

**CompositionDemo.java**

```

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

public class CompositionDemo {

    public static void main(String[] args)
    {
        Student s1 = new Student(1, "Scott", 420, "S R Nagar");

        Student s2 = new Student(2, "Smith", 410, "Ameerpet");

        Trainer.viewStudentProfile(s2);
    }
}
-----
```

**Composition (Strong reference) :**

Composition in Java is a way to design classes such that one class contains an object of another class. It is a way of establishing a "HAS-A" relationship between classes.

Composition represents a strong relationship between the containing class and the contained class. If the containing object (Car object) is destroyed, all the contained objects (Engine object) are also destroyed.

A car has an engine. Composition makes strong relationship between the objects. It means that if we destroy the owner object, its members will be also destroyed with it. For example, if the Car is destroyed the engine will also be destroyed as well.

**3 files :**

-----  
**Engine.java**

```

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

public class Engine
{
    private String engineType;
    private int horsePower;

    public Engine(String engineType, int horsePower)
    {
        super();

```

```

        this.engineType = engineType;
        this.horsePower = horsePower;
    }

    @Override
    public String toString() {
        return "Engine [engineType=" + engineType + ", horsePower=" +
horsePower + "]";
    }
}

Car.java
-----
package com.ravi.composition;

public class Car
{
    private String carName;
    private String carModel;
    private final Engine engine; //HAS-A relation [Blank final variable]

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

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

```

```

CompositionDemo.java
-----
package com.ravi.composition;

public class CompositionDemo {

    public static void main(String[] args)
    {
        Car car = new Car("Naxon", "EV2024");
        System.out.println(car);
    }
}

```

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 :

College.java

```

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

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

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

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

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

public class Student
{
    private int studentId;
    private String studentName;
    private String studentAddress;
    private College college; // HAS-A relation

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

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

AggregationDemo.java
-----
package com.ravi.aggregation;

public class AggregationDemo {

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

```

```

        Student s1 = new Student(1, "A", "Ampt", clg);
        System.out.println(s1);

        Student s2 = new Student(2, "B", "S R Nagar", clg);
        System.out.println(s2);

    }

}

```

Note :- IS-A relation is tightly coupled relation so if we modify the content of super class, sub class content will also modify but in HAS-A relation we are accessing the properties of another class so we are not allowed to modify the content, we can access the content or Properties.

-----  
Assignments :

Organization and Employee  
Person and Address  
Laptop and Motherboard  
University and Department

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

```

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

```

System.out.println();

Internally System.out.println() creates HAS-A relation because System class contains a predefined class called java.io.PrintStream as shown in the above example.

The following program describes that how System.out.println() works internally :

```

package com.nit.m1;

class Hello
{
    public static String out = "Hyderabad"; //HAS-A Relation
}

public class HasADemo {

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

```

-----  
\*\*\*Polymorphism :

Poly means "many" and morphism means "forms".

It is a Greek word whose meaning is "same object having different behavior".

In our real life a person or a human being can perform so many tasks, in the same way in our programming languages a method or a constructor can perform so many tasks.

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)
```

---

03-07-2024

---

Polymorphism can be divided into two types :

- 1) Static polymorphism OR Compile time polymorphism OR Early binding
  - 2) Dynamic Polymorphism OR Runtime polymorphism OR Late binding
- 

1) Static Polymorphism :

---

The polymorphism which exist at the time of compilation is called Static OR compile time polymorphism.

In static polymorphism, compiler has very good idea that which method is invoked depending upon METHOD PARAMETER.

Here the binding of the method is done at compilation time so, it is known as early binding.

We can achieve static polymorphism by using Method Overloading concept.

Example of static polymorphism : Method Overloading.

2) Dynamic Polymorphism 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.

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 method/static method ?

Yes, we can overload the main method OR static method but the execution of the program will start from main method which accept String [] array as a parameter.

Note :- The advantage of method overloading is same method name we can reuse for different functionality for refinement of the method.

Note :- In System.out.println() or System.out.print(), print()  
and println() methods are best example for Method Overloading.  
-----

//Program on Constructor Overloading :

-----  
2 files :

-----  
Addition.java

-----  
//Program on Constructor Overloading  
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)
    {
        super();
        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);

    }
}

```

-----  
instance block role with constructor :

-----  
Our instance block will be placed in the constructor which contains super(), if we have multiple constructor and all the constructors are executed according to constructor chaining then compiler will placed instance block to only one constructor which contains super()

2 files :

-----  
Addition.java

```

-----
package com.ravi.constructor_overloading2;

public class Addition
{
    public Addition()
    {
        this(100);
        System.out.println("No Argument Constructor");
    }

    public Addition(int x)
    {
        this(1000,2000);
        System.out.println("One Argument Constructor :" +x);
    }

    public Addition(int x, int y)
    {
        super();

```

```
        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();

    }
-----
```

04-07-2024

```
-----
program that describes we can change the return type of the method at the time
of method overloading :
```

2 files :

```
-----
Sum.java
-----
package com.ravi.method_overload;

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

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

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

}
```

Main.java

```
-----
package com.ravi.method_overload;

import java.text.DecimalFormat;

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

```
        Sum s = new Sum();
        System.out.println("Sum of two Integer is :" + s.add(12, 24));
        System.out.println("Concatenation of two String is :" + s.add("Data",
"Base"));
        double add = s.add(1.2, 2.4);
        DecimalFormat df = new DecimalFormat("00.00");
        System.out.println("Sum of two double is :" + df.format(add));
    }
}
-----
```

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.

The program says var args can accept 0 to n number of parameters.

2 files :

Test.java

```
package com.ravi.var_args;
```

```
public class Test
{
    public void accept(int ...x)
    {
        System.out.println("Var Args executed");
    }
}
```

Main.java

```
package com.ravi.var_args;
```

```
public class Main
{
    public static void main(String ...args)
    {

        Test t1 = new Test();
        t1.accept();
        t1.accept(12);
        t1.accept(12, 89);
        t1.accept(12, 78, 56);
    }
}
```

//Program to add sum of parameters passes inside a method using var args

2 files :

```

Test.java
-----
package com.ravi.var_args1;

public class Test
{
    public void sumOfParameters(int... values)
    {
        int sum = 0;

        for(int value : values)
        {
            sum = sum + value;
        }
        System.out.println("Sum of parameter is :" + sum);
    }
}

Main.Java
-----
package com.ravi.var_args1;

public class Main
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.sumOfParameters(12, 12, 12);
        t1.sumOfParameters(100, 200, 300);

    }
}

//Program that describes that 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)
    {
        new Test().accept(12, 10, 20, 30, 40);
    }
}
-----
//Program to show that var args can hold heterogeneous 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);
        }
    }
}
```

```
Main.java
-----
```

```
package com.ravi.var_args3;

public class Main {

    public static void main(String[] args)
    {
        new Test().acceptHetro(true, 45.90, 12, 'A', new String("Ravi"));
    }
}
-----
```

Wrapper classes in java :

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.

Note : ALL THE WRAPPER CLASSES ARE FINAL, IMMUTABLE (Unchanged), THREAD-SAFE AND OVERRIDDEN equals(Object obj) and hashCode() method.

From java 1.5 onwards we can convert primitive to wrapper and wrapper to primitive by using following concept.

- 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        |

How to convert primitive to wrapper object ?

In order to convert the primitive data type into corresponding object each wrapper class has provided valueOf() static method.

```
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);
    }
}
-----
package com.ravi.autoboxing;

public class AutoboxingDemo2
{
    public static void main(String[] args)
    {
        byte b = 12;
        Byte b1 = Byte.valueOf(b);
        System.out.println("Byte Object :" + b1);

        short s = 17;
        Short s1 = Short.valueOf(s);
        System.out.println("Short Object :" + s1);

        int i = 90;
        Integer i1 = Integer.valueOf(i);
        System.out.println("Integer Object :" + i1);

        long g = 12;
        Long h = Long.valueOf(g);
        System.out.println("Long Object :" + h);

        float f1 = 2.4f;
        Float f2 = Float.valueOf(f1);
        System.out.println("Float Object :" + f2);
    }
}
```

```

        double k = 90.90;
        Double l = Double.valueOf(k);
        System.out.println("Double Object :" + l);

        char ch = 'A';
        Character ch1 = Character.valueOf(ch);
        System.out.println("Character Object :" + ch1);

        boolean x = true;
        Boolean x1 = Boolean.valueOf(x);
        System.out.println("Boolean Object :" + x1);

    }
}

```

In the above program we have used 1.4 approach so we are converting primitive to wrapper object manually.

-----  
05-07-2024  
-----

Overloaded valueOf() method :

- 1) public static Integer valueOf(int x) : It will convert the given int value into Integer Object.
- 2) public static Integer valueOf(String str) : It will convert the given String into Integer Object.  
[valueOf() method will convert the String into Wrapper object where as parseInt() method will convert the String into primitive type]
- 3) public static Integer valueOf(String str, int radix/base) : It will convert the given String number into Integer object by using the radix or base.

Note :- We can pass base OR radix upto 36  
i.e A to Z (26) + 0 to 9 (10) -> [26 + 10 = 36], It can be calculated by using Character.MAX\_RADIX.  
Output will be generated on the basis of radix

```

//Integer.valueOf(String str)
//Integer.valueOf(String str, int radix/base)
public class AutoBoxing3
{
    public static void main(String[] args)
    {
        Integer a = Integer.valueOf(15);

        Integer b = Integer.valueOf("25");

        Integer c = Integer.valueOf("111", 36); //Here Base we can take upto
36

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);

    }
}

public class AutoBoxing4
{

```

```

public static void main(String[] args)
{
    Integer i1 = new Integer(100);
    Integer i2 = new Integer(100);
    System.out.println(i1==i2);

    Integer a1 = Integer.valueOf(15);
    Integer a2 = Integer.valueOf(15);
    System.out.println(a1==a2);
}

```

`==` operator always compares the memory address so it is returning false (two object are created using `new` keyword), on the other hand `valueOf()` return the same object so it will provide true.

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

```

`Integer` class has a predefined static method `toString(int x)`, which will convert `int` to `String` type.

-----  
`Unboxing :`

-----  
`Converting wrapper object to corresponding primitive type is called Unboxing.`

|                   |                   |
|-------------------|-------------------|
| Wrapper<br>Object | Primitive<br>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 `Number` class which represent numbers (either decimal OR non decimal)  
 so all the following six wrapper classes (which are sub class of `Number` class) are providing the following common methods.

- 1) `public byte byteValue()`
- 2) `public short shortValue()`

```

3) public int intValue()
4) public long longValue()
5) public float floatValue()
6) public double doubleValue()
-----
//Converting Wrapper object into primitive
public class AutoUnboxing1
{
    public static void main(String args[])
    {
        Integer obj = 15; //Upto 1.4
        int x = obj.intValue();
        System.out.println(x);
    }
}
-----
public class AutoUnboxing2
{
    public static void main(String[] args)
    {
        Integer x = 25;
        int y = x; //JDK 1.5 onwards
        System.out.println(y);
    }
}
-----
public class AutoUnboxing3
{
    public static void main(String[] args)
    {
        Integer i = 15;
        System.out.println(i.byteValue());
        System.out.println(i.shortValue());
        System.out.println(i.intValue());
        System.out.println(i.longValue());
        System.out.println(i.floatValue());
        System.out.println(i.doubleValue());
    }
}
-----
public class AutoUnboxing4
{
    public static void main(String[] args)
    {
        Character c1 = 'A';
        char ch = c1.charValue();
        System.out.println(ch);
    }
}
-----
public class AutoUnboxing5
{
    public static void main(String[] args)
    {
        Boolean b1 = true;
        boolean b = b1.booleanValue();
        System.out.println(b);
    }
}
-----
Unlike primitive types we can't convert one wrapper type object to another

```

wrapper object.

Example :

```
Long l = 12; //Invalid
Float f = 90; //Invalid
Double d = 123; //Invalid

package com.ravi.basic;

public class Conversion
{
    public static void main(String[] args)
    {
        long l = 12; //Implicit OR Widening
        byte b = (byte) 12L; //Explicit OR Narrowing

        Long a = 12L;
        Double d = 90D;
        Double d1 = 90.78;
        Float f = 12F;

    }
}
```

-----  
06-07-2024

Ambiguity issue while overloading a method :

When we overload a method then compiler is selecting appropriate method among the available methods based on the parameter type.

In order to solve the ambiguity issue while overloading a method compiler has probided the following rules :

1) Most Specific Type :

Compiler alwyas provide more priority to most specific data type or class type.

```
double > float [Here float is the most specific type]
float > long
long > int
int > char
int > short
short > byte
```

2) WAV [Widening -> Autoboxing -> Var Args]

Compiler gives the priority to select appropriate method by using the following sequence :  
Widening ---> Autoboxing ----> Var args

3) Nearest Data type or Nearest class (sub class)

While selecting the appropriate method in ambiguity issue compiler provides priority to nearest data type or nearest class i.e sub class

-----  
class Test

```

{
    public void accept(double d)
    {
        System.out.println("double");
    }
    public void accept(float d)
    {
        System.out.println("float");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(6);
    }
}

```

Here float will be executed because float is the nearest type.

```

-----
class Test
{
    public void accept(int d)
    {
        System.out.println("int");
    }
    public void accept(char d)
    {
        System.out.println("char");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        t.accept(6);
    }
}

```

Here int will be executed because int is the nearest type.

```

-----
class Test
{
    public void accept(int ...d)
    {
        System.out.println("int");
    }
    public void accept(char ...d)
    {
        System.out.println("char");
    }
}
public class AmbiguityIssue {

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

```
}
```

Here char will be executed because char is the most specific type

```
-----  
class Test  
{  
    public void accept(short ...d)  
    {  
        System.out.println("short");  
    }  
    public void accept(char ...d)  
    {  
        System.out.println("char");  
    }  
}  
public class AmbiguityIssue {  
  
    public static void main(String[] args)  
    {  
        Test t = new Test();  
        t.accept();  
    }  
}
```

Here we will get compilation error because there is no relation between char and short based on the specific type rule.

```
-----  
class Test  
{  
    public void accept(short ...d)  
    {  
        System.out.println("short");  
    }  
    public void accept(byte ...d)  
    {  
        System.out.println("byte");  
    }  
}  
public class AmbiguityIssue {  
  
    public static void main(String[] args)  
    {  
        Test t = new Test();  
        t.accept();  
    }  
}
```

Here byte will be executed because byte is the specific type.

```
-----  
class Test  
{  
    public void accept(double ...d)  
    {  
        System.out.println("double");  
    }  
    public void accept(long ...d)  
    {  
        System.out.println("long");  
    }  
}  
public class AmbiguityIssue {
```

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

}

}
```

Here long will be executed because long is the most specific type.

```
-----  
class Test
{
    public void accept(byte d)
    {
        System.out.println("byte");
    }
    public void accept(short s)
    {
        System.out.println("short");
    }
}
public class AmbiguityIssue {

    public static void main(String[] args)
    {
        Test t = new Test();
        //t.accept(9); //error
        t.accept((short)9);

    }
}
```

Here value 9 is of type int so, we can't assign directly to byte and short, If we want explicit type casting is reqd.

```
-----  
class Test
{
    public void accept(int d)
    {
        System.out.println("int");
    }
    public void accept(long s)
    {
        System.out.println("long");
    }
}
public class AmbiguityIssue {

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

    }
}
```

Note : Here int will be executed because int is the nearest type

```
-----  
class Test
{
    public void accept(int d, long l)
    {
        System.out.println("int-long");
    }
}
```

```
    }
    public void accept(long s, int i)
    {
        System.out.println("long-int");
    }
}
public class AmbiguityIssue {

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

    }
}
```

Here We will get ambiguity issue.

---

```
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
}
public class AmbiguityIssue {

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

    }
}
```

Here Object will be executed

---

```
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
}
public class AmbiguityIssue {

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

    }
}
```

```
Here String will be executed
-----
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
}
public class AmbiguityIssue {

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

    }
}
```

String will executed because String is the nearest type.

```
-----
class Test
{
    public void accept(Object s)
    {
        System.out.println("Object");
    }
    public void accept(String s)
    {
        System.out.println("String");
    }
    public void accept(Integer i)
    {
        System.out.println("Integer");
    }
}
public class AmbiguityIssue {

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

    }
}
```

Here We will get compilation error

```
-----
class Alpha
{
}
class Beta extends Alpha
{
}
class Test
{
    public void accept(Alpha s)
    {
```

```

        System.out.println("Alpha");
    }
    public void accept(Beta i)
    {
        System.out.println("Beta");
    }
}
public class AmbiguityIssue {

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

```

Here Beta will be executed.

---

```

class Test
{
    public void accept(Number s)
    {
        System.out.println("Number");
    }
    public void accept(Integer i)
    {
        System.out.println("Integer");
    }
}
public class AmbiguityIssue {

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

```

Here Integer will be executed.

---

```

class Test
{
    public void accept(long s)
    {
        System.out.println("Widening");
    }
    public void accept(Integer i)
    {
        System.out.println("Autoboxing");
    }
}
public class AmbiguityIssue {

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

```

Here widening is having more priority

```
-----  
class Test  
{  
    public void accept(int ...s)  
    {  
        System.out.println("Var args");  
    }  
    public void accept(Integer i)  
    {  
        System.out.println("Autoboxing");  
    }  
}  
public class AmbiguityIssue {  
  
    public static void main(String[] args)  
    {  
        Test t = new Test();  
        t.accept(12);  
    }  
}
```

Here Autoboxing will be executed.

\*\*\*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 and Downcasting :

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 Lion(); //valid [upcasting]

Downcasting :

By default we can't assign super class object to sub class reference variable.

Lion l = new Animal(); //Invalid

Even if we type cast Animal to Lion type then compiler will allow but at runtime JVM will not convert Animal object (Generic type) into Lion object (Specific type) and it will throw an exception java.lang.ClassCastException

Lion l = (Lion) new Animal(); //At runtime we will get  
java.lang.ClassCastException

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

-----  
08-07-2024  
-----

```
class Animal
{
    public void eat()
    {
        System.out.println("Genric Animal is eating...");
    }
}
class Dog extends Animal
{
    public void eat()
    {
        System.out.println("Dog is eating...");
    }
}
class Horse extends Animal
{
    public void eat()
    {
        System.out.println("Horse is eating...");
    }
}
public class MethodOverridingDemo1
{
    public static void main(String[] args)
    {
        Animal a = new Dog(); //Upcasting
        a.eat();

        Animal a1 = new Horse();
        a1.eat();

    }
}
```

-----  
package com.ravi.overriding;

```
class Bird
{
    public void roam()
    {
        System.out.println("Generic Bird is roaming");
    }
}
class Parrot extends Bird
{
    public void roam()
    {
        System.out.println("Parrot Bird is roaming");
    }
}
class Peacock extends Bird
{
    public void roam()
    {
        System.out.println("Peacock Bird is roaming");
    }
}
public class MethodOverridingDemo2
{
```

```

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

    b = new Parrot(); b.roam(); //Dynamic Method Dispatched
    b = new Peacock(); b.roam(); //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 metadata (Giving information that method is overridden) and it is optional but it is always a good practice to write @Override annotation before the Overridden method so compiler as well as user will get the confirmation that the method is overridden method and it is available in the super class.

If we use @Override annotation before the name of the overridden method in the sub class and if the method is not available in the super class then it will generate a compilation error so it is different from comments because comment will not generate any kind of compilation error if method is not an overridden method, so this is how it is different from comment.

```

package com.ravi.overriding;

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

public class MethodOverridingDemo3 {

    public static void main(String[] args)
    {
        Shape s = null;

        s = new Rectangle(); s.draw();
        s = new Square(); s.draw();

    }
}
```

```

}

-----
package com.ravi.overriding;

class Alpha
{
    @Override
    public String toString()
    {
        return "Alpha []";
    }
}

class Beta extends Alpha
{
    @Override
    public String toString()
    {
        return "Beta []";
    }
}

}

public class MethodOverridingDemo4 {

    public static void main(String[] args)
    {
        Alpha a = new Beta();
        System.out.println(a);

        Beta b = new Beta();
        System.out.println(b.toString());
    }
}

-----
Variable Hiding concept in upcasting :
-----
package com.ravi.overriding;

class Animal
{
    String name = "Generic Animal";

    public String roam()
    {
        return "Generic Animal is roaming";
    }
}
class Lion extends Animal
{
    String name = "Lion Animal"; //Variable Hiding

    @Override
    public String roam()
    {
        return "Lion Animal is roaming";
    }
}

public class VariableHidingDemo
{

```

```

public static void main(String[] args)
{
    Animal a = new Lion();
    System.out.println(a.name + " : "+a.roam());
}

```

Note : Here we will get the output Generic Animal : Lion Animal is roaming because in java variables are hidden not overridden and behavior can be changed but not the property.

-----  
09-07-2024  
-----

Can we override private method ?

No, We can't override private method because private methods are not visible (not available) to the sub class hence we can't override.

We can't use @Override annotation before overridden method which is declared private in the super class as shown in the program.

```

package com.ravi.poly;

class Super
{
    private void m1() //It can't be overridden
    {
        System.out.println("Private Method of super class");
    }
}
class Sub extends Super
{
    public void m1() //Re-declaring m1 method in the sub class
    {
        System.out.println("Private method of sub class");
    }
}

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

```

Note :- private method of super class is not available or not inherited in the sub class so if the class declare the method with same signature then it is not overridden method, actually it is re-declared in the sub class.

-----  
Role of access modifier while overriding a method :

While overriding the method from super class, the access modifier of sub class method must be greater or equal in comparison to access modifier of super class method otherwise we will get compilation error.

In terms of accessibility, public is greater than protected, protected is greater than default (public > protected > default)  
[default < protected < public]

So the conclusion is we can't reduce the visibility of the method while

overriding a method.

Note :- private method is not available (visible) in sub class so it is not the part of method overriding.

```
-----  
class RBI  
{  
    public void loan()  
    {  
        System.out.println("Bank Should provide loan");  
    }  
}  
class ICICI extends RBI  
{  
    @Override  
    public void loan()  
    {  
        System.out.println("ICICI provides loan @ 9.2%");  
    }  
}  
class HDFC extends RBI  
{  
    @Override  
    protected void loan() //error can't reduce the visibility  
    {  
        System.out.println("HDFC provides loan @ 9.4%");  
    }  
}  
  
public class VisibilityLevel  
{  
    public static void main(String[] args)  
    {  
        RBI r = null;  
  
        r = new ICICI(); r.loan();  
        r = new HDFC(); r.loan();  
    }  
}
```

---

Co-variant 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.

CoVariantDemo.java

```
-----  
class Super  
{  
    public void show()  
    {  
    }  
}  
class Sub extends Super  
{  
    @Override  
    public int show()  
    {  
        return 0;  
    }  
}
```

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

Note : Here we will get compilation error that return type int 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 relationship so it is compatible) called Co-Variant as shown in the program below.

Note :- Co-variant will not work with primitive data type, it will work only with classes.

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

class Super
{
    public Alpha 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 CoVariantDemo
{
    public static void main(String[] args)
    {
        Super s = new Sub();
        s.show();
    }
}

-----  
package com.ravi.poly;

class Vehicle
{
    public Vehicle run()
    {
        System.out.println("Generic Vehicle");
        return this;
    }
}
```

```

class Car extends Vehicle
{
    public Car run()
    {
        System.out.println("Car is Running");
        return this;
    }
}
public class CoVariantDemo {

    public static void main(String[] args)
    {
        Vehicle v = new Car();
        v.run();
    }
}

```

In the the return value, we can also use this keyword.

---

```

package com.ravi.poly;

class Alpha
{
    public Object show()
    {
        System.out.println("Show method of super class");
        return this;
    }
}
class Beta extends Alpha
{
    public Integer show()
    {
        System.out.println("Show method of Sub class");
        return null;
    }
}

```

```

public class CoVariantEx {

    public static void main(String[] args)
    {
        Alpha a = new Beta();
        a.show();
    }
}

```

---

While working with CoVariant(In the same direction sub class can vary)  
the return type of the method while calling the method must be super class  
object but not sub class object.

```

package com.ravi.poly;

class Base
{
    public Object m1()
    {
        System.out.println("m1 method of base class");
        return new Object();
    }
}

```

```

class Derived extends Base
{
    public String m1()
    {
        System.out.println("m1 method of Derived class");
        return new String();
    }
}

public class CoVariantExample
{
    public static void main(String[] args)
    {
        Base b = new Derived();
        Object obj = b.m1(); //Here return type is super type i.e.Object
        type, If we want to store the method return type in String then compilation
        error
    }
}
-----
```

10-07-2024

What is Method Hiding in java ?

OR

Can we override static method ?

OR

Can we override main method ?

While working with method hiding we have all different cases :

Case 1 :

A public static method of super class by default available to sub class so, from sub class we can call super class static method with the help of Class name as well as object reference as shown in the below program

```

package com.ravi.method_hiding;

class Super
{
    public static void m1()
    {
        System.out.println("static method of super class");
    }
}
class Sub extends Super
{
```

}

```

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

        Sub s1 = new Sub();
        s1.m1();
    }
}
```

-----

Case 2 :

-----  
We can't override a static method with non static method because static method belongs to class and non static method belongs to object, If we try to override static method with non static method then it will generate an error i.e overridden method is static as shown below.

```
class Base
{
    public static void m1()
    {
    }
}
class Derived extends Base
{
    public void m1() //error
    {
    }
}
public class StaticDemo1
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

-----

Case 3 :

-----  
We can't override any non static method with static method, If we try then it will generate an error, Overriding method is static.

```
class Base
{
    public void m1()
    {
    }
}
class Derived extends Base
{
    public static void m1()
    {
    }
}
public class StaticDemo2
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

So, the conclusion is we cannot overide static with non static method as well as non-static with static method because static method belongs to class and non-static method belongs to object.

-----

Case 4 :

-----  
Program that describes method hiding and sub class method can't hide super class method

```
class Base
{
    public static void m1()
    {
```

```

        System.out.println("Static method of super class..");
    }
}
class Derived extends Base
{
    public static int m1() //sub class method can't hide super
                           class method

        System.out.println("Static method of sub class..");
        return 0 ;
    }
}
public class StaticDemo1
{
    public static void main(String[] args)
    {
        Base b = new Derived();
        b.m1();
    }
}
-----
```

We can't override static method, If we write static method in the sub class with same signature and same return type then It is Method Hiding but not Method Overriding here compiler will search the method of super class and JVM will also execute the method of super class because method is not overridden.

Note :- We can't apply @Override annotation on static methods.

```

class Base
{
    public static void m1()
    {
        System.out.println("Static method of super class..");
    }
}
class Derived extends Base
{
    public static void m1() \\Method Hiding
    {
        System.out.println("Static method of sub class..");
    }
}
public class StaticDemo1
{
    public static void main(String[] args)
    {
        Base b = new Derived();
        b.m1();
    }
}
-----
```

Note : static variable, instance variable and static method are not overridden so we will get super class output.

```

package com.ravi.method_hiding;

class Super
{
    int a = 1000;
    static int x = 100;
    public static void m1()
    {
```

```

        System.out.println("static method of super class");
    }
}
class Sub extends Super
{
    int a = 2000;
    static int x = 200;
    public static void m1()
    {
        System.out.println("static method of sub class");
    }
}

```

```

public class MethodHidingDemo
{
    public static void main(String[] args)
    {
        Super s = new Sub();
        System.out.println("Instance Variable :" + s.a);
        System.out.println("Class Variable :" + s.x);
        s.m1();
    }
}
-----
```

Program that describes Polymorphic behaviour of sub classes :

```

-----
package com.ravi.poly;

class Animal
{
    public void sleep()
    {
        System.out.println("Generic Animal is sleeping..");
    }
}
class Dog extends Animal
{
    @Override
    public void sleep()
    {
        System.out.println("Dog is sleeping..");
    }
}
class Lion extends Animal
{
    @Override
    public void sleep()
    {
        System.out.println("Lion is sleeping..");
    }
}

public class PolymorphicBehavior {

    public static void main(String[] args)
    {
        Animal a1 = new Dog();
        animalSleep(a1);

        Animal a2 = new Lion();
        animalSleep(a2);
    }
}
```

```
        public static void animalSleep(Animal animal)
        {
            animal.sleep();
        }
    }
```

Note : From above program it is clear that based on the runtime object appropriate method is invoked.

---

```
-----  
package com.ravi.poly;  
  
class Animal  
{  
    public void roam()  
    {  
        System.out.println("generic Animal is roaming...");  
    }  
}  
class Lion extends Animal  
{  
    @Override  
    public void roam()  
    {  
        System.out.println("Lion Animal is roaming...");  
    }  
  
    public void roar()  
    {  
        System.out.println("Lion is roaring....");  
    }  
}  
class Dog extends Animal  
{  
    @Override  
    public void roam()  
    {  
        System.out.println("Dog Animal is roaming...");  
    }  
  
    public void bark()  
    {  
        System.out.println("Dog is barking...");  
    }  
}
```

```
public class PolymorphicBehaviorDemo {  
  
    public static void main(String[] args)  
    {  
        Animal a = new Lion();  
        animalBehavior(a);  
  
        Animal a1 = new Dog();  
        animalBehavior(a1);  
    }  
  
    public static void animalBehavior(Animal animal)  
    {  
        Lion lion = (Lion) animal;  
        lion.roam();  
        lion.roar();  
    }  
}
```

```
}
```

```
}
```

In the above program we will get `java.lang.ClassCastException` because here we are trying to convert `Dog` object into `Lion` object.

-----  
instanceof operator :

-----  
It is an operator as well as keyword.

This operator returns true\false.

The main purpose of instanceof operator to verify whether a reference variable holds a particular type of object or not.

If the reference variable holds same type of object then it will return true otherwise it is will return false.

We must have IS-A relation in between reference variable and class\interface type.

It is used to avoid `ClassCastException` in java.

```
package com.ravi.poly;

class Test
{
}

class Gamma
{
}

class Shape
{
}

class Square extends Shape
{
}
class Circle extends Shape
{
}

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

        if(t1 instanceof Test)
        {
            System.out.println("t1 is pointing to Test object");
        }

        Object obj = new Gamma();
```

```
        if(obj instanceof Gamma)
        {
            System.out.println("It is pointing to Gamma object");
        }

        Shape s1 = new Square();

        if(s1 instanceof Circle)
        {
            System.out.println("s1 holds circle object");
        }
        else
        {
            System.out.println("s1 does not hold circle object");
        }

    }

-----
package com.ravi.poly;

class Animal
{
    public void roam()
    {
        System.out.println("generic Animal is roaming...");
    }
}
class Lion extends Animal
{
    @Override
    public void roam()
    {
        System.out.println("Lion Animal is roaming...");
    }

    public void roar()
    {
        System.out.println("Lion is roaring....");
    }
}
class Dog extends Animal
{
    @Override
    public void roam()
    {
        System.out.println("Dog Animal is roaming...");
    }

    public void bark()
    {
        System.out.println("Dog is barking...");
    }
}
class Tiger extends Animal
{
    @Override
    public void roam()
    {
        System.out.println("Tiger Animal is roaming...");
    }
}
```

```

public void roar()
{
    System.out.println("Tiger is roaring....");
}
}

public class PolymorphicBehaviorDemo {

    public static void main(String[] args)
    {
        Animal a = new Lion();
        a.roam();
        animalBehavior(a);

        Animal a1 = new Dog();
        a1.roam();
        animalBehavior(a1);
    }

    public static void animalBehavior(Animal animal)
    {
        if(animal instanceof Lion)
        {
            Lion lion = (Lion) animal;
            lion.roar();
        }

        if(animal instanceof Dog)
        {
            Dog dog = (Dog) animal;
            dog.bark();
        }

        if(animal instanceof Tiger)
        {
            Tiger tiger = (Tiger) animal;
            tiger.roam();
            tiger.roar();
        }
    }
}

```

-----  
11-07-2024  
-----

**final keyword in java :**

-----  
It is used to provide some kind of restriction in our program.  
We can use final keyword in ways 3 ways in java.

- 1) To declare a class as a final. (Inheritance is not possible)
- 2) To declare a method as a final (Overriding is not possible)
- 3) To declare a variable (Field) as a final (Re-assignment is not possible)

**1) To declare a class as a final :**

-----  
Whenever we declare a class as a final class then we 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();
    }
}
```

Here A class is final so, we can't inherit class A hence we will get compilation error.

```
-----
final class Test
{
    private int data = 100;

    public Test(int data)
    {
        this.data = data;
        System.out.println("Data value is :" + data);
    }
}
public class FinalClassEx1
{
    public static void main(String[] args)
    {
        Test t1 = new Test(200);

    }
}
```

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.

SealedDemo1.Demo

```
-----
package com.ravi.sealed_demo;

sealed class Bird permits Parrot, Sparrow
{
    public void fly()
    {
        System.out.println("Generic Bird is flying");
    }
}
non-sealed class Parrot extends Bird
{
    @Override
    public void fly()
    {
        System.out.println("Parrot Bird is flying");
    }
}
final class Sparrow extends Bird
{
    @Override
    public void fly()
    {
        System.out.println("Sparrow Bird is flying");
    }
}

public class SealedDemo1 {

    public static void main(String[] args)
    {
        Bird b = null;
        b = new Parrot(); b.fly();
        b = new Sparrow(); b.fly();
    }
}
-----
```

```

package com.ravi.sealed_demo;

sealed class OnlineClass permits Mobile,Laptop
{
    public void attendOnlineJavaClass()
    {
        System.out.println("Online java class!!!");
    }
}
final class Mobile extends OnlineClass
{
    @Override
    public void attendOnlineJavaClass()
    {
        System.out.println("Attending Java class through mobile.");
    }
}
final class Laptop extends OnlineClass
{
    @Override
    public void attendOnlineJavaClass()
    {
        System.out.println("Attending Java class through Laptop.");
    }
}
public class SealedDemo2
{
    public static void main(String[] args)
    {
        OnlineClass c = null;
        c = new Mobile(); c.attendOnlineJavaClass();
        c = new Laptop(); c.attendOnlineJavaClass();
    }
}

```

-----  
2) To declare a method as a final (Overriding is not possible)

-----  
Whenever we declare a method as a final then we can't override that method in the sub class otherwise there will be a compilation error.

We should declare a method as a final if the body of the method i.e the implementation of the method is very important and we don't want to override or change the super class method body by sub class method body then we should declare the super class method as final method.

```

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 Alpha
{
    private final void accept()
    {
        System.out.println("Alpha class accept method");
    }
}
class Beta extends Alpha
{
    protected void accept() //Re-Declaration
    {
        System.out.println("Beta class accept method");
    }
}
public class FinalMethodEx1
{
    public static void main(String [] args)
    {
        new Beta().accept();
    }
}

```

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; //error [Re-Assignment is not possible]
        System.out.println("A value is :" + A);
    }
}

class FinalVarEx
{
    public static void main(String[] args)
    {
        A a1 = new A();
        a1.setData();
    }
}

class FinalVarEx1
{
    public static void main(String[] args)
    {
        final int A = 127;
        byte b = A;
        System.out.println(b);
    }
}

12-07-2024
-----
What is Method Chaining in java ?
-----
It is a technique through which we can call various methods in a single statement.

In order to call next method we depend upon current method return type.

It is basically used to write concise coding.

The final return type of the method is based on last method call.[12-JULY]
-----
package com.ravi.method_chaining;

public class MethodChainingDemo
{
    public static void main(String[] args)
    {
        String str = "india";
        char ch = str.concat(" is great").toUpperCase().charAt(0);
        System.out.println(ch);
    }
}

package com.ravi.method_chaining;

public class MethodChainingDemo2 {

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

Object class and its Method :
-----
```

Object is the super class of all the classes we have in java.

Object class is available with all the sub classes so, methods of object class will be available to all the sub classes hence a sub class can override Object class method (Except final method).

Object class plays a major role in Object creation because whenever we create an object in java, the control must reach to object class first using super().

Object class has so many method through which we can perform various operation on Object.

- 1) Object(){} : Used to initialize all the instance variables with default values.
- 2) public java.lang.Class getClass() : Provides Runtime object of a class.[Name of the class]
- 3) public int hashCode() : Used to provide hashCode of the object through which we can find out bucket location.
- 4) public String toString() : Used to return the Object properties in String representation
- 5) public boolean equals(Object obj) : Used to compare two objects based on the memory reference.

Note : In Object class we have one no argument constructor and 11 methods

- 1) 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.

We can't override this method because it is a final method.

-----  
Here we have 2 java classes both are in different package.

Demo.java [com.nit.m1]

-----  
package com.nit.m1;

public class Demo {

}

Main.java[com.nit.m2]

-----  
package com.nit.m2;

import com.nit.m1.Demo;

public class Main {

    public static void main(String[] args)  
    {

```

        Demo d = new Demo();
        System.out.println(d.getClass());
    }
}

//class com.nit.m1.Demo [FQN = Fully Qualified Name]
-----
getClass() method return type is java.lang.Class so, further we can apply
any method of java.lang.Class. [Method Chaining]

java.lang.Class provides a predefined method called getName() through which we
can get the Fully Qualified Name of the class

```

```

public native final Class getClass();
public String getName();

```

```

package com.ravi.object_class;

class Test
{
}

public class GetClassDemo {

    public static void main(String[] args)
    {
        Test t1 = new Test();
        String name = t1.getClass().getName();
        System.out.println(name); //FQN
    }
}
-----
```

```

public int hashCode() :
-----
It is used to provide hashcode value of an object.

```

It is not meant for comparison of two objects, If two objects are same as per equals(Object obj) method then hashCode of both the object must be same.

```

package com.ravi.object_class;

class Sample
{
}

public class HshCodeDemo1
{
    public static void main(String[] args)
    {
        Sample s1 = new Sample();
        Sample s2 = new Sample();

        System.out.println(s1==s2); //false
        System.out.println(s1.equals(s2)); //false

        System.out.println(s1.hashCode() +" : "+s2.hashCode());
        System.out.println(".....");

        Sample s3 = new Sample();
        Sample s4 = s3;
    }
}
```

```
        System.out.println(s3.equals(s4)); //true
        System.out.println(s3.hashCode() +" : "+s4.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.

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 //com.ravi.object_class.Foo@
{

}

public class ToStringDemo2
{
    public static void main(String[] args)
    {
        Foo f1 = new Foo();
        System.out.println(f1.toString());

        Foo f2 = new Foo();
        System.out.println(f2);

    }
}

-----
package com.ravi.object_class;

class Demo
{
    @Override
    public String toString() {
        return "Demo []";
    }
}

public class ToStringDemo
```

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

-----  
13-07-2024  
-----

```
public boolean equals(Object obj) :  
-----  
It is a predefined method of Object class.
```

It is used to compare two objects based on the memory reference OR  
memory address.

EqualsDemo1.java

```
-----  
package com.ravi.equals_demo;  
  
class Employee  
{  
    private int employeeId;  
    private String employeeName;  
  
    public Employee(int employeeId, String employeeName)  
    {  
        super();  
        this.employeeId = employeeId;  
        this.employeeName = employeeName;  
    }  
  
}  
  
public class EqualsDemo1 {  
  
    public static void main(String[] args)  
    {  
        Employee e1 = new Employee(111, "Scott");  
        Employee e2 = new Employee(111, "Scott");  
  
        System.out.println(e1==e2); //false  
        System.out.println(e1.equals(e2)); //false  
  
    }  
}
```

Note : In the above program we will get the output false, false because  
equals(Object obj) method of Object class, internally uses ==  
operator only which is meant for memory reference comparison.

-----  
In the above program while comparing both the object equals(Object obj)  
method is providing false but the content of both the object are same  
so, we can override Object class equals(Object obj) method for content  
comparison.

How to override equals(Object obj) method for content comparison :

```
-----  
package com.ravi.equals_demo;
```

```

class Employee
{
    private int employeeId;
    private String employeeName;

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

//Overriding Object class equals(Object obj) method for content comparison

@Override
public boolean equals(Object obj) //obj = e2
{
    //Fetching the 1st Object data
    int eid1 = this.employeeId;
    String ename1 = this.employeeName;

    //Fetching the 2nd Object data
    Employee e2 = (Employee) obj;
    int eid2 = e2.employeeId;
    String ename2 = e2.employeeName;

    if(eid1==eid2 && ename1.equalsIgnoreCase(ename2))
    {
        return true;
    }
    else
    {
        return false;
    }
}

}

public class EqualsDemo1 {

    public static void main(String[] args)
    {
        Employee e1 = new Employee(111, "Scott");
        Employee e2 = new Employee(111, "scott");

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

-----  

Overriding equals(Object obj) method for content comparsion in Customer class  

-----
package com.ravi.equals_demo;

class Customer
{
    private int customerId;
    private String customerName;

    public Customer(int customerId, String customerName)
    {

```

```

        super();
        this.customerId = customerId;
        this.customerName = customerName;
    }

    @Override
    public boolean equals(Object obj)
    {
        Customer c2 = (Customer) obj;

        if(this.customerId == c2.customerId &&
this.customerName.equals(c2.customerName))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

public class EqualsDemo2 {

    public static void main(String[] args)
    {
        Customer c1 = new Customer(111, "Raj");
        Customer c2 = new Customer(111, "Raj");

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

-----
Overriding equals(Object obj) method for Student content comparison :
-----
In the equals(Object obj) method, If we pass two different types of object
which are not related to each other then we will get
java.lang.ClassCastException, To avoid this we can use instanceof operator
as shown in the program below :

package com.ravi.equals_demo;

class Student
{
    private int studentId;
    private String studentName;

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

    @Override
    public boolean equals(Object obj)
    {
        if(obj instanceof Student)
        {
            Student s2 = (Student) obj;

```

```

        if(this.studentId == s2.studentId &&
this.studentName.equals(s2.studentName))
        {
            return true;
        }
        else
        {
            return false;
        }

    }
else
{
    System.err.println("Comparison is not possible");
    return false;
}
}

public class EqualsDemo3
{
    public static void main(String[] args)
    {
        Student s1 = new Student(111, "Smith");
        Student s2 = new Student(111, "Smith");
        Player p1 = new Player(111,"Smith");

        System.out.println(s1.equals(s2));
        System.out.println(".....");
        System.out.println(s1.equals(p1));
        System.out.println(".....");
        System.out.println(s1.equals(null));

    }
}

class Player
{
    private int playerId;
    private String playerName;

    public Player(int playerId, String playerName)
    {
        super();
        this.playerId = playerId;
        this.playerName = playerName;
    }
}

```

Note : instanceof null will always return false.

---

15-07-2024

---

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 (Plain Old Java Object) classes.

It is mainly used to concise our code as well as remove the boiler plate code.

In record, automatically constructor will be generated which is known as canonical constructor and the variables which are known as components are by default final.

In order to validate the outer world data, we can write our own constructor which is known as compact constructor.

Record will automatically generate the implementation of `toString()`, `equals(Object obj)` and `hashCode()` method.

We can define static and non static method as well as static variable and static block inside the record. We cannot define instance variable and instance block inside the record.

We can't extend or inherit records because by default every record is implicitly final and It is extending from `java.lang.Record` class, which is an abstract class.

We can implement an interface by using record.

We don't have setter facility in record because by default components are final.

-----  
3 files :

-----  
`EmployeeClass.java(C)`  
-----  
package com.ravi.record\_demo;  
  
import java.util.Objects;  
  
public class EmployeeClass  
{  
 private int employeeId;  
 private String employeeName;  
  
 public EmployeeClass(int employeeId, String employeeName)  
 {  
 super();  
  
 this.employeeId = employeeId;  
 this.employeeName = employeeName;  
 }  
  
 public int getEmployeeId() {  
 return employeeId;  
 }  
  
 public void setEmployeeId(int employeeId) {  
 this.employeeId = employeeId;  
 }  
  
 public String getEmployeeName() {  
 return employeeName;  
 }  
}

```

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

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

        @Override
        public int hashCode() {
            return Objects.hash(employeeId, employeeName);
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            EmployeeClass other = (EmployeeClass) obj;
            return employeeId == other.employeeId &&
Objects.equals(employeeName, other.employeeName);
        }
    }
}

```

EmployeeRecord.java(R)

```

-----
package com.ravi.record_demo;

//Canonical Constructor [Components are final]
public record EmployeeRecord(int employeeId, String employeeName)
{
    //Compact Constructor
    public EmployeeRecord
    {
        if(employeeId<=0)
        {
            System.err.println("Invalid Employee Id :");
        }
    }
}

```

Main.java(C)

```

-----
package com.ravi.record_demo;

public class Main
{
    public static void main(String[] args)
    {
        Record r = null;

        EmployeeClass cls1 = new EmployeeClass(111, "Ravi");
        EmployeeClass cls2 = new EmployeeClass(111, "Ravi");
        System.out.println(cls1.equals(cls2));
        System.out.println(cls1);
        System.out.println(cls1.getEmployeeName());

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

```

```
EmployeeRecord rd1 = new EmployeeRecord(222, "Smith");
EmployeeRecord rd2 = new EmployeeRecord(222, "Smith");
System.out.println(rd1.equals(rd2));
System.out.println(rd1);
System.out.println(rd1.employeeName());
}
}
```

JVM Architecture with class loader sub system :

The entire JVM Architecture is divided into 3 section :

- 1) Class Loader sub system
- 2) Runtime Data areas (Memory Areas)
- 3) Execution Engine

Class Loader Sub System :

The main purpose of Class Loader sub system to load the required .class file into JVM Memory.

In order to load the .class file into JVM Memory, we use an algorithm called Delegation Hierarchy Algorithm.

Internally, Class Loader sub system performs the following Task

- 1) LOADING
- 2) LINKING
- 3) INITIALIZATION

LOADING :

In order to load the required .class file, JVM makes a request to class loader sub system. The class loader sub system follows delegation hierarchy algorithm to load the required .class files from different areas.

To load the required .class file we have 3 different kinds of class loaders.

- 1) Bootstrap/Primordial class loader
- 2) Extension/Platform class loader
- 3) Application/System class loader

16-07-2024

Bootstrap/Primordial class Loader :-

It is responsible for loading all the predefined .class files that means all API level predefined classes are loaded by Bootstrap class loader.

It has the highest priority because Bootstrap class loader is the super class for Platform class loader.

It loads the classes from the following path  
C -> Program files -> Java -> JDK -> lib -> jrt-fs.jar

Platform/Extension class loader :

It is responsible to load the required .class file which is given by some 3rd party in the form of jar file.

It is the sub class of Bootstrap class loader and super class of Application class loader so it has more priority than Application class loader.

It loads the required .class file from the following path.  
C -> Program files -> Java -> JDK -> lib -> ext -> ThirdParty.jar

command to create the jar file :  
jar cf FileName.jar FileName.class

[If we want to compile more than one java file at a time then the command is :  
javac \*.java]

Application/System class loader :

-----  
It is responsible to load all userdefined .class file into JVM memory.

It has the lowest priority because it is the sub class Platform class loader.

It loads the .class file from class path level or environment variable.

-----  
How Delegation Hierarchy algorithm works :-

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

Bootstrap class loader will load the .class file from lib folder(jrt.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 .class file into JVM memory then we will get a Runtime exception i.e java.lang.ClassNotFoundException.

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

-----  
WAP in java that shows whenever class is loaded into JVM memory then it returns java.lang.Class class object.

```
package com.ravi.jvm_architecture;

class Employee{}
class Student{}
class Sample{}

public class LoadingClassReturnType
{
    public static void main(String[] args)
    {
        Class cls = Employee.class;
        System.out.println(cls.getName());

        cls = Student.class;
        System.out.println(cls.getName());

        cls = Sample.class;
```

```

        System.out.println(cls.getName());
    }

}

-----
Program that describes Application class loader is responsible to
load the user-defined .class file into JVM Memory.

package com.ravi.jvm_architecture;

class Foo
{
    //public ClassLoader getClassLoader()
}

public class ClassLoderInfo {

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

    }
}

```

Note : getClassLoader() is a predefined method of java.lang.Class class whose return type is ClassLoader class which is an abstract class so the method signature is :

```
    public ClassLoader getClassLoader();
```

-----
Program that describes Platform class loader is the super class for
Application class loader.

```

package com.ravi.jvm_architecture;

class Foo
{

}

public class ClassLoderInfo {

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

Note : ClassLoader class has provided a predefined method called getParent() to get the name of the super class, the return type of this method is again ClassLoader class.

```
    public ClassLoader getParent();
```

-----
package com.ravi.jvm\_architecture;

class Foo
{

```
}

public class ClassLoderInfo {

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

Here we will not get Bootstrap class Loader, instead of that we will get null because Bootstap class loader is built in class loader for JVM and it is represented by null. It does not have any parent class.

-----  
17-07-2024

-----  
Linking :

-----  
verify :-

-----  
It ensures the correctness of the .class files, If any suspicious activity is there in the .class file then It will stop the execution immediately by throwing a runtime error i.e java.lang.VerifyError.

There is something called ByteCodeVerifier(Component of JVM), responsible to verify the loaded .class file i.e byte code. Due to this verify module JAVA is highly secure language.

java.lang.VerifyError is the sub class of java.lang.linkageError

-----  
prepare :

-----  
[Static variable memory allocation + static variable initialization with default value]

It will allocate the memory for all the static data members, here all the static data member will get the default values so if we have static int x = 100; then for variable x memory will be allocated (4 bytes) and now it will initialize with default value i.e 0, even the variable is final.

static Test t = new Test();

Here, t is a static reference variable so for t variable (reference variable) memory will be allocated as per JVM implementation i.e for 32 bit JVM (4 bytes of Memory) and for 64 bit (8 bytes of memory).

-----  
Resolve :

-----  
All the symbolic references will be converted into direct references Or actual reference.

-----  
javap -verbose FileName.class

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

-----  
Initialization :

-----  
Here class initialization will take place. All the static data member will get their actual value and we can also use static block for static data member initialization.

Here, In this class initialization phase static variable and static block is having same priority so it will executed according to the order.(Top to bottom)

---

Static Block in java :

---

It is a special block in java which is automatically executed at the time of loading the .class file.

Example :

```
static
{
}
```

Static blocks are executed only once because in java we can load the .class files only once.

If we have more than one static block in a class then it will be executed according to the order [Top to bottom]

The main purpose of static block to initialize the static data member of the class so it is also known as static initializer.

In java, a class is not loaded automatically, it is loaded based on the user request so static block will not be executed everytime, It depends upon whether class is loaded or not.

static blocks are executed before the main or any static method.

A static blank final field must be initialized inside the static block only.

```
static final int A; //static blank final field

static
{
    A = 100;
}
```

A static blank final field also have default value.

We can't write any kind of return statement inside static block.

If we don't declare static variable before static block body execution then we can perform write operation(Initialization is possible) but read operation is not possible directly otherwise we will get an error Illegal forward reference, It is possible with class name bacause now compiler knows that it is coming from class area OR Method area.

---

18-07-2024

---

```
//static block
class Foo
{
    Foo()
    {
        System.out.println("No Argument constructor..");
    }

    {
        System.out.println("Instance block..");
    }
}
```

```

    }

    static
    {
        System.out.println("Static block...");
    }

}

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

```

Here Foo.class file is not loaded into JVM Memory so static block of Foo class will not be loaded.

---

```

class Test
{
    static int x;

    static
    {
        x = 100;
        System.out.println("x value is :" + x);
    }

    static
    {
        x = 200;
        System.out.println("x value is :" + x);
    }

    static
    {
        x = 300;
        System.out.println("x value is :" + x);
    }
}

public class StaticBlockDemo1
{
    public static void main(String[] args)
    {
        System.out.println("Main Method");
        System.out.println(Test.x);
    }
}

```

Note : If a class contains more than 1 static block then it will be executed from top to bottom.

---

```

class Foo
{
    static int x;

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

```

```
}
```

```
public class StaticBlockDemo2
{
    public static void main(String[] args)
    {
        new Foo();
    }
}
```

Note : static variables are also having default value.

```
-----
```

```
class Demo
{

    final static int a ;      //Blank static final field

    static
    {
        a = 100;
        System.out.println(a);
    }
}
```

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

Note : A blank static final field must be initialized through static block only.

```
-----
```

```
class A
{
    static
    {
        System.out.println("A");
    }

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

    A()
    {
        System.out.println("C");
    }
}
```

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

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

    B()
    {
```

```

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

}

public class StaticBlockDemo4
{
    public static void main(String[] args)
    {
        new B();
    }
}

```

Note : Always Parent class will be loaded first then only Child class will be loaded.

---

//illegal forward reference

```

class Demo
{
    static
    {
        i = 100;
    }

    static int i;
}

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

```

Note : For static variable i, already memory is allocated in the prepare phase so we can initialize (can perform write operation) in the static block without pre-declaration.

---

```

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

    static int i;
}

```

```

public class StaticBlockDemo6
{

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

```

Note : Without declaring the static variable if we try to access static variable

```
value in the static block directly then we will get compilation error, we can  
access with the help of class name (Class Area)
```

```
-----  
class StaticBlockDemo7  
{  
    static  
    {  
        System.out.println("Static Block");  
        return;  
    }  
  
    public static void main(String[] args)  
    {  
        System.out.println("Main Method");  
    }  
}
```

Note : We can't write return statement in static and non static block

```
-----  
public class StaticBlockDemo8  
{  
    final static int x; //Blank static final field  
  
    static  
    {  
        m1();  
        x = 15;  
    }  
  
    public static void m1()  
    {  
        System.out.println("Default value of x is :" + x);  
    }  
  
    public static void main(String[] args)  
    {  
        System.out.println("After initialization :" + StaticBlockDemo8.x);  
    }  
}
```

A blank static final field also has default value.

```
-----  
class Test  
{  
    public static final Test t1 = new Test(); //t1 = null  
  
    static  
    {  
        System.out.println("static block");  
    }  
  
    {  
        System.out.println("Non static block");  
    }  
  
    Test()  
    {  
        super();  
        System.out.println("No Argument Constructor");  
    }  
}  
  
public class StaticBlockDemo9  
{
```

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

#### GOLDEN LINES :

- 1) INSTANCE VARIABLE INITIALIZATION FLOW
- 2) HOW MANY WAYS WE CAN INITIALIZE INSTANCE VARIABLE
- 3) HOW STATIC VARIABLE MEMORY ALLOCATED AND INITIALIZED

#### Variable Memory Allocation and Initialization :

##### 1) static field OR Class variable :

Memory allocation done at prepare phase of class loading and initialized with default value even variable is final.

It will initialized with Original value (If provided by user at the time of declaration) at class initialization phase.

When JVM will shutdown then during the shutdown phase class will be un-loaded so static data members are destroyed. They have long life.

##### 2) Non static field OR Instance variable

Memory allocation done at the time of object creation using new keyword (Instantiation) and initialized as a part of Constructor with default values even the variable is final. [Object class-> at the time of declaration -> instance block -> constructor]

When object is eligible for GC then object is destroyed and all the non static data memebers are also destroyed with corresponding object. It has lower life in comparison to static data members becuase they belongs to object.

##### 3) Local Variable

Memory allocation done at stack area (Stack Frame) and developer is responsible to initialize the variable before use. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

##### 4) Parameter variable

Memory allocation done at stack area (Stack Frame) and end user is responsible to pass the value at runtime. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

#### Can we write a Java Program without main method ?

```
class WithoutMain
{
    static
    {
        System.out.println("Hello User!!!");
        System.exit(0);
    }
}
```

It was possible to write a java program without main method till JDK 1.6V. From JDK 1.7v onwards, at the time of loading the .class file JVM will verify the presence of main method in the .class file. If main method is not available then it will generate a runtime error that "main method not found in so so class".

-----  
How many ways we can load the .class file into JVM memory :  
-----

There are so many ways to load the .class file into JVM memory but the following are the common examples

- 1) By using java command

```
public class Test
{
}
```

```
javac Test.java
java Test
```

Here we are making a request to class loader sub system to load Test.class file into JVM memory

- 2) By using Constructor (new keyword at the time of creating object).

- 3) By accessing static data member of the class.

- 4) By using inheritance

- 5) By using Reflection API

-----  
//Program that describes we can load a .class file by using new keyword (Object creation) OR by accessing static data member of the class.

```
class Demo
{
    static int x = 10;
    static
    {
        System.out.println("Static Block of Demo class Executed!!! :" +x);
    }
}
public class ClassLoading
{
    public static void main(String[] args)
    {
        System.out.println("Main Method");
        //new Demo();
        System.out.println(Demo.x);
    }
}
```

-----  
//Program that describes whenever we try to load sub class, first of all super class will be loaded. [before parent child can't exist]

```
class Alpha
{
    static
    {
        System.out.println("Static Block of super class Alpha!!!");
    }
}
class Beta extends Alpha
{
    static
    {
        System.out.println("Static Block of Sub class Beta!!!");
    }
}
```

```

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

-----
Loading the .class file by using Reflection API :
-----
java.lang.Class class has provided a predefined static method called
forName(String className), by using this forName(String className) we can load
the .class file into JVM memory dynamically.

This forName(String className) method return type is java.lang.Class
Class cls = Class.forName(String className); //factory Method

It throws a checked Exception ClassNotFoundException.

package com.ravi.jvm_arch;

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

public class DynamicLoading
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.ravi.jvm_arch.Foo");
    }
}

```

Note : By using Class.forName(String className) we can load the .class file at runtime dynamically.

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.jvm_arch.Foo");
```

Here Foo.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.

\*What is the limitation of 'new' keyword ?

OR

What is the difference between new keyword and newInstance() method?

OR

How to create the Object for the classes which are coming dynamically from the database or from some file at runtime.

The limitation with new keyword is, It demands the class name at the begining or at the time of compilation so new keyword is not suitable to create the object for the classes which are coming from database or files at runtime dynamically.

In order to create the object for the classes which are coming at runtime from

database or files, we should use newInstance() method available in java.lang.Class class.

```
-----  
class Student{}  
class Player{}  
class Employee{}  
class Customer{}  
  
class ObjectAtRuntime  
{  
    public static void main(String [] args) throws Exception  
    {  
        Object obj = Class.forName(args[0]).newInstance();  
        System.out.println("Object created for :" + obj.getClass().getName());  
    }  
}
```

Here By using command line Argument, if we pass the class name at runtime then based on parameter appropriate class object will be created.

```
-----  
class Student  
{  
    public void read()  
    {  
        System.out.println("Let's read Java");  
    }  
}  
class Player  
{  
    public void play()  
    {  
        System.out.println("Let's play cricket");  
    }  
}  
class ObjectAtRuntime1  
{  
    public static void main(String[] args) throws Exception  
    {  
        Object obj = Class.forName(args[0]).newInstance();  
  
        if(obj instanceof Student)  
        {  
            Student s1 = (Student) obj;  
            s1.read();  
        }  
        else if(obj instanceof Player)  
        {  
            Player p1 = (Player) obj;  
            p1.play();  
        }  
    }  
}
```

\*\* What is the difference between java.lang.ClassNotFoundException and java.lang.NoClassDefFoundError

java.lang.ClassNotFoundException :-

-----  
It occurs when we try to load the required .class file at runtime by using Class.forName(String className) statement or loadClass() static of ClassLoader class and if the required .class file is not available at runtime then we will get an exception i.e java.lang.ClassNotFoundException

Note :- It does not have any concern at compilation time, at run time, JVM will simply check whether the required .class file is available or not.

```
class Foo
{
    static
    {
        System.out.println("static block of Foo class");
    }
}
public class ClassNotFoundExceptionDemo
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("Player");
    }
}
```

-----  
java.lang.NoClassDefFoundError :

-----  
It occurs when the class was present at the time of COMPILATION but at runtime the required .class file is not available(manually deleted by user ) Or it is not available in the current directory (Misplaced) then we will get an exception i.e java.lang.NoClassDefFoundError.

```
class Alpha
{
    public void greet()
    {
        System.out.println("Hello Everyone!!!");
    }
}

public class NoClassDefFoundErrorDemo
{
    public static void main(String[] args)
    {
        new Alpha().greet();
    }
}
```

Note : After compilation delete or place the Alpha.class in another location (folder)

-----  
20-07-2024

-----  
A non static variable can't access from static method, what is the reason ?

All the static members (static variable, static block, static method, static nested inner class) are loaded at the time of loading the .class file into JVM Memory.

At class loading phase object is not created because object is created in the 2nd phase i.e Runtime data area so at the TIME OF EXECUTION OF STATIC METHOD, NON STATIC VARIABLE WILL NOT BE AVAILABLE hence we can't access non static variable from static context[static block, static method and static nested inner class]

```
-----  
class Test
{
    int var;

    public Test(int var)
```

```
{  
    this.var = var;  
}  
public static void access()  
{  
    System.out.println(var);  
}  
}  
}  
public class StaticTest  
{  
    public static void main(String[] args)  
    {  
        Test t1 = new Test(15);  
        Test.access();  
    }  
}
```

-----  
Runtime Data Areas :

-----  
It is also known as Memory Area.

Once a class is loaded then based on variable type method type it is divided into different memory areas which are as follows :

- 1) Method Area
- 2) HEAP Area
- 3) Stack Area
- 4) PC Register
- 5) Native Method Stack

Actually the .class file is dumped inside method area and returns java.lang.Class class object.

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.

-----  
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;

import java.util.Scanner;

public class Test

```
{  
    Scanner sc = new Scanner(System.in);  
    int x = 100;  
  
    static Test t1 = new Test();  
}
```

```

static int y = 500;

public void input() {}

public void show() {}

public void accept() {}

public void display() {}

public void getData() {}

}

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("com.ravi.method_area.Test");
        System.out.println("Class Name is :" + cls.getName());

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

        System.out.println("Methods available in the class :");

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

        System.out.println("Total Number of methods :" + count);

        System.out.println("Variables available in the class :");

        Field[] fields = cls.getDeclaredFields();

        count = 0;
        for(Field field : fields)
        {
            System.out.println(field.getName());
            count++;
        }

        System.out.println("Total Number of Variables :" + count);

    }

}

```

Note :- `getDeclaredMethods()` is a predefined non static method available in `java.lang.Class` class , the return type of this method is `Method` array where

Method is a predefined class available in java.lang.reflect sub package.

getDeclaredFields() is a predefined non static method available in java.lang.Class class , the return type of this method is Field array where Field is a predefined class available in java.lang.reflect sub package.

Field and Method both the classes are providing getName() method to get the name of the field and Method.

-----  
HEAP AREA :

-----  
Whenever we create an object in java then the properties and behavior of the object are stored in a special memory area called HEAP AREA.

We have only one HEAP AREA per JVM.

-----  
STACK Area :

-----  
All the methods are executed as a part of Stack Area.

Whenever we call a method in java then internally one stack Frame will be created to hold method related information.

Every Stack frame contains 3 parts

- 1) Local Variable
- 2) Frame Data
- 3) Operand Stack.

We have multiple stack area for a single JVM.

JVM creates a separate thread for every runtime Stack.

-----  
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 and 1000x  
  
        //GC [2 objects 2000x and 4000x are eligible 4 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;
    }
}
-----
```

22-07-2024

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 + JIT Compiler]

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 like method signature, variables, native method code and make it available to JVM at the time of execution so the overall execution becomes very fast.  
-----

How to achieve abstraction in java ?  
-----

Abstraction is a technique through which we can display only the essential details without showing the background details.

In java we can achieve abstraction by using the following 2 ways :

- 1) Abstract class and abstract method (0-100% abstraction[Partial abstraction])
  - 2) By using interface [We can achieve 100% abstraction]
- 

23-07-2024

Working with abstract class and abstract method :  
-----

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 an abstract class as a final.

```
-----  
abstract class Shape  
{  
    public abstract void draw();  
}  
  
class Rectangle extends Shape  
{  
    @Override  
    public void draw()  
    {  
        System.out.println("Drawing Rectangle!!");  
    }  
}  
class Square extends Shape  
{  
    @Override  
    public void draw()  
    {  
        System.out.println("Drawing Square!!");  
    }  
}  
  
public class AbstractDemo1  
{  
    public static void main(String[] args)  
    {  
        Shape s;  
  
        s = new Rectangle(); s.draw();  
        s = new Square(); s.draw();  
  
    }  
}  
-----  
abstract class Calculate  
{  
    public abstract void doSum(int x, int y);  
    public abstract void getSquare(int a);  
    public abstract void getCube(int b);  
}  
class ArithmeticOperation extends Calculate  
{  
    @Override  
    public void doSum(int x, int y)
```

```

        {
            System.out.println("Sum is :" +(x + y));
        }

@Override
    public void getSquare(int x)
{
    System.out.println("Square is :" +(x*x));
}

@Override
public void getCube(int x)
{
    System.out.println("Cube is :" +(x*x*x));
}

}

public class AbstractDemo2
{
    public static void main(String[] args)
    {
        ArithmeticOperation a = new ArithmeticOperation();
        a.doSum(100,200);
        a.getSquare(5);
        a.getCube(9);
    }
}

-----
abstract class Car
{
    protected int speed = 100;

    public Car()
    {
        System.out.println("Car Class Constructor");
    }

    public void getCarDetails()
    {
        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 AbstractDemo3
{
    public static void main(String[] args)
    {
        Car c = new Honda();
        System.out.println("Car speed is :" +c.speed);
        c.getCarDetails();
        c.run();
    }
}

```

Note : Abstract class may contain constructor and it will be executed with the help of super keyword through sub class object.

\*If we can't create an object for abstract class then what is the need of writing constructor in the abstract class ?

Abstract class can contain instance variable i.e abstract class can have object properties, these object properties i.e instance variable we can initialize through constructor available in the abstract class.

This super abstract class constructor can be executed by sub class object through super keyword.

```
package com.ravi.collection;

abstract class Shape
{
    protected int data;

    public Shape(int data)
    {
        this.data = data;
    }

    public abstract void area();
}

class Rectangle extends Shape
{
    protected int breadth;

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

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

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

    @Override
    public void area()
    {
        double area = super.data * super.data;
        System.out.println("Area of Square is :" + area);
    }
}

class Circle extends Shape
{
    final double PI = 3.14;
    public Circle(int radius)
    {
```

```

        super(radius);
    }

    @Override
    public void area()
    {
        double area = PI * super.data * super.data;
        System.out.println("Area of Circle is :" +area);
    }
}

public class AbstractDemo4
{
    public static void main(String[] args)
    {
        Rectangle rr = new Rectangle(5, 9);
        rr.area();

        Square ss = new Square(10);
        ss.area();

        Circle cc = new Circle(3);
        cc.area();
    }
}

-----
package com.ravi.collection;

abstract class Alpha
{
    public abstract void show();
    public abstract void demo();
}
abstract class Beta extends Alpha
{
    @Override
    public void show() //demo();
    {
        System.out.println("Beta class implemented show method");
    }
}
class Gamma extends Beta
{
    @Override
    public void demo()
    {
        System.out.println("Gamma class implemented demo method");
    }
}
public class AbstractDemo5 {

    public static void main(String[] args)
    {
        Gamma g = new Gamma();
        g.show();
        g.demo();
    }
}

-----
Array Related Program :
-----
package com.ravi.collection;

```

```
import java.util.Scanner;

class Tiger
{
    private String name;
    private double height;
    private String color;

    public Tiger(String name, double height, String color)
    {
        super();
        this.name = name;
        this.height = height;
        this.color = color;
    }

    @Override
    public String toString() {
        return "Tiger [name=" + name + ", height=" + height + ", color=" +
color + "]";
    }
}

public class ArrayDemo {

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("How many tigers in Forest :");
        int noOfTigers = sc.nextInt();

        Tiger tigers[] = new Tiger[noOfTigers];

        for(int i=0; i<tigers.length; i++)
        {
            System.out.print("Enter the name of the tiger :");
            String name = sc.nextLine();
            name = sc.nextLine();

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

            System.out.print("Enter the Color of the tiger :");
            String color = sc.nextLine();
            sc.nextLine();

            tigers[i] = new Tiger(name, height, color);
        }

        System.out.println("Retrieve the Tiger object");

        for(Tiger tiger : tigers)
        {
            System.out.println(tiger);
        }
    }
}
```

```
-----  
package com.ravi.collection;  
  
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 Elephant extends Animal  
{  
    @Override  
    public void checkup()  
    {  
        System.out.println("Elephant Checkup ");  
    }  
}  
  
public class AbstractDemo6  
{  
    public static void main(String[] args)  
    {  
        Lion lions[] = {new Lion(), new Lion(), new Lion()};  
        Bird birds[] = {new Bird(), new Bird(), new Bird(), new Bird()};  
        Elephant elephants[] = {new Elephant(), new Elephant()};  
  
        checkAnimals(lions);  
        checkAnimals(birds);  
        checkAnimals(elephants);  
    }  
  
    public static void checkAnimals(Animal animals[])  
    {  
        for(Animal animal : animals)  
        {  
            animal.checkup();  
        }  
    }  
-----  
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 interfcae contains only abstract method that means there is a guarantee that inside an interfcae we don't have concrete or general or instance methods.

From java 8 onwards we have a facility to write default and static methods.

By using interface we can achieve 100% abstraction concept because it contains only abstract methods.

In order to implement the member of an interface, java software people has provided implements keyword.

All the methods declared inside an interface is by default public and abstract so at the time of overriding we can't reduce the visibility.

All the variables declared inside an interface is by default public, static and final so it must be initialized at the time of declaration.

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 interfcae we can acheive multiple inheritance in java.

We can achieve loose coupling using inetrface.

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 on interface concept :  
-----  
package com.ravi.interface_demo;  
  
sealed interface Moveable permits Car  
{  
    int speed = 100; //Static blank final field [public + static + final]  
    void move(); //public + abstract  
}  
  
non-sealed class Car implements Moveable  
{  
    @Override  
    public void move()  
    {  
        //speed = 150;           //Invalid  
        System.out.println("Moving with my car having speed :" + speed);  
    }  
}  
}  
public class InterfaceDemo1  
{  
    public static void main(String[] args)  
    {  
        Moveable m = new Car();  
        m.move();  
        System.out.println("Car Speed is :" + Moveable.speed);  
    }  
}
```

```

}

-----  

//Program on Banking Application

package com.ravi.interface_demo;

import java.util.Scanner;

interface Bank
{
    public abstract void deposit(double amount);
    public abstract void withdraw(double amount);
}

class Customer implements Bank
{
    private double balance = 10000;

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

    @Override
    public void withdraw(double amount)
    {
        if(amount > this.balance)
        {
            System.err.println("Insufficient Balance..");
        }
        else
        {
            this.balance = this.balance - amount;
            System.out.println("Balance after withdraw is :"+this.balance);
        }
    }
}
public class BankApplication
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        Customer ravi = new Customer();
        System.out.print("Enter the amount you want to deposit : ");
        double deposit = sc.nextDouble();
        ravi.deposit(deposit);

        System.out.print("Enter the amount you want to withdraw : ");
        double withdraw = sc.nextDouble();
        ravi.withdraw(withdraw);
    }
}

```

-----  
Assignment :

```
-----  
interface Calculator  
{  
    void doSum(int x, int y);  
    void doSub(int x, int y);  
    void doMul(int x, int y);  
}
```

-----

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. [passing interface as a parameter]

Tightly coupled :- If the degree of dependency of one class to another class is very high then it is called Tightly coupled. [passing more specific class as a parameter]

According to IT industry standard we should always prefer loose coupling so the maintenance of the project will become easy.

What is cohesion and coupling ?

-----

Cohesion : In between class variable and class method we must have tightly coupled relation [Encapsulation] is called cohesion.

Coupling : Two applications are must be loosely coupled so we can pass different objects as per our requirement. [passing as interface as a parameter]

-----

Can we take interface as a return type of the method ?

-----

According to the industry, It is always preferable to take interface as a return type of the method so we can return any type of class object which are implementing from the interface.

```
public HotDrink accept()  
{  
    return new Tea(), new Coffee(), new Horlicks()..... any class  
    which is added in future but must be implemented from HotDrink  
}
```

6 files :

-----

HotDrink.java(I)

-----

```
package com.ravi.loose_coupling;
```

```
public interface HotDrink  
{  
    void prepare();  
}
```

Tea.java

-----

```
package com.ravi.loose_coupling;  
  
public class Tea implements HotDrink  
{  
    @Override  
    public void prepare()
```

```

        {
            System.out.println("Preparing Tea");
        }
    }

Coffee.java
-----
package com.ravi.loose_coupling;

public class Coffee implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preapring Coffee");
    }
}

Horlicks.java
-----
package com.ravi.loose_coupling;

public class Horlicks implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preparing Horlicks");
    }
}

Restaurant.java
-----
package com.ravi.loose_coupling;

public class Restaurant
{
    public static void acceptObject(HotDrink hd) //loose coupling
    {
        hd.prepare();
    }
}

Main.java
-----
package com.ravi.loose_coupling;

public class Main
{
    public static void main(String[] args)
    {
        Restaurant.acceptObject(new Tea());
        Restaurant.acceptObject(new Coffee());
        Restaurant.acceptObject(new Horlicks());
    }
}

Note : Here we passing HotDrink interface as a parameter which provides loose
coupling, instead of HotDrink interface if we pass Tea
object then the same application will become tight coupling.

```

-----  
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 25-JULY)

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.

```
interface A
{
    void m1();
}

interface B
{
    void m1();
}

class Implementer implements A,B
{
    @Override
    public void m1()
    {
        System.out.println("Multiple Inheritance using interface");
    }
}
```

```
public class MultipleInheritance {
    public static void main(String[] args)
    {
        new Implementer().m1();
    }
}
```

-----  
26-07-2024

-----  
Extending one interface to another interafce :

-----  
One interface can extend another another, It cann't implement an interface. Only classes can implement an interface.

```
package com.ravi.interface_demo;
```

```
interface A
{
    void m1();
}
```

```
interface B extends A
{
    void m2();
}
```

```
class Implementer implements B
```

```
{  
    @Override  
    public void m1()  
    {  
        System.out.println("M1 method Overridden");  
    }  
  
    @Override  
    public void m2()  
    {  
        System.out.println("M2 method Overridden");  
    }  
}  
  
public class ExtendingInterface {  
  
    public static void main(String[] args)  
    {  
        Implementer i = new Implementer();  
        i.m1(); i.m2();  
    }  
}
```

-----  
interface from JDK 1.8 onwards :

Limitation of abstract method :

OR

Maintenance problem with interface in an Industry upto JDK 1.7

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 implmenting from interface because we can override default method inside the sub classes to provide our own specific implementation.

\*By using default method there is no boundation to override the default method in the sub class, if we really required it then we can override to provide my own implementation.

by default, default method access modifier is public so at the time of overriding we should use public access modifier.

default method we can write inside an interface only but not inside a class.

-----  
Vehicle.java(I)

```
package com.ravi.interface_8;

public interface Vehicle
{
    void run();
    void horn();

    default void digitalMeter()    //JDK 1.8
    {
        System.out.println("Digital Meter Facility");
    }
}

Car.java(C)
-----
package com.ravi.interface_8;

public class Car implements Vehicle
{
    @Override
    public void run()
    {
        System.out.println("Car is running");
    }

    @Override
    public void horn()
    {
        System.out.println("Car is having horn facility");
    }

    @Override
    public void digitalMeter()    //JDK 1.8
    {
        System.out.println("Car has Digital Meter..");
    }
}

Bike.java(C)
-----
package com.ravi.interface_8;

public class Bike implements Vehicle
{
    @Override
    public void run()
    {
        System.out.println("Bike is running");
    }

    @Override
    public void horn()
    {
        System.out.println("Bike is having horn facility");
    }
}
```

```
Main.java(C)
-----
```

```

package com.ravi.interface_8;

public class Main {

    public static void main(String[] args)
    {
        Vehicle v = null;
        v = new Car(); v.run(); v.horn(); v.digitalMeter();
        v = new Bike(); v.run(); v.horn();

    }
}

```

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.

Common method [Behavior] -> abstract method  
 Uncommon method [Behavior] -> default method

-----  
 Priority of deafult and concrete method :

-----  
 While working with class and interface, default method is having low priority than concrete method, In the same way class is more powerful than interface.

```

class C extends B implements A {} //Valid

class C implements A extends B {} //Invalid

package com.ravi.interface_8;

interface A
{
    default void m1()
    {
        System.out.println("Default method of interface :");
    }
}

class B
{
    public void m1()
    {
        System.out.println("Concrete method of Class :");
    }
}

class C extends B implements A
{
}

public class PriorityOfDefaultandConcrete
{
    public static void main(String[] args)
    {
        C c1 = new C();
        c1.m1();
    }
}

```

-----  
 Multiple Inheritance by using default Method :

Multiple inheritance is possible in java by using default method inside an interface, here we need to use super keyword to differentiate the super interface methods.  
Before java 1.8, we have abstract method inside an interface but now we can write method body so to execute the default method inside an interface we need to take super keyword with interface name(I.super.m1()).

```
package com.ravi.interface_8;

interface I
{
    public default void m1()
    {
        System.out.println("Default Method of I interface");
    }
}
interface J
{
    public default void m1()
    {
        System.out.println("Default Method of J interface");
    }
}

class Foo implements I,J
{
    @Override
    public void m1()
    {
        I.super.m1();
        J.super.m1();
        System.out.println("Overridden Method");
    }
}
public class MultipleInheritanceUsingDefaultMethod {

    public static void main(String[] args)
    {
        new Foo().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, It is not available to the implementer classes.

It is used to provide common functionality which we can apply/invoke from any BLC/ELC class.

By default static method of an interface contains public access modifier.

-----  
2 files :

-----  
Calculate.java(I)

-----  
package com.ravi.interface\_static;

```
public interface Calculate
{
    static double doSum(int x, int y) //by-default public
```

```

{
    return (x+y);
}

static double getSquare(int x)
{
    return x*x;
}

static double getCube(int x)
{
    return x*x*x;
}

}

Main.java(C)
-----
package com.ravi.interface_static;

public class Main {

    public static void main(String[] args)
    {
        double result = Calculate.doSum(12, 20);
        System.out.println("Sum is :" +result);

        result = Calculate.getSquare(5);
        System.out.println("Square is :" +result);

        result = Calculate.getCube(9);
        System.out.println("Cube is :" +result);
    }
}
-----
```

Program that describe that static method of an interface is only available to interface only that means we can access the static method of an interface by using only one way i.e interface name.

```

package com.ravi.interface_static;

interface Callable
{
    public static void m1()
    {
        System.out.println("m1 method of interface Callable");
    }
}

public class InterfaceDemoWithStatic implements Callable
{
    public static void main(String[] args)
    {
        Callable.m1(); //Only this is the possible way because
                      static method is not available to implementer classes.
    }
}
-----
```

Can we write main method inside an interface ?

Yes, we can write main method inside an interface and it will be executed without the help of class name.

```
package com.ravi.interface_static;

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

#### 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 in java :

If we declare a class without any name (Nameless class) then it is called Anonymous inner class.

Anonymous inner class declaration and object creation both are done in the same line and after the body of anonymous inner class ; is compulsory.

The main purpose of anonymous inner class is, either to extend a super class OR implement an interface i.e creating sub type.

Program that describes how to extend a concrete class by using Anonymous inner class.

```
package com.ravi.anonymous;

class Super
```

```

{
    public void show()
    {
        System.out.println("Super class show method");
    }
}

public class AnonymousWithConcreteClass
{
    public static void main(String[] args)
    {
        //Anonymous inner class
        Super sub = new Super()
        {
            @Override
            public void show()
            {
                System.out.println("Sub class show method");
            }
        };

        sub.show();
    }
}

```

-----  
Program that describes how to extend an abstract class by using Anonymous inner class.

```

package com.ravi.anonymous;

abstract class Shape
{
    public abstract void draw();
}

public class AnonymousWithAbstract {

    public static void main(String[] args)
    {
        //Anonymous inner class
        Shape rect = new Shape()
        {
            @Override
            public void draw()
            {
                System.out.println("Drawing Rectangle");
            }
        };

        //Anonymous inner class
        Shape square = new Shape()
        {
            @Override
            public void draw()
            {
                System.out.println("Drawing Square");
            }
        };

        rect.draw();  square.draw();
    }
}

```

```
    }
}

-----
Program that describes how to implements an interface by using Anonymous inner class.

package com.ravi.anonymous;

interface Drawable
{
    void draw();
}

public class AnonymousWithInterface
{
    public static void main(String[] args)
    {
        Drawable d = new Drawable()
        {
            @Override
            public void draw()
            {
                System.out.println("Drawing Something!!!");
            }
        };
        d.draw();
    }
}
```

29-07-2024

-----  
What is a Functional Interface in java ?

-----  
If an interface contains exactly one abstract method then it is called  
Functional interface.

A functional interface may contain 'n' number of default and static method but  
it must contain only one abstract method.

A functional interface can be annotated by `@FunctionalInterface` annotation.

Example :

```
@FunctionalInterface
public interface Printable
{
    void print();  [SAM = Single abstract Method]
}
```

Program :

-----  
FunctionalInterfaceDemo.java

-----  
package com.ravi.functional\_interface;

```
@FunctionalInterface
interface Student
{
    void writeExam(); // [SAM = Single Abstract Method]
}
```

```
public class FunctionalInterfaceDemo {
```

```
public static void main(String[] args)
{
    Student science = new Student()
    {
        @Override
        public void writeExam()
        {
            System.out.println("Science Students are Writing the Exam");
        }
    };

    Student commerce = new Student()
    {
        @Override
        public void writeExam()
        {
            System.out.println("commerce Students are Writing the Exam");
        }
    };

    science.writeExam();    commerce.writeExam();
}

}
```

-----  
What is Lambda Expression in java ?

-----  
It is a new feature introduced in java from JDK 1.8 onwards.  
It is an anonymous function i.e function without any name.  
In java it is used to enable functional programming.  
It is used to concise our code as well as we can remove boilerplate code.  
It can be used with functional interface only.  
If the body of the Lambda Expression contains only one statement then curly braces are optional.  
We can also remove the variables type while defining the Lambda Expression parameter.  
If the lambda expression method contains only one parameter then we can remove () symbol also.

In lambda expression return keyword is optional but if we use return keyword then {} are compulsory.

Independently Lamda Expression is not a statement.

It requires a target variable i.e functional interface reference only.

Lamda target can't be class or abstract class, it will work with functional interface only.

-----  
Program on Lambda Expression :

```
package com.ravi.lambda_expression;

@FunctionalInterface
interface Printable
{
    void print();
}
public class LambdaDemo1
{
    public static void main(String[] args)
```

```

    {
        Printable p = ()-> System.out.println("Printing");
        p.print();
    }

}

-----
package com.ravi.lambda_expression;

interface Vehicle
{
    void run();
}
public class LambdaDemo2 {

    public static void main(String[] args)
    {
        Vehicle car = ()-> System.out.println("Car is Running");
        Vehicle bike = ()-> System.out.println("Bike is Running");
        Vehicle bus = ()-> System.out.println("Bus is Running");

        car.run(); bike.run(); bus.run();
    }
}

-----
package com.ravi.lambda_expression;

@FunctionalInterface
interface Calculate
{
    void doSum(int x, int y);
}

public class LambdaDemo3
{
    public static void main(String[] args)
    {
        Calculate calc = (c,d) -> System.out.println(c+d);
        calc.doSum(12, 24);
    }
}

-----
package com.ravi.lambda_expression;

@FunctionalInterface
interface Length
{
    int getStringLength(String str);
}
public class LambdaDemo4 {

    public static void main(String[] args)
    {
        Length l = str -> str.length();
        System.out.println("Length is :" +l.getStringLength("India"));
    }
}

-----
package com.ravi.lambda_expression;

```

```

import java.util.Scanner;

/* If the input number is 0 or negative return -1
 * If the input number is even return square of the number
 * If the input number is odd return cube of the number
 */

@FunctionalInterface
interface Calculator
{
    double getSquareAndCube(int num);
}

public class LambdaDemo5 {

    public static void main(String[] args)
    {
        Calculator calc = num ->
        {
            if(num<=0)
            {
                return -1;
            }
            else if(num % 2== 0)
            {
                return (num*num);
            }
            else
            {
                return (num*num*num);
            }
        };

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number :");
        int no = sc.nextInt();

        System.out.println(calc.getSquareAndCube(no));
    }
}

```

What is type parameter<T> in java ?

It is a technique through which we can make our application independent of data type. It is represented by <T>

In java we can pass Wrapper classes as well as User-defined classes to this type parameter.

We cannot pass any primitive type to this type parameter.

```

package com.ravi.lambda_expression;

class Accept<T>
{
    private T x;

    public Accept(T x)
    {

```

```

        super();
        this.x = x;
    }

    public T getX()
    {
        return x;
    }
}

public class TypeParameterDemo
{
    public static void main(String[] args)
    {
        Accept<Integer> intType = new Accept<Integer>(12);
        System.out.println("Integer type :" + intType.getX());

        Accept<String> strType = new Accept<String>("NIT");
        System.out.println("String type :" + strType.getX());

        Accept<Student> stdType = new Accept<Student>(new Student(111));
        System.out.println(stdType.getX());
    }
}

```

record Student(Integer studentId)

}

-----  
30-07-2024  
-----

Working with predefined functional interfaces :

-----  
In order to help the java programmer to write concise java code in day to day programming java software people has provided the following predefined functional interfaces

|                        |                         |
|------------------------|-------------------------|
| 1) Predicate<T>        | boolean test(T x);      |
| 2) Consumer<T>         | void accept(T x);       |
| 3) Function<T, R>      | R apply(T x);           |
| 4) Supplier<T>         | T get();                |
| 5) BiPredicate<T, U>   | boolean test(T x, U y); |
| 6) BiConsumer<T, U>    | void accept(T x, U y);  |
| 7) BiFunction<T, U, R> | R apply(T x, U y);      |
| 8) UnaryOperator<T>    | T apply(T x)            |
| 9) BinaryOperator<T>   | T apply(T x, T y)       |

Note :-

-----  
All these predefined functional interfaces are provided as a part of java.util.function sub package.

Predicate<T> functional interface :

-----  
It is a predefined functional interface available in java.util.function sub package.

It contains an abstract method test() which takes type parameter <T> and returns boolean value. The main purpose of this interface to test one argument boolean expression.

```
@FunctionalInterface  
public interface Predicate<T>  
{  
    boolean test(T x);  
}
```

Note :- Here T is a "type parameter" and it can accept any type of User defined class as well as Wrapper class like Integer, Float, Double and so on.

We can't pass primitive type.

-----  
Programs on Predicate<T> functional interface :

-----  
package com.ravi.functional\_interface\_demo;

```
import java.util.Scanner;  
import java.util.function.Predicate;  
  
public class PredicateDemo1  
{  
    public static void main(String[] args)  
    {  
        //Verify whether a number is even or odd  
        Predicate<Integer> p1 = num -> num%2==0;  
  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter a number to verify even/odd :");  
        int no = sc.nextInt();  
  
        System.out.println("Is "+no+" even number ?"+p1.test(no));  
        sc.close();  
    }  
}
```

-----  
package com.ravi.functional\_interface\_demo;

```
import java.util.Scanner;  
import java.util.function.Predicate;  
  
public class PredicateDemo2 {  
  
    public static void main(String[] args)  
    {  
        // Verify my name starts with 'A' or not ?  
        Predicate<String> p2 = name -> name.startsWith("A");  
  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter your Name :");  
        String name = sc.nextLine();  
  
        System.out.println(name+ " starts with character  
A :" +p2.test(name));  
        sc.close();  
    }  
}
```

-----  
package com.ravi.functional\_interface\_demo;

```

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateDemo3 {

    public static void main(String[] args)
    {
        //Verify my name is Ravi or not
        Predicate<String> p3 = 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 ?"+p3.test(name));
    }
}

```

-----  
Assignments :

- 1) Using Predicate verify whether a person is eligible for vote or not?  
2) Using Predicate verify a year a leap year 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.functional_interface_demo;

import java.util.function.Consumer;

record Employee(Integer employeeId, String employeeName)
{ }

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<Employee> c4 = emp -> System.out.println(emp);
    }
}

```

```

        c4.accept(new Employee(111, "Scott"));

        Consumer<String> c5 = str -> System.out.println(str.toUpperCase());
        c5.accept("India");
    }

}

-----
Function<T,R> functional interface :
-----

```

Type Parameters:

T - the type of the input to the function.  
R - the type of the result of the function.

It is a predefined functional interface available in java.util.function sub package.

It provides an abstract method apply that accepts one argument(T) and produces a result(R).

Note :- The type of T(input) and the type of R(Result) both will be decided by the user.

```

@FunctionalInterface
public interface Function<T,R>
{
    public abstract R apply(T x);
}

-----
package com.ravi.functional_interface_demo;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionDemo1 {

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

        //Finding the square of the number
        Function<Integer, Integer> fn1 = num -> num*num;
        System.out.print("Enter a number to find the square :");
        int no = sc.nextInt();
        System.out.println("Square of "+no+" is :" +fn1.apply(no));

        //Finding the length of my city name
        Function<String, Integer> fn2 = str -> str.length();
        System.out.print("Enter your city name :");
        String cityName = sc.nextLine();
        cityName = sc.nextLine();
        System.out.println("Length of "+cityName+" is :" +fn2.apply(cityName));

        //getting String value
        Function<String, String> fn3 = str -> 50 + 50 + str + 40 + 40;
        System.out.println(fn3.apply("India"));

        sc.close();
    }
}

```

```
-----  
Supplier<T> Functional Interface :  
-----  
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();  
}  
-----  
package com.ravi.functional_interface_demo;  
  
import java.util.Scanner;  
import java.util.function.Supplier;  
  
record Product(Integer productId, String productName, Double productPrice)  
{  
}  
  
public class SupplierDemo1  
{  
    public static void main(String[] args)  
    {  
        Supplier<Product> s1 = ()->  
        {  
            Scanner sc = new Scanner(System.in);  
            System.out.print("Enter Product Id :");  
            int id = sc.nextInt();  
  
            System.out.print("Enter Product Name :");  
            String name = sc.nextLine();  
            name = sc.nextLine();  
  
            System.out.print("Enter Product Price :");  
            double price = sc.nextDouble();  
  
            return new Product(id, name, price);  
        };  
  
        Product product = s1.get();  
        System.out.println(product);  
  
    }  
}  
-----  
package com.ravi.functional_interface_demo;  
  
import java.util.function.Supplier;  
  
public class SupplierDemo2 {  
    public static void main(String[] args)  
    {  
        Supplier<String> s1 = ()-> 50 + 50 + "NIT" + 40 + 40;  
        System.out.println(s1.get());  
    }  
}
```

```

    }

}

-----
31-07-2024
-----
How to create our own functional interfaces with Type parameter :
-----
package com.ravi.functional_interface_ex;

@FunctionalInterface
interface TriFunction<T,U,V,R>
{
    R apply(T t, U u, V v);
}

public class UserDefinedFunctionalInterface
{
    public static void main(String[] args)
    {
        TriFunction<Integer, String, Integer, String> fn = (a,b,c) -> a + b + c;
        System.out.println(fn.apply(12, " NIT ", 18));
    }
}

-----
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 method test.

@FunctionalInterface
public interface BiPredicate<T, U>
{
    boolean test(T t, U u);
}

Type Parameters:

T - the type of the first argument to the predicate
U - the type of the second argument the predicate
Note : return type is boolean.
-----
import java.util.function.*;
public class Lambda11
{
    public static void main(String[] args)
    {
        BiPredicate<String, Integer> filter = (x, y) ->
        {
            return x.length() == y;
        };

        boolean result = filter.test("Ravi", 4);
    }
}

```

```

        System.out.println(result);

        result = filter.test("Hyderabad", 10);
        System.out.println(result);
    }
}

import java.util.function.BiPredicate;

public class Lambda12
{
    public static void main(String[] args)
    {
        // BiPredicate to check if the sum of two integers is even
        BiPredicate<Integer, Integer> isSumEven = (a, b) -> (a + b) % 2 == 0;

        System.out.println(isSumEven.test(2, 3));
        System.out.println(isSumEven.test(5, 7));
    }
}

```

**BiConsumer<T, U> functional interface :**

It is a predefined functional interface available in `java.util.function` sub package.

It is a functional interface in Java that represents an operation that accepts two input arguments and returns no result.

It takes a method named `accept`, which takes two parameters and performs an action without returning any result.

```

@FunctionalInterface
public interface BiConsumer<T, U>
{
    void accept(T t, U u);
}

import java.util.function.BiConsumer;

public class Lambda13
{
    public static void main(String[] args)
    {
        BiConsumer<Integer, String> updateVariables = (num, str) ->
        {
            num = num * 2;
            str = str.toUpperCase();
            System.out.println("Updated values: " + num + ", " + str);
        };

        int number = 15;
        String text = "nit";

        updateVariables.accept(number, text);

        // Values after the update (note that the original values are unchanged)
        System.out.println("Original values: " + number + ", " + text);
    }
}

import java.util.function.BiFunction;

public class Lambda14
{
    public static void main(String[] args)
    {
        BiFunction<String, Integer, String> addLength = (str, num) ->
        {
            String result = str.repeat(num);
            return result;
        };

        String text = "Hello";
        int length = 5;

        String result = addLength.apply(text, length);
        System.out.println(result);
    }
}

```

It is a predefined functional interface available in `java.util.function` sub package.

It is a functional interface in Java that represents a function that accepts two arguments and produces a result R.

The `BiFunction` interface has a method named `apply` that takes two arguments and returns a result.

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{
    R apply(T t, U u);
}
-----
import java.util.function.BiFunction;

public class Lambda14
{
    public static void main(String[] args)
    {
        // BiFunction to concatenate two strings
        BiFunction<String, String, String> concatenateStrings = (str1, str2) -> str1
+ str2;

        String result = concatenateStrings.apply("Hello", " Java");
        System.out.println(result);

        // BiFunction to find the length two strings
        BiFunction<String, String, Integer> concatenateLength = (str1, str2) ->
str1.length() + str2.length();

        Integer result1 = concatenateLength.apply("Hello", "Java");
        System.out.println(result1);
    }
}
```

`UnaryOperator<T>` :

It is a predefined functional interface available in `java.util.function` sub package.

It is a functional interface in Java that represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of `Function` for the case where the operand and result are of the same type.

It has a single type parameter, T, which represents both the operand type and the result type.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,R>
{
    public abstract T apply(T x);
}
-----
import java.util.function.*;
public class Lambda15
{
    public static void main(String[] args)
    {
```

```

        UnaryOperator<Integer> square = x -> x * x;
        System.out.println(square.apply(5));

        UnaryOperator<String> concat = str -> str.concat("base");
        System.out.println(concat.apply("Data"));
    }
}

-----
BinaryOperator<T>
-----
It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation upon two operands of the same type, producing a result of the same type as the operands.
```

This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

It has two parameters of same type, T, which represents both the operand types and the result type.

```

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, U, R>
{
    public abstract T apply(T x, T y);
}

-----
import java.util.function.*;
public class Lambda16
{
    public static void main(String[] args)
    {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(3, 5));
    }
}
```

-----  
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 it's implementing classes.

Using private methods in interfaces have four rules :

- 1) private interface method cannot be abstract.
- 2) private method can be used only inside interface.
- 3) private static method can be used inside other static and non-static interface methods.
- 4) private non-static methods cannot be used inside private static methods.

```

package com.ravi.interface_9;

interface Worker
{
    public abstract void m1(); //JDK 1.0

    public default void m2() //JDK 1.8
```

```

{
    m4();
    m5();
}

public static void m3() //JDK 1.8
{
    m5();
}

private void m4() //Java 9 [Private non static method]
{
    System.out.println("Private non static method");
}

private static void m5() //Java 9 [Private static method]
{
    System.out.println("Private static method");
}

class Implementer implements Worker
{
    @Override
    public void m1()
    {
        System.out.println("M1 method of Implementer class");
    }
}
public class InterfaceNewFeature {

    public static void main(String[] args)
    {
        Implementer i = new Implementer();
        i.m1();
        i.m2();
        Worker.m3();
    }
}

```

Note : We can achieve 100% abstraction by using private method inside interface.

\*What is marker interface in java ?

A marker interface is an interface which does not contain any field or method, basically a marker interface is an empty interface or tag interface.

```

public interface Drawable //Marker interface
{
}

```

The main purpose of Marker interface to provide additional information to the JVM regarding the object like object is Serializable, Clonable OR randomly accessible or not.

In java we have 3 predefined marker interfaces : java.io.Serializable, java.lang.Cloneable, java.util.RandomAccess.

Note : We can create our own marker interface by using instanceof operator.

Does an interface extends any class ?

An interface can't extend a class, It can extend only interface.

Every public method of Object class is implicitly re-declare inside every interface as an abstract method to support upcasting.

```
-----  
package com.ravi.interface_class_relation;  
  
interface A  
{  
}  
}  
public class InterfaceDemo1  
{  
    public static void main(String[] args)  
    {  
        A a1 = null;  
        a1.hashCode();  
        a1.equals(null);  
        a1.toString();  
    }  
}  
-----  
package com.ravi.interface_class_relation;  
  
public interface B  
{  
    @Override  
    public String toString();  
  
    @Override  
    public boolean equals(Object obj);  
  
    @Override  
    public int hashCode();  
}  
-----  
package com.ravi.interface_class_relation;  
  
@FunctionalInterface  
public interface C  
{  
    void m1();  
  
    @Override  
    public String toString();  
  
    @Override  
    public boolean equals(Object obj);  
  
    @Override  
    public int hashCode();  
  
}
```

\*\*\*\*What is difference between abstract class and interface ?

The following are the differences between abstract class and interface.

- 1) An abstract class can contain instance variables but interface variables are by default public , static and final.

- 2) An abstract class can have state (properties) of an object but interface can't have state of an object.
- 3) An abstract class can contain constructor but inside an interface we can't define constructor.
- 4) An abstract class can contain instance and static blocks but inside an interface we can't define any blocks.
- 5) Abstract class can't refer Lambda expression but using Functional interface we can refer Lambda Expression.
- 6) By using abstract class multiple inheritance is not possible but by using interface we can achieve multiple inheritance.

-----OOPs completed-----

01-08-2024

-----  
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.

-----  
Different Criteria of Exception :

-----  
The following are the different criteria for exception :

1) java.lang.ArithmaticException

Whenever we divide a number by 0 (an int value) then we will get  
java.lang.ArithmaticException

```
int x = 10;  
int y = 0;  
int z = x/y;  
System.out.println(z);
```

2) java.lang.NagativeArraySizeException

Whenever we create an array and if we pass some negative value as a size of an array then we will get java.lang.NagativeArraySizeException

```
int []arr = new int[-10];
```

3) java.lang.ArrayIndexOutOfBoundsException

If we try to access the index of the array where element is not available then we will get java.lang.ArrayIndexOutOfBoundsException

```
int []arr = {10,20,30};  
System.out.println(arr[3]); //No value available for 3rd index
```

4) `java.lang.NullPointerException`

Whenever we want to call any non static method on the reference variable which is pointing to null then we will get `java.lang.NullPointerException`

```
String str = null;  
System.out.println(str.length());
```

5) `java.lang.NumberFormatException`

If we try to convert any String into numeric format but the String value is not in a proper format then we will get `java.lang.NumberFormatException`

```
String str = "NIT";  
int no = Integer.parseInt(str);  
System.out.println(no);
```

6) `java.util.InputMismatchException`

At the time of taking the input from the Scanner class if input is not valid from user then we will get `java.util.InputMismatchException`

```
Scanner sc = new Scanner(System.in);  
System.out.println("Enter your Roll :");  
int roll = sc.nextInt();
```

-----  
Exception format :

-----  
The java software people has provided the format of exception so whenever we print exception object then the format is

Fully Qualified Name : `errorMessage`

Package Name + Class Name : `errorMessage`

-----  
WAP that describes Exception is the super class of all the exceptions we have in java.

```
package com.exception;  
  
public class ExceptionDemo {  
  
    public static void main(String[] args)  
    {  
        Exception e = new ArithmeticException();  
        System.out.println(e);  
  
        Exception e1 = new ArithmeticException("Dividing a number by zero");  
        System.out.println(e1);  
  
    }  
}
```

-----  
WAP that describes that whenever an exception is encounter in the program then program will be terminated in the middle.

```
package com.exception;  
  
import java.util.Scanner;  
  
public class ExceptionTermination {
```

```

public static void main(String[] args)
{
    System.out.println("Main method started!!!");
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter value of x :");
    int x = sc.nextInt();
    System.out.print("Enter value of y :");
    int y = sc.nextInt();
    int result = x/y;

    System.out.println("Result is :" + result);
    System.out.println("Main method ended!!!");
    sc.close();
}

}

```

In the above program, If we put the value of y as 0 then program will be terminated in the middle, IT IS CALLED ABNORMAL TERMINATION.  
Actually JVM has a default exception handler which is responsible to handle the exception and terminate the program in the middle abnormally.

-----  
In order to work with exception, java software people has provided the following keywords :

- 1) try block
- 2) catch block
- 3) finally block [Java 7 try with resources]
- 4) throw
- 5) throws

-----  
Key points to remember :

- > With try block we can write either catch block or finally block or both.
- > In between try and catch we can't write any kind of statement.
- > try block will trace our program line by line.
- > If we have any exception inside the try block,With the help of JVM, try block will automatically create the appropriate Exception object and then throw the Exception Object to the nearest catch block.
- > In the try block whenever we get an exception the control will directly jump to the nearest catch block so the remaining code of try block will not be executed.
- > catch block is responsible to handle the exception.
- > catch block will only execute if there is an exception inside try block.

-----  
try block :

Whenever our statement is error suspecting statement OR Risky statement then we should write that statement inside the try block.

try block must be followed either by catch block or finally block or both.

\*try block is responsible to trace our code line by line, if any 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 executed if there is an exception in the try block.

-----  
02-08-2024  
-----

Program on try-catch :

```
-----  
package com.exception;  
  
import java.util.Scanner;  
  
public class TryDemo  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main method started!!!");  
        Scanner sc = new Scanner(System.in);  
        try  
        {  
            System.out.print("Enter value of x :");  
            int x = sc.nextInt();  
            System.out.print("Enter value of y :");  
            int y = sc.nextInt();  
            int result = x/y;  
            System.out.println("Result is :" + result);  
            System.out.println("End of try block :");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Inside Catch Block");  
            System.err.println(e);  
        }  
        System.out.println("End of Main Method");  
        sc.close();  
    }  
}
```

In the above program if we put the value of y as 0 but still program will be executed normally because we have used try-catch so it is a normal termination even we have an exception in the program.

```
-----  
package com.exception;  
  
public class ExceptionThrow {  
  
    public static void main(String[] args)  
    {  
        try  
        {  
            throw new ArithmeticException("Zero division");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Inside Catch Block");  
            System.err.println(e);  
        }  
    }  
}
```

From the above program it is clear that try block implicitly creating the exception object with the help of JVM and throwing the exception object to the nearest catch block.

The main purpose of Exception handling to provide user-friendly message to our end user as shown in the program.

```
package com.ravi.basic;

import java.util.Scanner;

public class CustomerDemo
{
    public static void main(String[] args)
    {
        System.out.println("Hello Client, Welcome to my application");
        Scanner sc = new Scanner(System.in);
        try
        {
            System.out.print("Please enter the value of x :");
            int x = sc.nextInt();
            System.out.print("Please enter the value of y :");
            int y = sc.nextInt();
            int result = x /y;
            System.out.println("Result is :" +result);
        }
        catch(Exception e)
        {
            System.err.println("Don't put zero here");
        }
        sc.close();
        System.out.println("Thank you for visiting my application");
    }
}
```

Exception handling = No Abnormal Termination + User-friendly message on wrong input given by the client.

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 error message, exception class location, exception method name and exception line number.
- 3) public String toString() :- It will convert the exception into String representation.

package com.ravi.basic;

```
public class PrintStackTrace
{
    public static void main(String[] args)
    {
        System.out.println("Main method started...");
        try
        {
            String x = "NIT";
            int y = Integer.parseInt(x);
```

```

        System.out.println(y);
    }
    catch(Exception e)
    {
        e.printStackTrace(); //For complete Exception details
        System.out.println("-----");
        System.out.println(".....");
        System.err.println(e.getMessage()); //only for Exception
message
        System.out.println(".....");
        System.err.println(e.toString());
    }
    System.out.println("Main method ended...");
}

}
-----

```

#### Working with Specific Exception :

While working with exception, in the corresponding catch block we can take Exception (super class) which can handle any type of Exception.

On the other hand we can also take specific type of exception (ArithmetiException, InputMismatchException and so on) which will handle only one type i.e specific type of exception.

```

package com.ravi.basic;

import java.io.IOException;
import java.util.InputMismatchException;
import java.util.Scanner;

public class SpecificException
{
    public static void main(String[] args) throws IOException
    {
        System.out.println("Main started");

        Scanner sc = new Scanner(System.in);

        try
        {
            System.out.print("Enter your Roll :");
            int roll = sc.nextInt();
            System.out.println("Your Roll is :" + roll);

        }
        catch(InputMismatchException e)
        {
            e.printStackTrace();
        }
        sc.close();
        System.out.println("Main ended");
    }
}

-----
package com.ravi.basic;

public class SpecificException1
{
    public static void main(String[] args)
    {
        try

```

```

        {
            throw new OutOfMemoryError();
        }
    catch(Exception e)
    {
        System.out.println("Out of Memory ");
    }

    System.out.println("Main Method Completed");
}

```

Note : From the try block we are throwing the OutOfmemoryError so , Error OR Throwable is required in the catch block for handling purpose, If we pass Exception then catch block will not be executed.

-----  
03-08-2024  
-----

Working with Infinity and Not a number(NaN) :

-----  
10/0 -> Infinity (Java.lang.ArithmetricException)  
10/0.0 -> Infinity (POITIVE\_INFINITY)

0/0 -> Undefined (Java.lang.ArithmetricException)  
0/0.0 -> Undefined (NaN)

While dividing a number with Integral literal in both the cases i.e Infinity (10/0) and Undefined (0/0) we will get java.lang.ArithmetricException because java software people has not provided any final, static variable support to deal with Infinity and Undefined.

On the other hand while dividing a number with with floating point literal in the both cases i.e Infinity (10/0.0) and Undefined (0/0.0) we have final, static variable support so the program will not be terminated in the middle which are as follows

10/0.0 = POSITIVE\_INFINITY  
-10/0.0 = NEGATIVE\_INFINITY  
0/0.0 = NaN  
-----  
package com.ravi.basic;  
  
public class InfinityFloatingPoint  
{  
 public static void main(String[] args)  
 {  
 System.out.println("Main Started");  
 System.out.println(10/0.0);  
 System.out.println(-10/0.0);  
 System.out.println(0/0.0);  
 System.out.println(10/0);  
 System.out.println("Main Ended");  
 }  
}

-----  
Working with multiple try catch :

-----  
According to our application requirement we can provide multiple try-catch in my application to work with multiple execptions.

package com.ravi.basic;  
public class MultipleTryCatch

```

{
    public static void main(String[] args)
    {
        System.out.println("Main method started!!!!");

        try
        {
            int arr[] = {10,20,30};
            System.out.println(arr[3]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.err.println("Array index is out of limit!!!!");
        }

        try
        {
            String str = null;
            System.out.println(str.length());
        }
        catch(NullPointerException e)
        {
            System.err.println("ref variable is pointing to null");
        }

        System.out.println("Main method ended!!!!");
    }
}

```

Here User will get all the Exception messages at a time so it is not a recommended way, to avoid this we introduced a new concept try with multiple catch blocks.

-----  
\* Single try with multiple catch block :

-----  
According to industry standard we should write try with multiple catch blocks so we can provide proper information for each and every exception to the end user.

While working with multiple catch block always the super class catch block must be last catch block.

From java 1.7v this multiple exceptions we can write in a single catch block by using | symbol.

If try block is having more than one exception then always try block will 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(ArithmetricException 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!!!");
    }
}
-----
finally block [100% Guaranteed for Execution]
-----
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");
}

```

Note :- In the above program finally block will be executed, even we have an exception in the try block but here only the resources will be handled but not the exception.

```

-----
package com.ravi.basic;

public class FinallyWithCatch
{
    public static void main(String[] args)
    {
        try
        {
            int []x = new int[-2];    //We can't pass negative size of an
array
            x[0] = 12;
            x[1] = 15;
            System.out.println(x[0]+" : "+x[1]);

        }
        catch(NegativeArraySizeException e)
        {
            System.err.println("Array Size is in negative value...");

        }
        finally
        {
            System.out.println("Resources will be handled here!!!");
        }
        System.out.println("Main method ended!!!!");
    }
}

```

In the above program exception and resources both are handled because we have a combination of try-catch and finally.

Note :- In the try block if we write `System.exit(0)` and if this line is executed then finally block will not be executed.

-----
Limitation of finally Block :

-----
The following are the limitation of finally block :

- 1) In order to close the resources, user is responsible to write finally block manually.
- 2) Due to finally block the length of the program will be increased.

3) In order to close the resources inside the finally block, we need to declare the resources outside of try block.

```
package com.exception;

import java.util.InputMismatchException;
import java.util.Scanner;

public class LimitationOfFinally {

    public static void main(String[] args)
    {
        Scanner sc = null;
        try
        {
            sc = new Scanner(System.in);
            System.out.println("Enter your Employee Number :");
            int empId = sc.nextInt();
            System.out.println("Employee Number is :" + empId);

        }
        catch(InputMismatchException e)
        {
            System.out.println("Inside Catch");
            System.err.println("Input is not in a proper format");
        }
        finally
        {
            System.out.println("Inside Finally");
            sc.close();
        }
    }
}
```

-----  
05-08-2024  
-----

try with resources :

To avoid all the limitation of finally block, Java software people introduced a separate concept i.e try with resources from java 1.7 onwards.

Case 1:

```
-----
try(resource1 ; resource2) //Only the resources will be handled
{
}
```

Case 2 :

```
-----
//Resources and Exception both will be handled
try(resource1 ; resource2)
{
}
catch(Exception e)
{
}
```

Case 3 :

-----

```
try with resources enhancement from java 9v
```

```
Resource r1 = new Resource();
Resource r2 = new Resource();

try(r1; r2)
{
}
catch(Exception e)
{
}
```

There is a predefined interface available in `java.lang` package called `AutoCloseable` which contains predefined abstract method i.e `close()` which throws `Exception`.

There is another predefined interface available in `java.io` package called `Closeable`, this `Closeable` interface is the sub interface for `AutoCloseable` interface.

```
public interface java.lang.AutoCloseable
{
    public abstract void close() throws Exception;
}
public interface java.io.Closeable extends java.lang.AutoCloseable
{
    void close() throws IOException;
}
```

Whenever we pass any 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.

```
ResourceClass rc = new ResourceClass()
try(rc)
{
}
catch(Exception e)
{
}
```

//This `ResourceClass` must implements either `Closeable` or `AutoCloseable` interface so, try block will automatically call the `close()` method as well as try block will get the guarantee of `close()` method support in the respective class.

The following program explains how try block is invoking the `close()` method available in `DatabaseResource` class and `FileResource` class.

```
3 files :
-----
DatabaseResource.java
-----
package com.ravi.auto_closeable;

public class DatabaseResource implements AutoCloseable
{
    @Override
    public void close() throws Exception
    {
        System.out.println("Database resource closed");
    }
}
```

```

    }

}

FileResource.java
-----
package com.ravi.auto_closeable;

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");
    }
}

TryWithResources.java
-----
package com.ravi.auto_closeable;

public class TryWithResources {

    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(ArithmetricException e)
        {
            System.err.println("Divide by zero problem");
        }
    }
}

//Program to close Scanner class automatically using try with resources
package com.ravi.auto_closeable;

import java.util.InputMismatchException;
import java.util.Scanner;

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

        try(sc)
        {
            System.out.println("Enter Student roll number :");
            int roll = sc.nextInt();

            System.out.println("Enter Student Name :");
        }
    }
}

```

```

        String name = sc.nextLine();
        name = sc.nextLine();

        System.out.println("Roll Number is :" + roll);
        System.out.println("Name is :" + name);

    }
    catch(InputMismatchException e)
    {
        System.err.println("Input is not in a proper format");
    }
    catch(Exception e)
    {
        System.err.println("General Problem");
    }
}

}

```

Note :- Scanner class internally implementing Closeable interface so it is providing auto closing facility from java 1.7, as a user we need to pass the reference of Scanner class inside try with resources try()

-----  
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");
    }
}
-----
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 be executed that means if we have an exception in the try block.

```

package com.ravi.basic;

import java.util.InputMismatchException;
import java.util.Scanner;

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

        try(sc )
        {
            System.out.print("Enter your Roll number :");
            int roll = sc.nextInt();
            System.out.println("Your Roll is :" +roll);

        }
    catch(InputMismatchException e)
    {
        System.err.println("Provide Valid input!!");

        try
        {
            System.out.println(10/0);
        }
    catch(ArithmaticException e1)
    {
        System.err.println("Divide by zero problem");
    }

    }
    finally
    {
        try
        {
            throw new ArrayIndexOutOfBoundsException("Array is out
of bounds");
        }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.err.println("Array is out of Bounds");
    }
}

```

```
        }
    }
}
```

Note : inside finally block we can write try catch block

-----  
06-08-2024  
-----

06-08-2024  
-----

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 10/0;
        }
        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 = 100;
            System.out.println(x);
        }
        catch(Exception e)
        {
            x = 200;
            System.out.println(x);
        }
        System.out.println("Main completed!!!");
    }
}

```

-----  
\*\*Difference between Checked Exception and Unchecked Exception :

-----  
Checked Exception :

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, EOFException` 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`

`RuntimeException` is also Unchecked Exception.

Eg:

`ArithmaticException, 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`

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.

\*Why compiler takes very much care regarding the checked Exception ?

As we know Checked Exceptions are very common exception so in case of checked exception "handling is compulsory" because checked Exception depends upon other resources as shown below.

`IOException` (we are depending upon System Keyboard OR Files )  
`FileNotFoundException`(We are depending upon the file)  
`InterruptedException` (Thread related problem)  
`ClassNotFoundException` (class related problem)  
`SQLException` (SQL related or database related problem)  
`CloneNotSupportedException` (Object is the resource)  
`EOFException`(we are depending upon the file)

-----  
07-08-2024

\* What is the difference between throw and throws :

throw [THROWING THE EXCEPTION OBJCET EXPLICITLY.]

We should use throw keyword to throw the exception object explicitly, In case of try block, try block is responsible to create the exception object as well as throw the exception object to the nearest catch block but if we want to throw exception object explicitly then we use throw keyword.

```
throw new ArithmeticException();
throw new LowBalanceException();
```

after using throw keyword the control will transfer to the nearest catch block so after throw keyword statement, the remaining statements are un-reachable.

throws :-

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

How to develop User-defined Exceptions :

As a developer we can develop user-defined checked and user-defined unchecked exception.

If we want to develop checked exception then our user-defined class must extends from `java.lang.Exception`, on the other hand if we want to develop un-checked exception then our user-defined class must extends from `java.lang.RuntimeException`.

In the user-defined exception class we should write No argument constructor(in case if we don't want to pass any error message) and we should write parameterized constructor with `String errorMessage` as a parameter (in case if we want to pass any error message) with super keyword.

In order to throw the exception object explicitly we should use throw keyword as well as our user-defined class object must be of `Throwable` type.

The following program explains the user-defined Exception class must be of sub type of Throwable.

```
-----  
package com.ravi.exception;  
  
class Test extends Throwable  
{  
    public Test(String errorMessage)  
    {  
    }  
}  
  
public class CustomDemo  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            throw new Test("Test Problem");  
        }  
        catch(Throwable e)  
        {  
        }  
    }  
}  
-----  
//Program that describes how to develop userdefined checked exception :  
package com.ravi.exception;  
  
import java.util.Scanner;  
  
@SuppressWarnings("serial")  
class InvalidAgeException extends Exception  
{  
    public InvalidAgeException()  
    {  
    }  
    public InvalidAgeException(String errorMessage)  
    {  
        super(errorMessage);  
    }  
}  
  
public class UserDefinedChecked  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main Application started");  
        Scanner sc = new Scanner(System.in);  
        try(sc)  
        {  
            System.out.print("Enter your Age :");  
            int age = sc.nextInt();  
  
            if(age<18)  
            {  
                throw new InvalidAgeException("Age is Invalid");  
            }  
        }  
    }  
}
```

```

        else
        {
            System.out.println("Age is greater than 18 so you can
vote");
        }
    }
    catch(InvalidAgeException e)
    {
        System.err.println(e.getMessage());
    }

    System.out.println("Main Application ended");
}

}

```

Assignment :

//WAP to develop un-checked Exception :

```

class GreaterMarksException extends RuntimeException
{
}

```

Some Basic rule we should follow while dealing with Checked Exception :

a) If the try block does not throw any checked exception then in the corresponding catch block we can't handle checked exception. It will generate compilation error i.e "exception never thrown from the corresponding try statement"

Example :-

```

public class Test
{
    public static void main(String[] args)
    {
        try
        {
            //try block is not throwing checked exception
            //i.e. InterruptedException
        }
        catch (InterruptedException e) //error
        {
        }

    }
}

```

Note :- The above rule is not applicable for Unchecked Exception

```

        try
        {

        }
    catch(ArithmeticException e) //Valid
    {
        e.printStackTrace();
    }

```

b) If the try block does not throw any exception then in the corresponding catch block we can write Exception, Throwable because both are the super classes for all types of Exception whether it is checked or unchecked.

```

package com.ravi.method_related_rule;

import java.io.EOFException;
import java.io.FileNotFoundException;

public class CatchingWithSuperClass
{
    public static void main(String[] args)
    {
        try
        {

        }
        catch(Exception e) //Exception and Throwable both are allowed
        {
            e.printStackTrace();
        }
    }
}

```

c) At the time of method overriding if the super class method does not reporting or throwing checked exception then the overridden method of sub class not allowed to throw checked exception otherwise it will generate compilation error but overridden method can throw Unchecked Exception.

```

package com.ravi.method_related_rule;

class Super
{
    public void show()
    {
        System.out.println("Super class method not throwing checked
Exception");
    }
}
class Sub extends Super
{
    @Override
    public void show() // throws InterruptedException
    {
        System.out.println("Sub class method should not throw checked
Exception");
    }
}

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

```

d) If the super class method declare with throws keyword to throw a checked exception, then at the time of method overriding, sub class method may or may not use throws keyword.

If the Overridden method is also using throws keyword to throw checked exception then it must be either same exception class or sub class, it should not be super class as well as we can't add more exceptions in the overridden method.

```

package com.ravi.method_related_rule;

import java.io.FileNotFoundException;
import java.io.IOException;

class Base
{
    public void show() throws FileNotFoundException
    {
        System.out.println("Super class method ");
    }
}
class Derived extends Base
{
    public void show() //throws IOException
    {
        System.out.println("Sub class method ");
    }
}

public class MethodOverridingWithThrows
{
    public static void main(String[] args)
    {
        System.out.println("Overridden method may or may not throw checked
exception but if it is throwing then must be same or sub class");
    }
}
-----
e) Just like return keyword we can't use throw keyword inside static and non
static block to throw an exception because all initializers must be executed
normally.

```

We can use throw keyword in the protection of try-catch so the code will be executed normally.

```

package com.ravi.method_related_rule;

public class Main
{
    static
    {
        try
        {
            throw new ArithmeticException();
        }
        catch(ArithmaticException e)
        {

        }
    }

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

08-08-2024

-----  
Exception propagation [Propagation of Exception from Callee to Caller]

-----  
Whenever we call a method and if the the callee method contains any kind of exception 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. [08-August]

-----

```
package com.ravi.exception;

public class ExceptionPropagation
{
    public static void main(String[] args)
    {
        System.out.println("Main method started.");
        try
        {
            m1();
        }
        catch(Exception e)
        {
            System.err.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."); //This line will not be
executed
    }
    public static void m2()
    {
        System.out.println(10/0);
    }
}
```

In the above program exception is handled by main method so m2 and m1 both the methods are terminated by JVM so the remaining part of m1() method will not be executed, even we handled the exception in main method.

-----

Handling Exception while calling a method ?

-----

Whenever we are calling a method and if that method is throwing any checked Exception OR (Exception) by using throws keyword then at the time of calling caller method, It must be protected either by try-catch or declare the method as throws

MethodCalling.java

-----

```
package com.ravi.exception;

public class MethodCalling {
```

```

public static void main(String[] args)
{
    try
    {
        m1();
    }
    catch(InterruptedException e)
    {

    }
}

public static void m1() throws InterruptedException
{
}

}

-----
public class MethodCalling {

    public static void main(String[] args)
    {
        m1(); //Valid because method is throwing un-checked
    }

    public static void m1() throws NullPointerException
    {

    }
}

-----
public class MethodCalling {

    public static void main(String[] args)
    {
        try
        {
            m1(); //Here try-catch OR throws
        }
        catch(InterruptedException e)
        {

        }
    }

    public static void m1() throws Exception
    {

    }
}

}

-----
09-08-2024
-----
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 should 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.

In order to compare two enum constants we have `final equals(Object obj)` method in the `java.lang.Enum` class which will compare two objects based on the memory reference same as `==` operator.

In the `Enum` class `name()` and `ordinal()` both are final method so we can't change the name and order position of an enum constant but `Enum` class has also provided `toString()` method through which we can provide our own user-defined name by overriding `toString()` method in the enum constant.

```
-----  
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 }

    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");
        }
    }
}

```

Here == operator and equals(Object obj) method both will return true because equals(Object obj) method internally uses == operator only.

---

```

public class Test4
{
    private enum Season //private, public, protected, static
    {
        SPRING, SUMMER, WINTER, RAINY;
    }

    public static void main(String[] args)
    {
        System.out.println(Season.RAINY);
    }
}

```

An enum declared inside a class can be private, protected, public and static.

---

```

//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 by default extends from java.lang.Enum class so enum can't extend any other class.

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

Note : Every enum is implicitly final so any class can't extend enum

-----  
//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.name()+" 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

    private 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;

    private Season(String msg)
    {
        this.msg = msg;
    }
}

```

```

private 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
    }
}
-----
public class Test14
{
    enum Day
    {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
    }

    public static void main(String args[])
    {
        Day day = Day.SUNDAY;

        switch(day)

```

```
{  
    case SUNDAY:  
        System.out.println("Sunday");  
        break;  
    case MONDAY:  
        System.out.println("Monday");  
        break;  
    default:  
        System.out.println("other day");  
}  
}  
}
```

We can pass enum constants in switch case statement.

```
-----  
enum Color  
{  
    RED , BLUE, PINK;  
  
    static  
    {  
        System.out.println("static block ");  
    }  
  
    {  
        System.out.println("instance block");  
    }  
}  
public class Test15  
{  
    public static void main(String[] args)  
    {  
        System.out.println(Color.RED);  
    }  
}
```

While working with enum, first line of an enum is reserved for enum objects so we write static block inside an enum then always static block will be executed after constructor.

-----  
10-08-2024

-----  
Input Output in java :

-----  
In order to provide input output operation like creating a file, reading/writing the data from the file and so on.

Java software has provided a predefined package called java.io.

What is the need of File Handling ?

-----  
As we know, to store the data we should use variable in our program but the variable life is temporarily, once the program execution is over, we will not get the value back from the variable.

-----  
In order to store the data permanently we should use files in java.

-----  
In order to send and receive the data from the file we are using 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.

Working with classes :

FileOutputStream :

It is a predefined class available in java.io pacakge which comes under binary Stream.

It is used to create a file and write the data to the file.

It will write the data to file but the data must be available in binary format.

How to Convert character data into binary format :

String class has provided a predefined method called getBytes(), which will convert the character into binary or byte format. The return type of this method is byte[]

```
public byte[] getBytes()
```

Program :

```
package com.ravi.file_handling;
```

```
public class CharacterToBinary
```

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

```
    {        String str = "ABCDEF";
```

```
        //Character to binary
```

```
        byte []b = str.getBytes();
```

```
        for(byte c : b)
```

```
        {
```

```
            System.out.println(c); \\65 66 67 68 69 70
```

```
        }
```

```
}
```

```
}
```

WAP to create a file and write the data by using Binary Stream.

```
package com.ravi.file_handling;
```

```

import java.io.FileOutputStream;
import java.io.IOException;

public class CreateAndWrie
{
    public static void main(String[] args) throws IOException
    {
        var fout = new FileOutputStream("C:\\new\\India.txt");

        try(fout)
        {
            String str = "India";
            byte[] bytes = str.getBytes();

            fout.write(bytes);

        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

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

```

Note : FileOutputStream class a provided a predefined non static method write(byte []a) through which we can write the binary data to the file.

-----  
How to read the data from the File in binary format :

-----  
In order to read the data in binary format, java.io has probided a predefined class called FileInputStream.

FileInputStream :

-----  
It is a predefined class available in java.io package. It is used to read the data from the existing file in binary format.

In order to read the data we have a predefined method called read() which will read one character at a time and return the UNICODE (ASCII) value of that particular character. If the data is not available in the file(which represents EOF , End of File) then it will return -1.

```

public int read()
-----
package com.ravi.file_handling;

import java.io.FileInputStream;
import java.io.IOException;

public class ReadDataFronFile {

    public static void main(String[] args) throws IOException
    {
        var fin = new FileInputStream("C:\\new\\India.txt");

        try(fin)
        {
            int i;
            while(true)
            {

```

```

        i = fin.read();
        if(i==-1)
            break;
        System.out.print((char)i);
    }
}
catch(Exception e)
{
    e.printStackTrace();
}

}

```

12-08-2024

#### Working with Character Stream :

In order to work with character Stream, we have two predefined abstract classes Reader and Writer to perform read and write operation.

#### 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");

        try(fw)
        {
            fw.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");

        try(fw)
        {

```

```

        char c[ ] = {'H','E','L','L','O', ' ', ' ', 'W','O','R','L','D'};
        fw.write(c);
        System.out.println("Success....");
    }
    catch(Exception 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.

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**.

While reading the object data from the file, if the object is not available in the file then it will throw an exception **java.io.EOFException**. (End of file Exception)

```

3 files (Writing Single Object to the file)
-----
package com.ravi.ser_des;

import java.io.Serializable;

public class Employee implements Serializable
{
    private Integer employeeId;
    private String employeeName;
    private Double employeeSalary;

    public Employee(Integer employeeId, String employeeName, Double
employeeSalary) {
        super();
        this.employeeId = employeeId;
        this.employeeName = employeeName;
        this.employeeSalary = employeeSalary;
    }

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

```
package com.ravi.ser_des;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class StoreEmployeeObject {

    public static void main(String[] args) throws IOException
    {
        var fout = new FileOutputStream("C:\\new\\EmployeeData.txt");
        var oos = new ObjectOutputStream(fout);

        try(fout; oos)
        {
            Employee e1 = new Employee(111, "Scott", 45890.90);
            oos.writeObject(e1);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        System.out.println("Object data stored .....");
    }
}
```

```
-----
package com.ravi.ser_des;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class RetrieveEmployeeObject {

    public static void main(String[] args) throws IOException
    {
        var fin = new FileInputStream("C:\\new\\EmployeeData.txt");
        var ois = new ObjectInputStream(fin);

        try(fin ; ois)
        {
            Employee emp = (Employee) ois.readObject();
            System.out.println(emp);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```
3 files : (Writing Multiple objects to the file)
-----
Product.java
-----
StoreProductObject.java
```

```

RetrieveProductObject
-----
package com.ravi.ser_des;

import java.io.Serializable;
import java.time.LocalDate;
import java.util.Scanner;

public class Product implements Serializable
{
    private Integer productId;
    private String productName;
    private Double productPrice;
    private LocalDate manufacturedDate;

    public Product(Integer productId, String productName, Double productPrice,
LocalDate manufacturedDate) {
        super();
        this.productId = productId;
        this.productName = productName;
        this.productPrice = productPrice;
        this.manufacturedDate = manufacturedDate;
    }

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

    public static Product getProductObject()
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Product Id :");
        Integer id = sc.nextInt();
        System.out.print("Enter Product Name :");
        String name = sc.nextLine();
        name = sc.nextLine();
        System.out.print("Enter Product Price :");
        Double price = sc.nextDouble();

        LocalDate d = LocalDate.now();

        Product p1 = new Product(id, name, price, d);
        return p1;
    }
}
-----
package com.ravi.ser_des;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Scanner;

public class StoreProductObject
{
    public static void main(String[] args) throws IOException
    {
        var fout = new FileOutputStream("C:\\new\\ProductData.txt");
        var oos = new ObjectOutputStream(fout);
        var sc = new Scanner(System.in);

```

```

        try(fout; oos)
        {
            System.out.print("How many Product Object you want to
store :");
            int obj = sc.nextInt();

            for(int i=1; i<=obj; i++)
            {
                Product product = Product.getProductObject();
                oos.writeObject(product);
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        System.out.println("Object Stored Successfully");

    }

}

-----
package com.ravi.ser_des;

import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class RetrieveProductObject {

    public static void main(String[] args) throws IOException, Exception
    {
        var fin = new FileInputStream("C:\\\\new\\\\ProductData.txt");
        var ois = new ObjectInputStream(fin);

        try(fin;ois)
        {
            Product p = null;

            while((p = (Product)ois.readObject())!=null)
            {
                System.out.println(p);
            }
        }
        catch(EOFException e)
        {
            System.err.println("End of File Reached!!!"+e);
        }
    }

}

-----
transient keyword in java :

While performing serialization operation, If we don't want to serialize any
particular field then we should declare that field with transient
keyword.
Decalring transient keyword on a particular field will not serailzed that field
and we will get default value.

```

Example :

```
-----  
public class Employee implements Serializable  
{  
    private transient int employeeId;  
    private transient String employeeName;  
    private transient Double employeeSalary;  
}
```

Note : In the Employee class variables are declared with transient keyword so they will not serialized and we will get default value for all the fields.

```
employeeId -> 0  
employeeName -> null  
employeeSalary -> null
```

Multithreading :

-----  
Program Name : ThreadDemo.java

```
-----  
package com.ravi.thread;  
  
class UserThread extends Thread  
{  
    @Override  
    public void run()  
    {  
        System.out.println("Child Thread is Running");  
    }  
}  
  
public class ThreadDemo  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main Thread started");  
        UserThread ut = new UserThread();  
        ut.start();  
        System.out.println("Main Thread ended");  
    }  
}
```

=====

Program IsAlive.java

```
-----  
package com.ravi.basic;  
  
class Foo extends Thread  
{  
    @Override  
    public void run()  
    {  
        System.out.println("Child thread is running...");  
        System.out.println("It is running with separate stack");  
    }  
}  
public class IsAlive  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main Thread is started..");  
    }  
}
```

```

        Foo f = new Foo();
        System.out.println("Thread has not started yet so :" + f.isAlive());
        f.start(); //new Thread has created
        System.out.println("Thread has started so :" + f.isAlive());
        f.start(); //java.lang.IllegalThreadStateException
    }
=====
ExceptionDemo.java
-----
package com.ravi.basic;

class Stuff extends Thread
{
    @Override
    public void run()
    {
        System.out.println("Child Thread is Running!!!!");
    }
}
public class ExceptionDemo
{
    public static void main(String[] args)
    {
        System.out.println("Main Thread Started");

        Stuff s1 = new Stuff();
        Stuff s2 = new Stuff();

        s1.start();
        s2.start();

        System.out.println(10/0);
        System.out.println("Main Thread Ended");
    }
}

```

Note :- Here main thread is interrupted due to AE but still child thread will be executed because child 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)
{
    Sample s = new Sample();
    s.start(); //start a new Thread

    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 : Here processor is frequently switching from main thread to Thread-0  
thread so output is un-predicatable

---

ThreadName.java

```

-----
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) throws InterruptedException
    {
        DoStuff t1 = new DoStuff();
        DoStuff t2 = new DoStuff();

        t1.start();
        t2.start();
    }
}

```

---

System.out.println(Thread.currentThread().getName()+" thread is  
running.....");  
}

}

---

ThreadName1.java

```

-----
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 Here..");
    }
}
=====
package com.ravi.basic;

import java.util.Scanner;

class BatchAssignment extends Thread
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();

        if(name !=null && name.equalsIgnoreCase("Placement"))
        {
            this.placementBatch();
        }
        else if(name !=null && name.equalsIgnoreCase("Regular"))
        {
            this.regularBatch();
        }
        else
        {
            throw new NullPointerException("Name can't be null");
        }
    }

    public void placementBatch()
    {
        System.out.println("I am a placement batch student.");
    }

    public void regularBatch()
    {
        System.out.println("I am a Regular batch student.");
    }
}

public class ThreadName2
{
    public static void main(String[] args)

```

```

{
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter your Batch Title :");
    String title = sc.next();

    BatchAssignment b = new BatchAssignment();
    b.setName(title);

    b.start();
    sc.close();
}

}

=====
Thread.sleep(long millisecond) :
-----
Program Name : SleepDemo.java
-----

package com.ravi.basic;

class Sleep extends Thread
{
    @Override
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println("i value is :" + i);
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}

public class SleepDemo
{
    public static void main(String[] args)
    {
        Sleep s = new Sleep();
        s.start();
    }
}

=====

17-08-2024
-----
Program Name : SleepDemo1.java
-----
package com.ravi.basic;

class MyTest extends Thread
{
    @Override
    public void run()
    {
        Thread thread = Thread.currentThread();

```

```

        System.out.println("Child Id is "+thread.getId());

        for(int i=1; i<=5; i++)
        {
            System.out.println("i value is :" +i); //t1 t2 11 22 33
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                System.err.println("Thread has Interrupted");
            }
        }
    }

public class SleepDemo1
{
    public static void main(String[] args)
    {
        Thread thread = Thread.currentThread();
        System.out.println("Id Of Main Thread " +thread.getId());

        MyTest m1 = new MyTest();
        MyTest m2 = new MyTest();

        m1.start();
        m2.start();
    }
}

```

Note : Every time after printing the value the thread will move into sleeping mode so, the output is 11 22 33 44 55

Assignment :

-----  
Thread.sleep(long millis, int nanos)  
-----

Thread life cycle :

After Diagram :

-----  
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 it's time period to complete. Once the time period will be completed then it will re-enter inside the Runnable state to complete its remaining task.

Dead or Exit :

Once a thread has successfully completed its run method 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.

=====

join() Method of Thread class :

=====

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(); //main Thread will be in waiting state
        System.out.println("Main Thread wake up");

        j2.start();
        j3.start();

        System.out.println("Main Thread Ended");
    }

}

-----  

package com.ravi.basic;

class Alpha extends Thread
{
    @Override
    public void run()
    {
        Thread t = Thread.currentThread();
        String name = t.getName(); //Alpha_Thread is current thread

        Beta b1 = new Beta();
        b1.setName("Beta_Thread");
        b1.start();
        try
        {
            b1.join(); //Alpha thread is waiting 4 Beta Thread to complete
            System.out.println("Alpha thread re-started");
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        for(int i=1; i<=10; i++)
        {
            System.out.println(i+" by "+name);
        }
    }
}

public class JoinDemo2
{
    public static void main(String[] args)
    {
        Alpha a1 = new Alpha();
        a1.setName("Alpha_Thread");
        a1.start();
    }
}

class Beta extends Thread
{
    @Override
    public void run()
}

```

```

{
    Thread t = Thread.currentThread();
    String name = t.getName();
    for(int i=1; i<=20; i++)
    {
        System.out.println(i+" by "+name);
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException e) {

        }
    }
    System.out.println("Beta Thread Ended");
}
=====
```

19-08-2024

```

-----
package com.ravi.basic;

public class JoinDemo1
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main Thread Started");

        Thread t = Thread.currentThread();
        String name = t.getName();

        for(int i=1; i<=10; i++)
        {
            System.out.println("i value is :" + i + " by " + name);
            Thread.sleep(500);
        }
        t.join(); // Main thread (Current Thread) is waiting for main thread
                  // to complete [Deadlock]

        System.out.println("Main Thread Ended");
    }
}
```

Note: Here main thread is current thread so main thread is waiting for main thread to complete hence is a Deadlock situation.

-----  
Anonymous Thread class approach with Reference :

-----  
AnonymousInnerWithReference.java

```

-----
package com.ravi.thread;

public class AnonymousInnerWithReference
{
    public static void main(String[] args)
    {
        // Anonymous inner class approach
        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
```

```

        String name = Thread.currentThread().getName();
        System.out.println("Thread Name is :" + name);
    }

    String name = Thread.currentThread().getName();
    System.out.println("Running Thread Name is :" + name);

    t1.start();

}

=====
2) Anonymous inner without Reference :
-----
AnonymousInnerWithoutRef.java
-----
package com.ravi.thread;

public class AnonymousInnerWithoutRef {

    public static void main(String[] args)
    {
        new Thread()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println("Thread Name is :" + name);
            }
        }.start();

        String name = Thread.currentThread().getName();
        System.out.println("Running Thread Name is :" + name);
    }

}

=====
Assigning target by Runnable interface :
-----
RunnableDemo.java
-----
package com.ravi.basic;

class Ravi implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " thread is running Here");
    }
}
public class RunnableDemo
{
    public static void main(String [] args)
    {
        System.out.println("Main Thread is Running Here...");

        Thread t1 = new Thread(new Ravi());
    }
}

```

```

        t1.start();
    }
}
=====
Assigning different targets to different Threads :
-----
RunnableDemo1.java
-----
package com.ravi.basic;

class Tatkal implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println("Tatkal ticket is booked for :" + name);

    }
}

class PremiumTatkal implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println("Premium Tatkal ticket is booked for :" + name);

    }
}

public class RunnableDemo1
{
    public static void main(String[] args)
    {
        Thread tatkal = new Thread(new Tatkal(), "Scott");
        tatkal.start();

        Thread premiumTatkal = new Thread(new PremiumTatkal(), "Smith");
        premiumTatkal.start();
    }
}
-----
Anonymous inner class using Runnable interface :
-----
AnonymousRunnable.java
-----
package com.ravi.basic;

class AnonymousRunnable
{
    public static void main(String [] args)
    {
        System.out.println("Main Thread Started!!!");

        //Anonymous inner class using Runnable
        Runnable r1 = new Runnable()
        {
            @Override
            public void run()
            {

```

```

        String name = Thread.currentThread().getName();
        System.out.println(name+" Thread is Running Here");
    }
};

Thread t1 = new Thread(r1);
t1.start();

}

-----
RunnableUsingLambda.java
-----
package com.ravi.basic;

class RunnableUsingLambda
{
    public static void main(String [] args)
    {
        System.out.println("Main Thread Started!!!");

        //Lambda
        Runnable r1 = ()->
        {
            String name = Thread.currentThread().getName();
            System.out.println(name);
        };

        Thread t1 = new Thread(r1);
        t1.start();
    }
}

-----
ThreadCreation.java
-----
public class ThreadCreation
{
    public static void main(String[] args)
    {
        //Approach 1

        Thread t1 = new Thread(new Runnable()
        {

            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println(name+" is Running");

            }
        });
        t1.start();

        //Approach 2
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println(name+" is Running");

            }
        });
    }
}

```

```

        }, "Child1").start();

//Approach3
Thread thread = new Thread(()->
{
    String name = Thread.currentThread().getName();
    System.out.println("Thread Name is :" + name);
}, "UserThread");
thread.start();

//Approach 4
new Thread(()->System.out.println("My
Thread :" + Thread.currentThread().getName()), "Child2").start();
}

-----
Limitation of Multithreading :
-----
package com.ravi.thread_demo;

class Customer implements Runnable
{
    int availableSeat = 1;
    int wantedSeat;

    public Customer(int wantedSeat)
    {
        super();
        this.wantedSeat = wantedSeat;
    }

    @Override
    public void run()
    {
        String name = null;

        if(availableSeat >= wantedSeat)
        {
            name = Thread.currentThread().getName();
            System.out.println(wantedSeat + " birth is reserved for " + name);
            availableSeat = availableSeat - wantedSeat;
        }
        else
        {
            name = Thread.currentThread().getName();
            System.err.println("Sorry!! " + name + " Tickets are not
available");
        }
    }
}

public class RailwayReservation {
    public static void main(String[] args) throws InterruptedException
    {
        Customer c1 = new Customer(1);

        Thread t1 = new Thread(c1, "Scott");
    }
}

```

```

        Thread t2 = new Thread(c1,"Smith");
        t1.start();
        t2.start();
    }

}

-----
Assignment :
-----
Banking application where two threads are trying to withdraw the amount at the
same time(Lambda Approach)
-----

class Customer
{
    private double balance = 20000;

    public void withdrawAmount(double withdrawAmount)
    {
        String name = null;

        if(withdrawAmount <= balance)
        {
            name = Thread.currentThread().getName();
            System.out.println(name + " has successfully
withdraw :" + withdrawAmount+ " amount");
            balance = balance - withdrawAmount;
        }
        else
        {
            name = Thread.currentThread().getName();
            System.err.println("Sorry "+name+" you have insufficient
Balance");
        }
        // System.out.println("After withdraw balance is :" +balance);
    }
}

public class BankingApplication {
    public static void main(String[] args)
    {
        Customer c1 = new Customer();

        Runnable r1 = ()-> c1.withdrawAmount(20000);

        Thread t1 = new Thread(r1,"Martin");

        Thread t2 = new Thread(r1,"John");

        t1.start(); t2.start();

    }
}
-----
```

21-08-2024

-----  
Synchronization :

-----  
MethodLevelSynchronization.java

-----  
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 :

-----  
BlockLevelSynchronization.java

```

-----
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 BlockLevelSynchronization
{
    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();
    }
}
-----
```

22-08-2024

-----  
Problem with Object level synchronization :

```

ProblemWithObjectLevelSynchronization.java
-----
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(8); //lock2
            }
        };

        Thread t4 = new Thread()
        {
            @Override
            public void run()
            {
                pt2.printTable(9); //lock2
            }
        };
        t1.start();      t2.start();  t3.start();  t4.start();
    }
}

```

From the above program it is clear that, even method is synchronized but two threads are allowed from two different locks so to avoid the problem java software people introduced static synchronization.

---

Static Synchronization :

---

StaticSynchronization.java

---

```

package com.ravi.advanced;
class MyTable
{
    public static synchronized void printTable(int n) //static
synchronization
    {
        for(int i=1; i<=10; i++)
        {
            try
            {
                Thread.sleep(100);

```

```

        }
        catch(InterruptedException e)
        {
            System.err.println("Thread is Interrupted...");
        }
        System.out.println(n+" X "+i+" = "+(n*i));
    }
    System.out.println("-----");
}
}

public class StaticSynchronization
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                MyTable.printTable(5);
            }
        };

        Thread t2 = new Thread()
        {
            @Override
            public void run()
            {
                MyTable.printTable(10);
            }
        };

        Runnable r3 = ()-> MyTable.printTable(15);
        Thread t3 = new Thread(r3);

        t1.start();
        t2.start(); t3.start();
    }
}

```

-----  
Thread Priority :

-----  
PriorityDemo1.java

-----  
package com.ravi.thread\_priority;

-----  
public class PriorityDemo1 {

```

        public static void main(String[] args)
        {
            System.out.println(Thread.MIN_PRIORITY);
            System.out.println(Thread.NORM_PRIORITY);
            System.out.println(Thread.MAX_PRIORITY);
        }
}

```

-----  
PriorityDemo2.java

-----  
package com.ravi.thread\_priority;

-----  
public class PriorityDemo2 {

```

public static void main(String[] args)
{
    Thread t = new Thread();
    System.out.println(t.getPriority());

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

    t.setPriority(11); //java/lang/IllegalArgumentException
    System.out.println(t.getPriority());
}

-----
PriorityDemo3.java
-----
package com.ravi.thread_priority;

public class PriorityDemo3
{
    public static void main(String[] args) //main group, main thread, 1, 5
    {
        Thread t = Thread.currentThread();
        t.setPriority(10);
        System.out.println(t.getPriority());

        Thread t1 = new Thread();
        System.out.println(t1.getPriority());
    }
}

-----
PriorityDemo4.java
-----
package com.ravi.thread_priority;

class MyThread extends Thread
{
    @Override
    public void run()
    {
        int count = 0;
        for(int i=1; i<=100000; i++)
        {
            count++;
            Thread.yield();
        }

        System.out.println("Thread Name
is :"+Thread.currentThread().getName());
        System.out.println("Thread Priority
is :"+Thread.currentThread().getPriority());
    }
}

public class ProrityDemo4 {
    public static void main(String[] args)
    {

```

```

        MyThread mt1 = new MyThread();
        MyThread mt2 = new MyThread();

        mt1.setPriority(Thread.MIN_PRIORITY);
        mt2.setPriority(Thread.MAX_PRIORITY);

        mt1.setName("Last");
        mt2.setName("First");

        mt1.start();  mt2.start();

    }

}

-----
23-08-2024
-----
YieldDemo.java
-----
Thread.yield() :
-----
package com.ravi.thread;

class MyThread extends Thread
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();

        for(int i=1; i<=10; i++)
        {
            System.out.println(i+" by "+name);

            if(name.equals("Child1"))
            {
                Thread.yield(); //Give a chance to Child2
            }
        }
    }
}

public class YieldDemo {

    public static void main(String[] args)
    {
        MyThread mt1 = new MyThread();
        MyThread mt2 = new MyThread();

        mt1.setName("Child1");
        mt2.setName("Child2");

        mt1.start();  mt2.start();
    }
}

-----
ThreadGroup class :
-----
ThreadGroupDemo1.java
-----
package com.ravi.thread;

```

```

class Sample implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();

        for(int i=1; i<=3; i++)
        {
            System.out.println(i+" by "+name);
        }
    }
}

public class ThreadGroupDemo1 {

    public static void main(String[] args)
    {
        ThreadGroup tg = new ThreadGroup("NIT_Threads");

        Thread t1 = new Thread(tg, new Sample(), "Child1");
        Thread t2 = new Thread(tg, new Sample(), "Child2");
        Thread t3 = new Thread(tg, new Sample(), "Child2");

        t1.start();
        t2.start();
        t3.start();

        System.out.println("Group Name is :"+tg.getName());
        System.out.println("Total active threads under this
group :" +tg.activeCount());
    }
}

-----
ThreadGroupDemo2.java
-----
package com.ravi.thread;

class Tatkal implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name+ " booked tatkal Ticket");
    }
}

class PremiumTatkal implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name+ " booked Premium tatkal Ticket");
    }
}

```

```

public class ThreadGroupDemo2 {
    public static void main(String[] args)
    {
        ThreadGroup tatkal = new ThreadGroup("Tatkal");
        ThreadGroup preimumTatkal = new ThreadGroup("PreimumTatkal");

        Thread t1 = new Thread(tatkal, new Tatkal(), "A");
        Thread t2 = new Thread(tatkal, new Tatkal(), "B");

        Thread t3 = new Thread(premiumTatkal, new PremiumTatkal(), "X");
        Thread t4 = new Thread(premiumTatkal, new PremiumTatkal(), "Y");

        t1.start(); t2.start(); t3.start(); t4.start();
    }
}

-----
package com.ravi.thread;

public class ThreadGroupDemo3 {
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println(t.toString());

        System.out.println(".....");
        Thread t1 = new Thread("Child1");
        System.out.println(t1.toString());
    }
}

-----
Daemon thread in java : [Service provider thread]
-----
DaemonThreadDemo1.java
-----
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();24-08-2024
                }
            }
        });
        userThread.start();
        System.out.println("Main Thread Ended...");24-08-2024
    }
}
-----24-08-2024-----
***Inter-Thread Communication :
-----
ITCDemo1.java
-----
package com.ravi.inter_thread_communication;

public class ITCDemo1 {

    public static void main(String[] args) throws InterruptedException
    {
        Object obj = new Object();
        obj.wait();
    }
}

ITCProblem.java
-----
package com.ravi.inter_thread_communication;

//If we don't use communication between two threads then the problem is
class Test extends Thread
{
    private int data = 0;

    @Override
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            this.data = this.data + i; //data = 0 1 3 6 10 15
            try

```

```

        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

public int getData()
{
    return this.data;
}
}

public class ITCProblem
{
    public static void main(String[] args) throws InterruptedException
    {
        Test t1 = new Test();
        t1.start();
        Thread.sleep(1000);
        System.out.println(t1.getData());
    }
}

-----
ITCSolution.java
-----
package com.ravi.inter_thread_communication;

class Demo extends Thread
{
    private int x = 0;

    @Override
    public void run()
    {
        synchronized(this)
        {
            for(int i=1; i<=100; i++)
            {
                x = x + i; //x = 55
            }
            notify();
            System.out.println("Sending notification");
        }
    }
    public int getValue()
    {
        return this.x;
    }
}

public class ITCSolution {
    public static void main(String[] args) throws InterruptedException
    {
        Demo d1 = new Demo();
        d1.start();
    }
}
```

```

        synchronized(d1)
    {
        System.out.println("Main thread is waiting here");
        d1.wait(); //Waiting state by releasing the lock
        System.out.println("Main Thread wake up");

    }
    System.out.println(d1.getValue());
}

}

-----
InterThreadBalance.java
-----
package com.ravi.inter_thread_communication;

class Customer
{
    private double balance = 10000;

    public synchronized void withdraw(double amount)
    {
        System.out.println("Son is going to withdraw");

        if(amount > this.balance)
        {
            System.out.println("Less Balance, Waiting for deposit");
            try
            {
                wait();
                System.out.println("Got Notification, Going to
withdraw");
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
        this.balance = this.balance - amount;
        System.out.println("Balance after withdraw "+this.balance);
    }

    public synchronized void deposit(double amount)
    {
        System.out.println("Going for deposit");
        this.balance = this.balance + amount;
        System.out.println("Amount after deposit :"+this.balance);
        notify();
    }
}

public class InterThreadBalance
{
    public static void main(String[] args) throws InterruptedException
    {
        Customer c1 = new Customer();

        new Thread(()-> c1.withdraw(15000)).start();

        Thread.sleep(10);

        new Thread(()-> c1.deposit(10000)).start();
    }
}

```

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

Lab Question :

-----  
Problem Statement:

You are tasked with creating an education institute course enrollment system using Java. The system should provide courses and offers to students, allowing them to view available courses, ongoing offers, and enroll in their preferred courses.

Classes:

Take one BLC class Course

Attributes:

- > courseId (int): Unique identifier for the course.
- > courseName (String): Name of the course.
- > corseFee (double): Fee for the course.

Methods:

- > Parameterized Constructor to initialize the instance variable.
- > Generate getters for all field
- > Override toString() method

class Offer:

Attributes:

- > offerText (String): Description of the special offer provided by the education institute.

Methods:

- > Offer(String offerText): Constructor to initialize the offer description.
- > getOfferText(): Returns the offer description.

class EducationInstitute:

Attributes:

- > courses (Course[]): An array to store the available courses.
- > offers (Offer[]): An array to store ongoing offers.

Methods:

- > EducationInstitute(): Constructor to initialize courses and offers.
- > getCourses(): Returns the array of available courses.
- > getOffers(): Returns the array of ongoing offers.
- > enrollStudentInCourse(int courseId, String studentName): Simulates the enrollment process and prints a message when a student -> enrolls in a course.

class Student:

Attributes:

- > name (String): Name of the student.
- > institute (EducationInstitute): Reference to the education institute where the student interacts.

Methods:

- > Student(String name, EducationInstitute institute): Constructor to initialize the student with their name and the education institute reference.
- > viewCoursesAndFees(): Displays the available courses and their fees.
- > viewOffers(): Displays the ongoing offers.
- > enrollInCourse(int courseId): Enrolls the student in the specified course using the education institute's enrollment process.

class Main :

The EducationInstituteApp class is the main program that simulates concurrent student interactions using threads.

It creates an education institute, initializes students, and allows them to view course details, ongoing offers, and enroll in courses concurrently without disturbing the execution flow of each thread.

Instructions for Students:

- > Implement the above classes and their methods following the given specifications.
- > Create an instance of EducationInstitute, and initialize courses and offers with hardcoded data for simplicity.
- > Create two students: Virat and Dhoni. Allow them to view available courses, check ongoing offers, and enroll in their preferred courses concurrently using threads.

- > Use the Thread class to simulate concurrent student interactions. Ensure that the system provides a responsive user experience for multiple students.
- > Test your program with multiple executions and verify that students can view course details, offers, and enroll without conflicts.
- > Feel free to enhance the program with additional features or error handling to further improve its functionality.

[Note : Include appropriate comments and use meaningful variable names to make your code more readable and understandable.]

Sample Output :

Available Courses:

1. Mathematics - Fee: Rs.1000.0
2. Science - Fee: Rs.1200.0
3. English - Fee: Rs.900.0

Ongoing Offers:

Special discount: Get 20% off on all courses!

Limited time offer: Enroll in any two courses and get one course free!

Virat has enrolled in the course: Mathematics

Available Courses:

1. Mathematics - Fee: Rs.1000.0
2. Science - Fee: Rs.1200.0
3. English - Fee: Rs.900.0

Ongoing Offers:

Special discount: Get 20% off on all courses!

Limited time offer: Enroll in any two courses and get one course free!

Dhoni has enrolled in the course: Science

-----  
5 files :

-----  
Course.java

-----  
package com.lab;

```
public class Course {  
    private int courseId;  
    private String courseName;  
    private double courseFees;
```

```
public Course(int courseId, String courseName, double courseFees) {
    super();
    this.courseId = courseId;
    this.courseName = courseName;
    this.courseFees = courseFees;
}

public int getCourseId() {
    return courseId;
}

public String getCourseName() {
    return courseName;
}

public double getCourseFees() {
    return courseFees;
}

@Override
public String toString() {
    return "Course [courseId=" + courseId + ", courseName=" + courseName
+ ", courseFees=" + courseFees + "]";
}

}
```

-----  
Offer.java

```
-----
package com.lab;

public class Offer {
    private String offerText;

    public Offer(String offerText)
    {
        super();
        this.offerText = offerText;
    }

    public String getOfferText() {
        return offerText;
    }

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

-----  
EducationInstitute.java

```
-----
package com.lab;

public class EducationInstitute
{
    private Course[] courses;
    private Offer[] offers;

    public EducationInstitute(Course[] courses, Offer[] offers) {
```

```

        super();
        this.courses = courses;
        this.offers = offers;
    }

    public Course[] getCourses()
    {
        return courses;
    }

    public Offer[] getOffers()
    {
        return offers;
    }

    public void enrollStudentInCourse(int courseId, String studentName)
    {
        for(int i=0; i<courses.length; i++)
        {
            if(courseId == courses[i].getCourseId())
            {
                System.out.println(studentName+" has enrolled
in :" +courses[i].getCourseName());
                return;
            }
        }
        System.err.println(courseId+" is not available");
    }

}

/*
*111 Java    27000
* 222 Python 25000
* 333 .net    24000
*
*
*/
-----
```

**Student.java**

---

```

package com.lab;

public class Student {
    private String studentName;
    private EducationInstitute institute;

    public Student(String studentName, EducationInstitute institute)
    {
        super();
        this.studentName = studentName;
        this.institute = institute;
    }

    public void viewCoursesAndFees()
    {
        System.out.println("Available Courses and fees :");
        Course[] courses = institute.getCourses();
        for(Course course : courses)
        {
            System.out.println(course);
        }
    }
}
```

```

    }

    public void viewOffers()
    {
        System.out.println("Available offers :");
        Offer[] offers = institute.getOffers();

        for(Offer offer : offers)
        {
            System.out.println(offer);
        }
    }

    public void enrollInCourse(int courseId)
    {
        institute.enrollStudentInCourse(courseId, studentName);
    }
}

-----

```

### Main.java

```

package com.lab;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) throws InterruptedException
    {
        Scanner sc = new Scanner(System.in);
        Course course[] = new Course[2];

        for(int i=0; i<course.length; i++)
        {
            System.out.print("Enter courseId :");
            int id = sc.nextInt();
            System.out.print("Enter Course Name :");
            String name = sc.nextLine();
            name = sc.nextLine();
            System.out.println("Enter course fees :");
            double fees = sc.nextDouble();

            course[i] = new Course(id, name, fees);
        }

        Offer [] offers = new Offer[2];
        offers[0] = new Offer("Special discount: Get 20% off on all
courses!");
        offers[1] = new Offer("Limited time offer: Enroll in any two courses
and get one course free!");

        EducationInstitute e = new EducationInstitute(course, offers);

        Thread t1 = new Thread()
        {
            Student s1 = new Student("Virat", e);
            @Override
            public void run()
            {
                s1.viewCoursesAndFees();
                s1.viewOffers();
            }
        }.start();
    }
}

```

```

                s1.enrollInCourse(111);
            }
        };

        Thread t2 = new Thread()
        {
            Student s2 = new Student("Dhoni", e);
            @Override
            public void run()
            {
                s2.viewCoursesAndFees();
                s2.viewOffers();
                s2.enrollInCourse(999);
            }
        };

        t1.start();
        t1.join();
        t2.start();
    }

}

-----
26-08-2024
-----
InterThreadNotifyAll.java
-----
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");

    t1.start();
    t2.start();
    t3.start();

    Thread setter = new Thread(() -> r1.setMethod(), "Setter_Thread");

    try
    {
        Thread.sleep(2000);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }

    setter.start();
}
-----
interrupt method of Thread class :
-----
InterruptThread.java
-----
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 (Exception e)
            {
                System.err.println("Thread is Interrupted ");
                e.printStackTrace();
            }
        }
    }
}
public class InterruptThread
{
    public static void main(String[] args)
    {
        Interrupt it = new Interrupt();
        System.out.println(it.getState()); //NEW
    }
}

```

```
        it.start();
        it.interrupt(); //main thread is interrupting the child thread
    }
}
```

Note : Here main thread is interrupting child Thread so whenever child thread will go into sleep or wait state catch block will be executed.

-----  
InterruptThread1.java  
-----

```
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)
    {
        Interrupt it = new Interrupt();
        it.start();
    }
}
```

-----  
InterruptThread2.java  
-----

```
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 by locking the resource");
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Thread interrupted gracefully.");
        }
        finally
        {
            System.out.println("Thread resource can be release here.");
        }
    }
}

```

Note : Here in this program, if main thread will not interrupt the child thread then child thread will not come out from the infinite loop as well as if any resource is locked by child thread then that resource will also not released by child thread.

-----  
27-08-2024  
-----

Deadlock in java :

-----  
Program Name : DeadlockExample.java  
-----

```

public class DeadlockExample
{
    public static void main(String[] args)
    {
        String resource1 = "Ameerpet";
        String resource2 = "S R Nagar";

        // t1 tries to lock resource1 then resource2

        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                synchronized (resource1)
                {
                    System.out.println("Thread 1: locked resource 1");
                    try
                    {
                        Thread.sleep(1000);
                    }
                    catch (Exception e)
                    {}

                    synchronized (resource2) //Nested synchronized block
                    {
                        System.out.println("Thread 1: locked resource 2");
                    }
                }
            }
        }
    }
}

```

```

};

// t2 tries to lock resource2 then resource1
Thread t2 = new Thread()
{
    @Override
    public void run()
    {
        synchronized (resource2)
        {
            System.out.println("Thread 2: locked resource 2");
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {}

            synchronized (resource1) //Nested synchronized block
            {
                System.out.println("Thread 2: locked resource 1");
            }
        }
    }
};

t1.start();
t2.start();
}
}
-----
```

01-09-2024

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 location.

ShallowCopy.java

```
-----
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());
    }
}

```

What is deep copy in java ?

-----  
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);

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

    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 as well as if we modify one object content then another object content will not be modified.

`clone()` method of `Object` class has protected access modifier so we need to override `clone()` method in sub class.

```

-----package
com.ravi.clone_method;

class Customer implements Cloneable
{
    private Integer customerId;
    private String customerName;

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

```

        public Integer getCustomerId() {
            return customerId;
        }

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

        public String getCustomerName() {
            return customerName;
        }

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

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

        @Override
        protected Object clone() throws CloneNotSupportedException
        {
            return super.clone();
        }
    }

public class CloneMethod
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Customer c1 = new Customer(111, "Scott");
        Customer c2 = (Customer)c1.clone();

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

        System.out.println("After Modification :");
        c2.setCustomerId(222);
        c2.setCustomerName("Smith");

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

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

```

Steps to follow for developing of above program :

- 
- 1) class must implement from Cloneable
  - 2) Override clone() method from Object class
  - 3) Invoke clone() method in the protection of try-catch OR throws
  - 4) Perform downcasting
- 

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 (01-SEP-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...");
        System.out.println("Just before Student object destruction");
    }

    public static void main(String[] args) throws InterruptedException
    {
        Student s1 = new Student(111, "Ravi");
        System.out.println(s1);

        s1 = null;

        System.gc();
        Thread.sleep(5000);
        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()`.

-----  
Collection Frameworks in Java (40 - 45% IQ):

Collections framework is nothing but handling individual Objects(Collection Interface) and Group of objects(Map interface).

We know only object can move from one network to another network.

A collections framework is a class library to handle group of Objects.

It is implemented by using `java.util` package.

It provides an architecture to store and manipulate group of objects.

All the operations that we can perform on data such as searching, sorting, insertion and deletion can be done by using collections framework because It is the data structure of Java.

The simple meaning of collections is single unit of Objects.

-----  
It provides the following sub interfaces :

- 1) List (Accept duplicate elements)
- 2) Set (Not accepting duplicate elements)
- 3) Queue (Storing and Fetching the elements based on some order i.e FIFO)

Note : Collection is a predefined interface available in `java.util` package where as Collections is a predefined class which is available from JDK 1.2V which contains only static methods (Constructor is private)

-----  
Methods of Collection interface :

- a) `public boolean add(E element)` :- It is used to add an item/element in the 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 :

---

List is the sub interface of Collection interface available from JDK 1.2V.

List interface accepts duplicate elements.

It stores the elements based on the order (index wise) hence It supports all the index related methods.

By using Collections.sort(List list) we can perform sorting operation on List interface.

---

List interface hierarchy : 28-Aug-24

---

Behaviour of List interface Specific classes :

---

- \* It stores the elements on the basis of index.
  - \* It can accept duplicate, homogeneous and heterogeneous elements.
  - \* It stores everything in the form of Object.
  - \* When we accept the collection classes without generic concept then compiler generates a warning message because It is unsafe object.
  - \* By using generic (<>) we can eliminate compilation warning and still we can take homogeneous as well as heterogeneous.(<Object>)
  - \* In list interface few classes are dynamically Growable like Vector and ArrayList.
- 

29-08-2024

---

Methods of List interface :

---

As we know List interface works on the basis of index so, in the entire List interface we will get index related Methods.

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/Retrieve the Collection Object ?

---

There are 9 ways to retrieve the Collection Object :

- 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 interface (JDK 1.0)
- 5) By using Iterator interface (JDK 1.2)
- 6) By using ListIterator interface (JDK 1.2)
- 7) By using SplIterator interface (JDK 1.8)
- 8) By using For Each Method (JDK 1.8)
- 9) By using Method Reference (JDK 1.8)

Among all these 9 ways, Enumeration, Iterator, ListIterator, SplIterator are the cursors so they can move from one direction to another direction.

---

Enumeration :

It is a predefined interface available in `java.util` package from JDK 1.0 onwards(Legacy interface).

We can use Enumeration interface to fetch or retrieve the Objects one by one from the Collection because it is a cursor.

We can create Enumeration object by using `elements()` method of the legacy Collection class.

```
public Enumeration elements();
```

Enumeration interface contains two methods :

- 1) public boolean hasMoreElements() :- It will return true if the Collection is having more elements.
- 2) public Object nextElement() :- It will return collection object so return type is Object and move the cursor to the next line.

Note :- It will only work with legacy Collections classes.

---

Iterator interface :

It is a predefined interface available in `java.util` package available from 1.2 version.

It is used to fetch/retrieve the elements from the Collection in forward direction only because it is also a cursor.

```
public Iterator iterator();
```

Example :

```
Iterator itr = vector.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 :

-----  
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

-----  
Spliterator interface :

-----  
It is a predefined interface available in java.util package from java 1.8 version.

It is a cursor through which we can fetch the elements from the Collection [Collection, array, Stream]

It is the combination of hasNext() and next() method.

It is using forEachRemaining(Consumer <T>) method for fetching the elements.

-----  
By using forEach() method :

-----  
From java 1.8 onwards every collection class provides a method forEach() method, this method takes Consumer functional interface as a parameter.  
This method is available in java.lang.Iterable interface.

-----  
The following program explain how to retrieve the Collection object by using 9 ways :

```

package com.ravi.collection;

import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Spliterator;
import java.util.Vector;

public class RetrieveCollectionObject {

    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.add("KIWI");

        System.out.println("BY USING TOSTRING METHOD");
        System.out.println(fruits.toString());

        System.out.println("BY USING ORDINARY FOR LOOP");

        for(int i=0; i<fruits.size(); i++)
        {
            System.out.println(fruits.get(i));
        }

        System.out.println("BY 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> lstItr = fruits.listIterator();

        System.out.println("IN FORWARD DIRECTION");
        while(lstItr.hasNext())
        {
            System.out.println(lstItr.next());
        }

        System.out.println("IN BACKWARD DIRECTION");
    }
}

```

```

        while(lstItr.hasPrevious())
        {
            System.out.println(lstItr.previous());
        }

        System.out.println("BY USING SPLITERATOR INTERFACE");
        Spliterator<String> splt = fruitsspliterator();
        splt.forEachRemaining(fruit -> System.out.println(fruit));

        System.out.println("BY USING FOR EACH METHOD");
        fruits.forEach(fruit -> System.out.println(fruit));

        System.out.println("By Using Method Reference ");
        fruits.forEach(System.out::println);
    }
}

```

-----  
30-08-2024

-----  
How forEach(Consumer<T> cons) is working internally ?

-----  
Case 1 :

```

package com.ravi.collection;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternal
{
    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.add("KIWI");

        //Anonymous inner class
        Consumer<String> cons = new Consumer<String>()
        {
            @Override
            public void accept(String t)
            {
                System.out.println(t);
            }
        };

        fruits.forEach(cons);
    }
}

```

-----  
Case 2 :

```

package com.ravi.collection;

import java.util.Vector;

```

```
import java.util.function.Consumer;

public class ForEachMethodInternal
{
    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.add("KIWI");

        //Lambda
        Consumer<String> cons =(str)-> System.out.println(str);

        fruits.forEach(cons);
    }
}
```

-----  
Case 3 :

```
-----
package com.ravi.collection;

import java.util.Vector;
import java.util.function.Consumer;

public class ForEachMethodInternal
{
    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.add("KIWI");

        fruits.forEach(fruit -> System.out.println(fruit));
    }
}
```

-----  
Vector<E> :

```
-----
public class Vector<E> extends AbstractList<E> implements List<E>,
Serializable, Clonable, RandomAccess
```

Vector is a predefined class available in java.util package under List interface.

Vector is always from java means it is available from jdk 1.0 version.

It can accept duplicate, null, homogeneous as well as heterogeneous elements.

Vector and Hashtable, these two classes are available from jdk 1.0, remaining

Collection classes were added from 1.2 version. That is the reason Vector and Hashtable are called legacy(old) classes.

The main difference between Vector and ArrayList is, ArrayList methods are not synchronized so multiple threads can access the method of ArrayList where as on the other hand most the methods are synchronized in Vector so performance wise Vector is slow.

\*We should go with ArrayList when ThreadSafety is not required on the other hand we should go with Vector when we need ThreadSafety for 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

-----  
package com.ravi.iq;

```
import java.util.Collections;
import java.util.Vector;
```

```
public class VectorExample {
```

```
    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.add("KIWI");
```

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

```
        fruits.remove(3); //removing based on the index
        fruits.remove("KIWI"); //Removing based on the Object
```

```
        System.out.println("After removing :" +fruits);
```

```
        System.out.println("Is list empty :" +fruits.isEmpty());
```

```

        System.out.println("Size of the list :" + fruits.size());
        Collections.sort(fruits);
        fruits.forEach(System.out::println);
    }

}

-----
//Vector Program on capacity

package com.ravi.vector;

import java.util.*;

public class VectorDemo1 {
    public static void main(String[] args)
    {
        Vector<Integer> v = new Vector<>(100 ,5);
        System.out.println("Initial capacity is :" + v.capacity());

        for (int i = 0; i < 100; i++)
        {
            v.add(i);
        }

        System.out.println("After adding 100 elements  capacity is :" + v.capacity());

        v.add(101);
        System.out.println("After adding 101th elements  capacity is :" + v.capacity()); // 200

        for(Integer i : v)
        {
            System.out.print(i+"\t");

            if(i%5==0)
            {
                System.out.println();
            }
        }

    }

}

-----
package com.ravi.collection;

import java.util.Vector;

//To work with Custom Object

record Employee(Integer empId, String empName)
{

}

public class VectorDemo2 {

    public static void main(String[] args)

```

```

    {
        Vector<Employee> listOfEmployees = new Vector<>();
        listOfEmployees.add(new Employee(444, "Aryan"));
        listOfEmployees.add(new Employee(111, "Raj"));
        listOfEmployees.add(new Employee(222, "Zuber"));
        listOfEmployees.add(new Employee(333, "Scott"));

        listOfEmployees.forEach(System.out::println);
    }

}

=====
31-08-2024
-----
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();
}
-----
package com.ravi.vector;
//Array To Collection
import java.util.*;
public class VectorDemo3
{
    public static void main(String args[])
    {
        Vector<Integer> v = new Vector<>();

        int x[]={22,20,10,40,15,58};

        //Adding array values to Vector
        for(int i=0; i<x.length; i++)
        {
            v.add(x[i]);
        }
        Collections.sort(v);
        System.out.println("Maximum element is :" + Collections.max(v));
        System.out.println("Minimum element is :" + Collections.min(v));
        System.out.println("Vector Elements :");
        v.forEach(y -> System.out.println(y));
        System.out.println(".....");
        Collections.reverse(v);
        v.forEach(y -> System.out.println(y));
    }
}
-----
```

Program that shows performance wise Vector is not good in comparison to ArrayList

System is a predefined class available in java.lang package and it contains a predefined static method currentTimeMillis() , the return type of this method is long, actually it returns the current time of the system in ms.

```

public static native long currentTimeMillis()

//Program to describe that ArrayList is better than Vector in performance
```

```

package com.ravi.vector;

import java.util.ArrayList;
import java.util.Vector;

public class VectorDemo4
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<>();
        long startTime = System.currentTimeMillis();
        for(int i = 0; i<=1000000; i++)
        {
            al.add(i);
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time Taken by ArrayList :"+(endTime-startTime)+" ms");
        Vector<Integer> v1 = new Vector<>();
        startTime = System.currentTimeMillis();
        for(int i = 0; i<=1000000; i++)
        {
            v1.add(i);
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time Taken by Vector :"+(endTime-startTime)+" ms");
    }
}
-----  

package com.ravi.vector;

import java.util.Vector;

public class VectorDemo6 {

    public static void main(String[] args)
    {
        Vector<String> listOfCity = new Vector<>();
        listOfCity.add("Surat");
        listOfCity.add("Pune");
        listOfCity.add("Ahmadabad");
        listOfCity.add("Vanaras");

        listOfCity.sort(String::compareTo);

        listOfCity.forEach(System.out::println);

        Vector<Integer> listOfNumbers = new Vector<>();
        listOfNumbers.add(500);
        listOfNumbers.add(400);
        listOfNumbers.add(300);
        listOfNumbers.add(200);
    }
}

```

```

        listOfNumbers.add(100);

        listOfNumbers.sort(Integer::compareTo);

        listOfNumbers.forEach(System.out::println);

        //Collection to array toArray()
        Object[] array = listOfNumbers.toArray();
        System.out.println("Using Array");
        for(Object value : array)
        {
            System.out.println(value);
        }

    }

}
-----
```

**Stack<E>**

---

**public class Stack<E> extends Vector<E>**

It is a predefined class available in `java.util` package. It is the sub class of `Vector` class introduced from JDK 1.0 so, It is also a legacy class.

It is a linear data structure that is used to store the Objects in LIFO (Last In first out) order.

Inserting an element into a Stack is known as push operation where as extracting an element from the top of the stack is known as pop operation.

It throws an exception called `java.util.EmptyStackException`, if Stack is empty and we want to fetch the element.

It has only one constructor as shown below

```
Stack s = new Stack();
```

---

**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(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<E> :

```

-----
public class ArrayList<E> extends AbstractList<E> implements List<E>,
Serializable, Clonable, RandomAccess
```

It is a predefined class available in `java.util` package under `List` interface from `java 1.2v`.

It accepts duplicate 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  

$$\text{new capacity} = (\text{current capacity} * 3)/2 + 1 \quad [\text{Almost 50\% increment}]$$

\*All the methods declared inside an `ArrayList` is not synchronized so multiple thread can access the method of `ArrayList`.

\*It is highly suitable for fetching or 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)

-----  
02-09-2024

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

        al.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));
    }
}

-----
How to create fixed length and Immutable object :
-----
a) Creating a fixed length array by using asList():
-----
Arrays class has provided a predefined static method called asList(T ...x), by
using this asList() method we create a fixed length array

```

In this fixed length array we can't add any new element but we can replace the existing element with new element.

If we try to add a new element then we will get an Exception  
java.lang.UnsupportedOperationException.

```
List list = Arrays.asList(T ...x);

//Program:
-----
package com.ravi.arraylist;

import java.util.Arrays;
import java.util.List;

public class ImmutableList {

    public static void main(String[] args)
    {
        List<String> listOfNames =
Arrays.asList("Scott", "Smith", "Martin", "John");

        listOfNames.forEach(System.out::println);
        //listOfNames.add("Virat"); //Invalid
java.lang.UnsupportedOperationException

        listOfNames.set(0, "Virat");
        listOfNames.forEach(System.out::println);

    }
}
```

b) Creating an immutable List by using List.of(E ...x)

-----  
List interface has provided a static method called of(E ...x) which creates an immutable List. We can't perform any add or replace operation otherwise we will get java.lang.UnsupportedOperationException.

```
List list = List.of(E...x)

package com.ravi.arraylist;

import java.util.List;

public class ImmutableList1 {

    public static void main(String[] args)
    {
        List<String> listOfString = List.of("A", "B", "C");
        System.out.println(listOfString);

        listOfString.add("E"); //Invalid
        listOfString.set(0, "D"); //Invalid
    }
}

//Program to fetch the elements in forward and backward
//direction using ListIterator interface
```

```

package com.ravi.arraylist;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;

public class ArrayListDemo3
{
    public static void main(String args[])
    {
        List<String> listOfName =
        Arrays.asList("Rohit", "Akshar", "Pallavi", "Sweta"); //Length is fixed

        listOfName.sort(String::compareTo);

        //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
package com.ravi.arraylist;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class ArrayListDemo4
{
    public static void main(String[] args) throws IOException
    {
        ArrayList<String> cityName = new ArrayList<>();
        cityName.add("Hyderabad");
        cityName.add("Pune");
        cityName.add("Banglore");
        cityName.add("Chennai");

        //Serialization
        var fout = new FileOutputStream("C:\\new\\city.txt");
        var oos = new ObjectOutputStream(fout);

        try(fout;oos)
        {
            oos.writeObject(cityName);
            System.out.println("Object Data stored....");
        }
        catch(Exception e)
    }
}

```

```

        {
            e.printStackTrace();
        }

        var fin = new FileInputStream("C:\\\\new\\\\city.txt");
        var ois = new ObjectInputStream(fin);

        try(fin;ois)
        {
            ArrayList<String> list = (ArrayList<String>)ois.readObject();
            System.out.println(list);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

**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 Hyd null 11
}
-----
package com.ravi.arraylist;

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

record Professor(String name, String 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)
    {

        Department csd = new Department("Computer Science");

        Professor prof1 = new Professor("Scott", "Java");
        Professor prof2 = new Professor("Rahul", "Python");
        Professor prof3 = new Professor("Samir", ".Net");
    }
}

```

```

        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() + ":");

        List<Professor> professors = csd.getProfessors();
        professors.forEach(System.out::println);

    }

}

-----

```

Time Complexity of ArrayList :

The time complexity of ArrayList to insert and delete an element from the middle would be O(n) [Big O of n] because 'n' number of elements will be re-located so it is not a good choice to perform insertion and deletion operation in the middle of the List.

On the other hand time complexity of ArrayList to retrieve an element from the List would be O(1) because by using get(int index) method we can retrieve the element randomly from the list. ArrayList class implements RandomAccess marker interface which provides the facility to fetch the elements Randomly. [02-SEP]

03-09-2024

-----

LinkedList :

-----  
**public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable**

It is a predefined class available in java.util package under List interface from JDK 1.2v.

It is ordered by index position like ArrayList except the elements (nodes) are doubly linked to one another. This linkage provide us new method for adding and removing the elements from the middle of LinkedList.

\*The important thing is, LikedList may iterate more 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.

It inserts the elements by using Doubly linked List so insertion and deletion is very easy.

ArrayList is using Dynamic array data structure but LinkedList class is using LinkedList (Doubly Linked List) data structure.

At the time of searching an element, It will start searching from first(Head) node or last node OR the closer one.

\*\*Here Iterators are Fail Fast Iterator.

Constructor:

-----

It has 2 constructors

- 1) `LinkedList list1 = new LinkedList();`  
It will create a `LinkedList` object with 0 capacity.
- 2) `LinkedList list2 = new LinkedList(Collection c);`  
Interconversion between the collection

Methods of `LinkedList` class:

- 
- 1) `void addFirst(Object o)`
  - 2) `void addLast(Object o)`
  - 3) `Object getFirst()`
  - 4) `Object getLast()`
  - 5) `Object removeFirst()`
  - 6) `Object removeLast()`

Note :- It stores the elements in non-contiguous memory location.

The time complexity for insertion and deletion is  $O(1)$

The time complexity for searching  $O(n)$  because it search the elements using node reference.

-----

\*\*\*\*What is Fail Fast Iterator :

Whenever we are working with Traditional Collection and If we fetching the elements by using Iterator, At the time of fetching the elements if there is a change or modification in collection structure then we will get `java.util.ConcurrentModificationException`

```
package com.ravi.fails_fast;

import java.util.Iterator;
import java.util.Spliterator;
import java.util.concurrent.CopyOnWriteArrayList;

class Concurrent extends Thread
{
    private CopyOnWriteArrayList<String> arl;

    public Concurrent(CopyOnWriteArrayList<String> arl)
    {
        super();
        this.arl = arl;
    }

    @Override
    public void run()
    {
        try
        {
            Thread.sleep(2000);
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }

        arl.add("Kiwi");
    }
}
```

```

        }

}

public class FailFastDemo {
    public static void main(String[] args) throws InterruptedException
    {
        CopyOnWriteArrayList<String> fruits = new CopyOnWriteArrayList<>();
        fruits.add("Orange");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Banana");
        fruits.add("Gauva");

        Concurrent cm = new Concurrent(fruits);
        cm.start();

        Iterator<String> iterator = fruits.iterator();

        while(iterator.hasNext())
        {
            System.out.println(iterator.next());
            Thread.sleep(500);
        }
        System.out.println(".....");
    }
}

```

04-09-2024

```

-----
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 (add(), set(), remove())
import java.util.*;
public class LinkedListDemo3
{
    public static void main(String[] args)
    {
        LinkedList<String> city = new LinkedList<> ();
        city.add("Kolkata");
        city.add("Bangalore");
        city.add("Hyderabad");
        city.add("Pune");
        System.out.println(city);

        ListIterator<String> lt = city.listIterator();

        while(lt.hasNext())
        {
            String cityName = lt.next();

            if(cityName.equals("Kolkata"))
            {
                lt.remove();
            }
            else if(cityName.equals("Hyderabad"))
            {
                lt.add("Ameerpet");
            }
            else if(cityName.equals("Pune"))
            {
                lt.set("Mumbai");
            }
        }
        city.forEach(System.out::println);
    }
}

```

Here there is no ConcurrentModificationException because ListIterator is modifying the structure by it's own method hence there is no problem because it is internal structure modification.

```

-----
package com.ravi.linked_list;

import java.util.Arrays;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Stream;

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);
    }
}

-----//Insertion, deletion, displaying and exit

import java.util.LinkedList;
import java.util.Scanner;

public class LinkedListDemo4
{
    public static void main(String[] args)
    {
        LinkedList<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 removed =
linkedList.remove(Integer.valueOf(elementToDelete));

                        if (removed)
                        {
                            System.out.println("Element " + elementToDelete + "
deleted from the linked list.");
                        }
                        else
                        {
                            System.out.println("Element not found in the linked
list.");
                        }
                    }
                    break;
                case 3:
                    System.out.println("Elements in the linked

```

```

list.");
        linkedList.forEach(System.out::println);
        break;
    case 4:
        System.out.println("Exiting the program.");
        scanner.close();
        System.exit(0);
    default:
        System.out.println("Invalid choice. Please try again.");
    }
}
}

-----import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

record Dog(String name)
{
}

public class LinkedListDemo5
{
    public static void main(String[] args)
    {
        List<Dog> list = new LinkedList<>();
        Dog dog = new Dog("Tiger");
        list.add(dog);
        list.add(new Dog("Tommy"));
        list.add(new Dog("Rocky"));

        Iterator<Dog> i3 = list.iterator();
        i3.forEachRemaining(x ->
System.out.println(x.name().toUpperCase())); //java 8

        System.out.println("size " + list.size());
        System.out.println("Get 1st Position Object " +
list.get(1).name());
    }
}

-----import java.util.Deque;
import java.util.LinkedList;

public class LinkedListDemo6
{
    public static void main(String[] args)
    {
        // Create a LinkedList and treat it as a Deque
        Deque<String> deque = new LinkedList<>();

        deque.addFirst("Ravi"); // Ravi Pallavi
        deque.addFirst("Raj");

        deque.addLast("Pallavi");
        deque.addLast("Sweta");
    }
}

```

```
        System.out.println("Deque: " + deque);

        String first = deque.removeFirst();
        String last = deque.removeLast();

        System.out.println("Removed first element: " + first);
        System.out.println("Removed last element: " + last);
        System.out.println("Updated Deque: " + deque);
    }
}
-----
```

Set interface :

Set interface is the sub interface of Collection interface available JDK 1.2v.

Set interface never accept duplicate elements, here internally it uses equals(Object obj) method to eliminte the duplicate element.

IT DOES NOT MAINTAIN ANY ORDER.(No Index)

It is unordered so ListIterator interface will not work with Set interface.

Set interface :

It is the sub interface of Collection interface available from JDK 1.2

It does not store the element on the basis of index.

It does not accept duplicate element here intrenally equals(Object obj) method verifies the objects, If two objects are identical then it will accept only one object.

ListIterartor interface does not work with Set interface.

It supports all the methods of Collection interface, some methods are added from java 9v.

Set interface Hierarchy :

Hierarchy is available in the paint diagram [04-SEP-24]

What is hashing algorithm ?

Hashing algorithm is a technique through which we can search, insert and delete an element in more efficient way in comparison to our classical indexing approach.

Hashing algorithm, internally uses Hashtable data structute, Hashtable data structure internally uses Bucket data structure.

Here elements are inserted by using hashing algorithm so the time complaxcity to insert, delete and search an element would be O(1)

05-09-2024

HashSet (UNSORTED, UNORDERED , NO DUPLICATES)

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Clonabale, Serializable

It is a predefined class available in java.util package under Set interface and introduced from JDK 1.2V.

It is an unsorted and unordered set.

It accepts heterogeneous and homogeneous both kind of data.

\*It uses the hashCode of the object being inserted into the Collection. Using this hashCode it finds the bucket location.

It doesn't contain any duplicate elements as well as It does not maintain any order while iterating the elements from the collection.

It can accept one null value.

HashSet methods are not synchronized.

HashSet is used for fast searching operation.

It contains 4 types of constructors

1) HashSet hs1 = new HashSet();

    It will create the HashSet Object with default capacity is 16. The default load factor 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));

}

}

-----  
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));

    }

}

-----  

add() method return type is boolean

package com.ravi.collection;

import java.util.Arrays;
import java.util.HashSet;

public class HashSetDemo2 {

    public static void main(String[] args)
    {
        Boolean arr[] = new Boolean[5];

        HashSet<Object> hs = new HashSet<>();
        arr[0] = hs.add(12);
        arr[1] = hs.add(new String("A"));
        arr[2] = hs.add(12);
        arr[3] = hs.add(null);
        arr[4] = hs.add("A");

        System.out.println(Arrays.toString(arr));

        if(hs.contains("A"))
        {
            System.out.println("Object A is available");
        }
        else
        {
            System.err.println("Object is not available");
        }

        hs.forEach(System.out::println);
    }

}

-----  

//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();
}

}
}
-----
LinkedList<E> [It maintains order]
-----
public class LinkedHashSet extends HashSet implements Set, Clonable,
Serializable

```

It is a predefined class in `java.util` package under `Set` interface and introduced from java 1.4v.

It is the sub class of `HashSet` class.

It is an 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.*;  
  
public class LinkedHashSetDemo1  
{  
    public static void main(String[] args)  
    {  
        LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();  
  
        linkedHashSet.add(10);  
        linkedHashSet.add(5);  
        linkedHashSet.add(15);  
        linkedHashSet.add(20);  
        linkedHashSet.add(5);  
  
        System.out.println("LinkedHashSet elements: " + linkedHashSet);  
        System.out.println("LinkedHashSet size: " + linkedHashSet.size());  
  
        int elementToCheck = 15;  
        if (linkedHashSet.contains(elementToCheck))  
        {  
            System.out.println(elementToCheck + " is present in the  
LinkedHashSet.");  
        }  
        else  
        {  
            System.out.println(elementToCheck + " is not present in the  
LinkedHashSet.");  
        }  
  
        int elementToRemove = 10;  
        linkedHashSet.remove(elementToRemove);  
        System.out.println("After removing " + elementToRemove + ",  
LinkedHashSet elements: " + linkedHashSet);  
  
        linkedHashSet.clear();  
        System.out.println("After clearing, LinkedHashSet elements: " +  
linkedHashSet); //[]  
    }  
}
```

```
}
```

```
-----  
SortedSet interface :
```

```
-----  
As we know Collections.sort(List list) method accept list as a parameter so, we  
can't perform sorting operation by using sort() method on HashSet and  
LinkedHashSet.
```

```
In order to provide automatic sorting facility, Set interface has provided one  
more interface i.e SortedSet interface available from JDK 1.2.
```

```
SortedSet interface provided default natural sorting order, default natural  
sorting order means, if it is number then ascending order but if it is String  
then alphabetical Or dictionary order.
```

```
In order to sort the element either in default natural sorting order or user-  
defined sorting order we are using Comparable or Comparator  
interfaces.
```

```
****Difference between Comparable and Comparator :
```

```
-----  
Difference is available in paint diagram :
```

```
-----  
Program on Comparable interface :
```

```
-----  
2 Files :
```

```
-----  
Employee.java(R)
```

```
-----  
package com.ravi.comparable;
```

```
public record Employee(Integer employeeId, String employeeName) implements  
Comparable<Employee>  
{  
    @Override  
    public int compareTo(Employee e2)  
    {  
        return this.employeeId.compareTo(e2.employeeId);  
    }
```

```
}
```

```
-----  
EmployeeComparable.java(C)
```

```
-----  
package com.ravi.comparable;
```

```
import java.util.ArrayList;  
import java.util.Collections;
```

```
public class EmployeeComparable {
```

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

```
        ArrayList<Employee> listOfEmployees = new ArrayList<>();  
        listOfEmployees.add(new Employee(222, "Raj"));  
        listOfEmployees.add(new Employee(333, "Aryan"));  
        listOfEmployees.add(new Employee(111, "Zuber"));
```

```
        //Sorting the Employee based on the ID  
        Collections.sort(listOfEmployees);
```

```
        listOfEmployees.forEach(System.out::println);
```

```
}

}
=====
06-09-2024
```

```
-----  
Limitation of Comparable interface :
```

- ```
-----  
1) We can write only one sorting logic using comparable (We can sort based on ID  
or based on name but not both)  
  
2) If the BLC class is given by some 3rd party in .class format  
then we can't modify the source code hence Comparable will not work here.
```

```
To avoid the above said problems we introduced Comparator interface :
```

```
-----  
Program on Comparator :
```

```
-----  
2 files :
```

```
-----  
Product.java(R)
```

```
-----  
package com.ravi.comparator;
```

```
public record Product(Integer productId, String productName)  
{
```

```
}
```

```
ProductComparator.java
```

```
-----  
package com.ravi.comparator;
```

```
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;
```

```
public class ProductComparator  
{
```

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

```
        ArrayList<Product> listofProduct = new ArrayList<>();  
        listofProduct.add(new Product(333, "Laptop"));  
        listofProduct.add(new Product(111, "Mobile"));  
        listofProduct.add(new Product(222, "Camera"));
```

```
        Comparator<Product> compId = new Comparator<Product>()  
        {
```

```
            @Override  
            public int compare(Product p1, Product p2)  
            {  
                return p1.productId() - p2.productId();  
            }
```

```
        };  
        Collections.sort(listofProduct, compId);  
        System.out.println("Sorted Based on the ID");  
        listofProduct.forEach(System.out::println);
```

```
        System.out.println("Sort Based on the Product Name :");
```

```
        Comparator<Product> compName = (p1, p2)->  
        p1.productName().compareTo(p2.productName());
```

```
        Collections.sort(listOfProduct, compName);
        listOfProduct.forEach(System.out::println);
    }
}
```

Comparable is not a Functional interface because due to current object support(this keyword) we need to write the sorting logic in the BLC class only so BLC class must implements Comparable interface.

[We can say Comparable as a Functional interafce because it contains exactly one abstract method but we can't uas as Lambda]

We can't write the logic of compareTo(T x) method in another class because It accept only one parameter for second parameter we need to depend upon the Current Object.

Program to sort the Integer object in descending order :

```
package com.ravi.comparator;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
```

```
public class DescendingOrder {
```

```
    public static void main(String[] args)
    {
        ArrayList<Integer> listOfNumbers = new ArrayList<>();
        listOfNumbers.add(99);
        listOfNumbers.add(89);
        listOfNumbers.add(15);
        listOfNumbers.add(88);
        listOfNumbers.add(9);
        listOfNumbers.add(2);

        Comparator<Integer> desc = (i1, i2)-> i2.compareTo(i1);

        Collections.sort(listOfNumbers, desc);

        listOfNumbers.forEach(System.out::println);
    }
}
```

List intreface sort() method :

List interface has provided sort(Comparator<t> cmp) as a parameter as shown in the program

```
package com.ravi.comparator;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
```

```
public class DescendingOrder {
```

```
    public static void main(String[] args)
    {
        ArrayList<Integer> listOfNumbers = new ArrayList<>();
```

```

        listOfNumbers.add(99);
        listOfNumbers.add(89);
        listOfNumbers.add(15);
        listOfNumbers.add(88);
        listOfNumbers.add(9);
        listOfNumbers.add(2);

        listOfNumbers.sort((i1,i2)-> i2.compareTo(i1));

        listOfNumbers.forEach(num -> System.out.println(num));

    }
}
-----
```

**TreeSet<E>**

```
-----  
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>,  
Cloneable, 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`)

It does not accept non comparable type of data if we try to insert it will throw a runtime exception i.e `java.lang.ClassCastException`

`TreeSet` implements `NavigableSet`.

`NavigableSet` extends `SortedSet`.

It contains 4 types of constructors :

```
-----  
1) TreeSet t1 = new TreeSet();  
    create an empty TreeSet object, elements will be inserted in using  
Comparable.
```

```
2) TreeSet t2 = new TreeSet(Comparator c);  
    Customized sorting order
```

```
3) TreeSet t3 = new TreeSet(Collection c);  
    loose coupling
```

```
4) TreeSet t4 = new TreeSet(SortedSet s);  
    Another implemented class of SortedSet we can pass as a parameter.  
-----
```

```
import java.util.*;  
public class Main  
{  
    public static void main(String [] args)  
    {  
        TreeSet ts = new TreeSet();  
        ts.add(19);  
        ts.add(15);  
        ts.add(11);
```

```

        ts.add(20);
        ts.add("Ravi");

        System.out.println(ts);
    }

}

```

Note : We will get java.lang.ClassCastException.

```

-----  

//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 : Here Comparable interface is working to sort the elements in default natural sorting order using compareTo(T x) method.

```

-----  

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<String> descendingIterator()
-----
package com.ravi.treeset;

import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetDemo2 {

    public static void main(String[] args)
    {
        Set<String> t = new TreeSet<>((s1,s2)-> s2.compareTo(s1));
        t.add("6");
        t.add("5");
        t.add("4");
        t.add("2");
        t.add("9");
        Iterator<String> iterator = t.iterator();
        iterator.forEachRemaining(x -> System.out.println(x));
    }
}
```

sorting the elements using Comparator.

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

public class TreeSetDemo3
{
    public static void main(String[] args)
    {
        Set<Character> t = new TreeSet<>(Character::compareTo);
        t.add('A');
        t.add('C');
        t.add('B');
        t.add('E');
        t.add('D');
        Iterator<Character> iterator = t.iterator();
        iterator.forEachRemaining(x -> System.out.println(x));
    }
}
```

sorting the elements using Comparator.

```
-----
package com.ravi.treeset;

import java.util.TreeSet;

record Customer(Integer customerId, String custName, Double custBill) implements Comparable<Customer>
{
    @Override
    public int compareTo(Customer c2)
    {
        return customerId.compareTo(c2.customerId);
    }
}
```

```

}

public class TreeSetDemo4 {
    public static void main(String[] args)
    {
        TreeSet<Customer> ts1 = new TreeSet<>();
        ts1.add(new Customer(333, "Satish", 23789.90));
        ts1.add(new Customer(111, "Zuber", 25789.90));
        ts1.add(new Customer(222, "Raj", 29789.90));
        ts1.add(new Customer(444, "Rahul", 21789.90));
        ts1.add(new Customer(555, "Aryan", 22789.90));

        ts1.forEach(System.out::println);
    }
}

-----  

package com.ravi.treeset;

import java.util.TreeSet;

record Product(Integer productId, String productName, Double productPrice)
{
}

public class TreeSetDemo5
{
    public static void main(String[] args)
    {
        TreeSet<Product> products = new TreeSet<Product>((p1,p2)->
p1.productPrice().compareTo(p2.productPrice()));
        products.add(new Product(333, "Camera", 86000.90));
        products.add(new Product(111, "Mobile", 89000.90));
        products.add(new Product(222, "Laptop", 82000.90));

        products.forEach(System.out::println);
    }
}

-----  

//Sort the Student data based on different criteria

package com.ravi.treeset;

import java.util.TreeSet;

record Student(Integer studentId, String studentName, Integer studentAge)
{
}

public class TreeSetDemo6 {
    public static void main(String[] args)
    {
        TreeSet<Student> ts1 = new TreeSet<Student>((s1,s2)->
s1.studentId().compareTo(s2.studentId()));
        ts1.add(new Student(333,"Raj",22));
        ts1.add(new Student(222,"Ratan",19));
        ts1.add(new Student(111,"Purvi",20));
    }
}

```

```

        ts1.add(new Student(444, "Ahana", 21));

        System.out.println("Sorting in Ascending order based on the ID");
        ts1.forEach(System.out::println);

        System.out.println("=====");
        TreeSet<Student> ts2 = new TreeSet<Student>((s1, s2)->
s2.studentId().compareTo(s1.studentId()));
        ts2.add(new Student(333, "Raj", 22));
        ts2.add(new Student(222, "Ratan", 19));
        ts2.add(new Student(111, "Purvi", 20));
        ts2.add(new Student(444, "Ahana", 21));

        System.out.println("Sorting in Descending order based on the ID");
        ts2.forEach(System.out::println);

        System.out.println("=====");
        TreeSet<Student> ts3 = new TreeSet<Student>((s1, s2)->
s1.studentName().compareTo(s2.studentName()));
        ts3.add(new Student(333, "Raj", 22));
        ts3.add(new Student(222, "Ratan", 19));
        ts3.add(new Student(111, "Purvi", 20));
        ts3.add(new Student(444, "Ahana", 21));

        System.out.println("Sorting in Dictionary order based on the
Name :");
        ts3.forEach(System.out::println);

    }
}

```

-----  
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]
    }
}
-----
```

10-09-2024

NavigableSet :

It is a predefined interface available in java.util package from JDK 1.6v

It is used to navigate among the elements, Unlike SortedSet which provides range of value. Here we can navigate among the values as shown below.

```

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 :

Hierarchy is available in paint daigram [10-SEP-24]

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

Map interface has provided few methods to convert the map into collection which are known as Collection Views method.

Methods of map interface to convert the map into Collection :

We have map interface methods through which we can convert map interface into collection interface which is known as collection views method.

- 1) public Set<Object> keySet() : It will retrieve all the keys.
- 2) public Collection values() : It will retrieve all the values.
- 3) public Set<Map.Entry> entrySet() : It will retrieve key and value both in a single object.

\*\*\*\* 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 in the same Bucket by using Singly LinkedList

Note :- equals(Object obj) method is invoked only when two keys are having same hashCode as well as their references are different.

- e) Actually by calling hashCode method we are not comparing the objects, we are just storing the objects in a group so the currently adding key object will be compared with its SAME HASHCODE GROUP objects, but not with all the keys which are available in the Map.

- f) The main purpose of storing objects into the corresponding group to decrease the number of comparison so the efficiency of the program will increase.

- g) To insert an entry in the HashMap, HashMap internally uses Hashtable data structure.

- h) Now, for storing same hashCode object into a single group, hash table data structure internally uses one more data structure called Bucket.

i) The Hashtable data structure internally uses Node class array object.

j) The bucket data structure internally uses LinkedList data structure, It is a single linked list again implemented by Node class only.

\*k) A bucket is group of entries of same hash code keys.

l) Performance wise LinkedList is not good to search, so from java 8 onwards LinkedList is changed to Binary tree to decrease the number of comparison within the same bucket hashCode if the number of entries are greater than 8.

---

\*\* equals() and hashCode() method contract :

---

Both the methods are working together to find out the duplicate objects in the Map.

\*If equals() method invoked on two objects and it returns true then hashCode of both the objects must be same.

Note : IF TWO OBJECTS ARE HAVING SAME HASH CODE THEN IT MAY BE SAME OR DIFFERENT BUT IF EQUALS(OBJECT OBJ) METHOD RETURN TRUE THEN BOTH OBJECTS MUST RETURN SAME HASHCODE.

---

```

package com.ravi.map;

import java.util.HashMap;

public class HashMapInternalDemo
{
    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);
}

```

---

What will happen if we don't follow the contract ?  
Case 1 :

---

```
If we override only equals(Object obj)
-----
If we override only equals(Object obj) method for content comparison
then same object will have different hashCode (due to new keyword) hence same
object (content wise) will move into two different
buckets [Duplication].
```

Case 2 :

```
-----
If we override only hashCode() method
```

```
-----
If we overrdie only hashCode() method then two objects which are having same
hashCode (due to overriding) will go to same bucket but == operator and
equals(Object obj) method of Object class, both will return false hence
duplicate object will be inserted into same bucket.
```

So, the conclusion is, compulsory we need to override both the methods for removing duplicate elements.

```
package com.ravi.map;

import java.util.HashMap;
import java.util.Objects;

class Customer
{
    private int customerId;
    private String customerName;

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

    @Override
    public int hashCode()
    {
        return Objects.hash(customerId, customerName);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Customer other = (Customer) obj;
        return customerId == other.customerId &&
    Objects.equals(customerName, other.customerName);
    }
}

public class HashMapInternalDemo
{

    public static void main(String[] args)
    {
        Customer c1 = new Customer(111, "Scott");
        Customer c2 = new Customer(111, "Scott");
    }
}
```

```

        //System.out.println(c1.hashCode()+" : "+c2.hashCode());

        HashMap<Customer, String> map = new HashMap<>();
        map.put(c1, "A");
        map.put(c2, "B");

        System.out.println(map.size());
    }

}

```

Customer class we are using as a HashMap key so it must override hashCode() and equals(Object obj) as well as at advanced level, It must be immutable class.

All the Wrapper classes and String class are immutable as well as hashCode() and equals(Object obj) methods are overridden in these classes so perfectly suitable to becoming HashMap key.

---

```

package com.ravi.map;

import java.util.HashMap;

record Customer(Integer id, String name)
{
}

public class HashMapInternalDemo
{

    public static void main(String[] args)
    {
        Customer c1 = new Customer(111, "Scott");
        Customer c2 = new Customer(111, "Scott");

        //System.out.println(c1.hashCode()+" : "+c2.hashCode());

        HashMap<Customer, String> map = new HashMap<>();
        map.put(c1, "A");
        map.put(c2, "B");

        System.out.println(map.size());
    }

}

```

so final conclusion is, In our user-defined class which we want to use as a HashMap key must be immutable and hashCode() and equals(Object obj) method must be overridden. Instead of BLC class we can also use simply record because record is implicitly final and hashCode() and equals(Object obj) method overridden.

---

How to generate CustomHashCode for String value.

---

```

package com.ravi.map;

import java.util.Scanner;

public class CustomStringHashCode
{
    public static int customHashCode(String str)
    {

```

```

        if (str == null)
        {
            return 0; // defualt hashCode value for null is 0
        }

        int hashCode = 0;

        for (int i = 0; i < str.length(); i++) //NIT
        {
            char charValue = str.charAt(i);
            hashCode = 31 * hashCode + charValue;
        }

        return hashCode;
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your String :");
        String str = sc.next();

        System.out.println(str+" hashcode from String class :" +str.hashCode());

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

        System.out.println(str+" hashcode for my
class :" +CustomStringHashCode.customHashCode(str));
        sc.close();
    }
}
=====
HashMap<K,V> :- [Unsorted, Unordered, No Duplicate keys]
-----
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,
Serializable, Clonable
```

It is a predefined class available in java.util package under Map interface available from JDK 1.2v.

It gives us unsorted and Unordered map. when we need a map and we don't care about the order while iterating the elements through it then we should use HashMap.

It inserts the element based on the hashCode of the Object key using hashing technique [hasing alogorithm]

It does not accept duplicate keys but value may be duplicate.

It accepts only one null key(because duplicate keys are not allowed) but multiple null values are allowed.

HashMap is not synchronized.

Time complexcity of search, insert and delete will be O(1)

We should use HashMap to perform fast searching opeartion.

For eliminating duplicate keys in hashMap object we should compulsory follow the contract between hashCode and equals(Object obj)

It contains 4 types of constructor

- 1) `HashMap hm1 = new HashMap();`

It will create the HashMap Object with default capacity is 16. The default load fator or Fill Ratio is 0.75 (75% of HashMap is filled up then new HashMap Object will be created having double capacity)

```
2) HashMap hm2 = new HashMap(int initialCapacity);
   will create the HashMap object with user specified capacity

3) HashMap hm3 = new HashMap(int initialCapacity, float loadFactor);
   we can specify our own initialCapacity and loadFactor(by default load factor
is 0.75)

4)HashMap hm4 = new HashMap(Map m);
   Interconversion of Map Collection
-----
//Program 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); //{{key = value}

        map.forEach((k,v)-> System.out.println("Key is :" + k + " value is
"+v));
    }
}
-----
package com.ravi.map;

import java.util.HashMap;

record Product(Integer productId, String productName, Double productPrice)
{

}

public class HashMapCustom {

    public static void main(String[] args)
    {
        HashMap<Product, String> products = new HashMap<>();
        products.put(new Product(111, "Camera", 12789.90), "Hyderabad");
        products.put(new Product(222, "Mobile", 22789.90), "Pune");
        products.put(new Product(333, "Smart TV", 32789.90), "Nagpur");
        products.put(new Product(111, "Camera", 12789.90), "Banglore");

        products.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); // [1, 2, 3, 4]

        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("C","Key is not available");
        System.out.println(value);
        System.out.println(map);
    }
}

-----  

//interconversion of two HashMap using Constructor
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);
    }
}

-----  

If we don't override hashCode() method :
-----  

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

13-09-2024

-----  
**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(iny initialCapacity);

3) LinkedHashMap hm1 = new LinkedHashMap(iny 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));
    }
}
-----
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);
    }
}
-----
WeakHashMap<K, V> :
-----
public class WeakHashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>

```

It is a predefined class in `java.util` package under `Map` interface. It was introduced from JDK 1.2v onwards.

While working with `HashMap`, keys of `HashMap` are of strong reference type. This

means the entry of map will not be deleted by the garbage collector even though the key is set to be null 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);

-----  
package com.ravi.weak\_hash\_map;

import java.util.WeakHashMap;

record Product(Integer productId, String productName)

{

    @Override

    public void finalize()

    {

        System.out.println("Object is eligible for GC");

    }

}

public class WeakHashMapDemo {

    public static void main(String[] args) throws InterruptedException

    {

        Product p1 = new Product(111, "Laptop");

        WeakHashMap<Product, String> map = new WeakHashMap<>();  
        map.put(p1, "HP Model");

        System.out.println(map);

        p1 = null;

        System.gc();

        Thread.sleep(5000);

```
        System.out.println(map); //{}
    }
}
```

-----  
How to generate System hashCode :

As we know Object class has provided hashCode() method but if we want to generate System hashCode (system generated hashCode) then System class has provided a predefined static method identityHashCode(Object obj).

```
public native static int identityHashCode(Object obj)
```

```
-----  
package com.ravi.weak_hash_map;
```

```
public class IdentityHashCodeDemo {
```

```
    public static void main(String[] args)
    {
        String str = "india";
        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 serach the bucket location and insert the entry their only.

So We should use IdentityHashMap where we need to check the reference or memory address instead of logical equality.

HashMap uses hashCode of the "Object key" to find out the bucket loaction in Hashtable, on the other hand IdentityHashMap does not use hashCode() method actually It uses System.identityHashCode(Object o)

IdentityHashMap is more faster than HashMap in case of key Comparison.

The default capacity is 32.

It has three constrcutors, It does not contain loadFactor specific constructor.

```
-----  
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 avaialble in java.util package under Map interface available for 1.2V.

It is a sorted map that means it will sort the elements by natural sorting order based on the key or by using Comparator interface as a constructor parameter.

It does not allow non comparable keys.

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 soting 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.treemap;

import java.util.TreeMap;

record Student(Integer studentId, String studentName) implements
Comparable<Student>
{
    @Override
    public int compareTo(Student s2)
    {
        return this.studentId.compareTo(s2.studentId);
    }
}
```

```

}

public class TreeMapDemo3
{
    public static void main(String[] args)
    {
        TreeMap<Student, String> tm = new TreeMap<>();
        tm.put(new Student(2, "Sailesh"), "Hyderabad");
        tm.put(new Student(1, "Zuber"), "BBSR");

        tm.forEach((k,v)-> System.out.println("Key is :" +k+ " and Address
is :" +v));
    }
}

```

Here, Keys are sorted based on the Comparable i.e ascending order of studentid

---

-----  
package com.ravi.treemap;

```

import java.util.TreeMap;

record Employee(Integer employeeId, String employeeName)
{



public class TreeMapDemo4 {

    public static void main(String[] args)
    {
        TreeMap<Employee, String> tm1 = new TreeMap<>((e1,e2)->
e2.employeeName().compareTo(e1.employeeName()));


        tm1.put(new Employee(333, "Raj"), "TCS");
        tm1.put(new Employee(111, "Dinesh"), "WIPRO");
        tm1.put(new Employee(222, "Aryan"), "NIT");

        tm1.forEach((k,v)-> System.out.println("Key is :" +k+ " value
is :" +v));
    }
}

```

---

-----  
package com.ravi.treemap;

```

import java.util.TreeMap;

public class TreeMapDemo5 {

    public static void main(String[] args)
    {
        TreeMap<String, Integer> map = new TreeMap<>(String::compareTo);
        map.put("raj", 1);
        map.put("ravi", 2);
        map.put("rahul", 3);
    }
}

```

```

        System.out.println(map);
    }

}

-----  

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 Method :

14-09-2024

#### Properties :

```
public class Properties extends Hashtable<K,V>
```

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 (load method).

Properties class has provided a non static method load(InputStream/Reader r) through which we can load the properties file.

It has also provided non static method getProperty(String key) to get the value based on the specified key.

System class has provided a predefined static method properties() through which we can get the complete properties of a System/laptop.

-----  
Steps to work with Properties class :

-----  
Step 1 :

-----  
Create one properties file, fileName.properties [Extension must be .properties]

Store some data in the form of key and value pair in String format as shown below.

db.properties

-----  
driver = oracle.jdbc.driver.OracleDriver  
user = scott  
password = manager

-----  
Step 2 :

-----  
Create a java program to read the data from the properties file :

```
import java.io.*;  
import java.util.*;  
public class ReadPropertiesFile  
{  
    public static void main(String[] args) throws IOException  
    {  
        FileReader fr = new FileReader("db.properties");  
        Properties p = new Properties();  
        p.load(fr);  
  
        System.out.println(p.getProperty("driver"));  
        System.out.println(p.getProperty("user"));  
        System.out.println(p.getProperty("password"));  
    }  
}
```

-----  
Step 3 :

-----  
We can modify the Properties file data and retived those updated data from java file without re-compilation.

-----  
Program that describes how to read System properties :

```
package com.ravi.properties;  
  
import java.util.Iterator;  
import java.util.Map;  
import java.util.Map.Entry;  
import java.util.Properties;  
import java.util.Set;  
  
public class SystemProperties {  
  
    public static void main(String[] args)  
    {  
  
        Properties p=System.getProperties();  
        Set<Entry<Object, Object>> entrySet = p.entrySet();  
  
        Iterator<Entry<Object, Object>> iterator = entrySet.iterator();  
        iterator.forEachRemaining(System.out::println);  
    }  
}
```

```
    }  
}  
  
It will provide complete System properties.
```

-----  
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 to provide casting as shown below.

```
import java.util.*;  
public 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 String only because we can add anything in the Collection as a result java.lang.ClassCastException as shown in the program below.

```
import java.util.*;  
public class Test1  
{  
    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 outside.

Example:-

```
ArrayList<String > al = new ArrayList<>();
```

Now here we have a guarantee that only String can be inserted as well as only String will come out from the Collection so we can perform String related operation.

Advantages of Generics :

- 
- 1) Type Safe Object (No Compilation warning)
  - 2) No need of type casting
  - 3) Strict compile time checking. (\*Type Erasure)
- 

```

import java.util.*;
public class Test3
{
public static void main(String[] args)
{
    ArrayList<String> al = new ArrayList<>(); //Generic type
    al.add("Ravi");
    al.add("Ajay");
    al.add("Vijay");

    for(int i=0; i<al.size(); i++)
    {
        String name = al.get(i); //no type casting is required
        System.out.println(name.toUpperCase());
    }
}

```

---

//Program that describes the return type of any method can be type safe  
//[We can apply generics on method return type]

```

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.
```

---

16-09-2024

---

Mixing Generic With Non Generic :

We can assign type safe object into raw type, here if the raw type is not adding any new element then we will not get any compilation warning.

```

ArrayList<Lion> al = new ArrayList<>();
al.add(new Lion());
al.add(new Lion());
al.add(new Lion());

ArrayList lions = al; //Assigning safe type object to raw type
import java.util.*;

class Car
{
}
public class Test5
{
    public static void main(String [] args)
    {
        ArrayList<Car> a = new ArrayList<>();
        a.add(new Car());
        a.add(new Car());
        a.add(new Car());

        ArrayList b = a; //assigning Generic to raw type

        System.out.println(b);
    }
}

//Mixing generic to non-generic

import java.util.*;
public class Test6
{
    public static void main(String[] args)

```

```

    {
        List<Integer> myList = new ArrayList<>();
        myList.add(4);
        myList.add(6);
        myList.add(5);

        UnknownClass u = new UnknownClass();
        int total = u.addValues(myList);
        System.out.println("The sum of Integer Object is :" + total);
    }
}

class UnknownClass
{
    public int addValues(List list) //generic to raw type OR
    {
        Iterator it = list.iterator();
        int total = 0;
        while (it.hasNext())
        {
            int i = ((Integer)it.next());
            total += i; //total = 15
        }
        return total;
    }
}

```

Note :-

In the above program the compiler will not generate any warning message because even though we are assigning type safe Integer Object to unsafe or raw type List Object but this List Object is not inserting anything new in the collection so there is no risk to the caller.

-----  
//Mixing generic to non-generic

```

import java.util.*;
public class Test7
{
    public static void main(String[] args)
    {
        List<Integer> myList = new ArrayList<>();
        myList.add(4);
        myList.add(6);
        UnknownClass u = new UnknownClass();
        int total = u.addValues(myList);
        System.out.println(total);
    }
}

class UnknownClass
{
    public int addValues(List list)
    {
        list.add(5); //adding object to raw type
        Iterator it = list.iterator();
        int total = 0;
        while (it.hasNext())
        {
            int i = ((Integer)it.next());
            total += i;
        }
        return total;
    }
}

```

Here Compiler will generate warning message because the unsafe object is

```
inserting the value 5 to safe object.
```

```
-----  
*Type Erasure
```

```
-----  
In the above program the compiler will generate warning message because the  
unsafe List Object is inserting the Integer object 5 so the type safe Integer  
object is getting value 5 from unsafe type so there is a problem to the caller  
method.
```

By writing `ArrayList<Integer>` actually JVM does not have any idea that our `ArrayList` was suppose to hold only Integers.

All the type safe generics information does not exist at runtime. All our generic code is Strictly for compiler.

There is a process done by java compiler called "Type erasure" in which the java compiler converts generic version to non-generic type.

```
List<Integer> myList = new ArrayList<Integer>();
```

At the compilation time it is fine but at runtime for JVM the code becomes

```
List myList = new ArrayList();
```

Note :- GENERIC IS STRICTLY COMPILE TIME PROTECTION.

```
-----  
-----  
//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());
    birds.add(new Bird());

    Test9 t = new Test9();
    t.checkAnimals(dogs);
    t.checkAnimals(cats);
    t.checkAnimals(birds);
}

}

```

Note :- The above program will generate compilation error.

So from the above program it is clear that polymorphism does not work in the same way for generics as it does with arrays.

Example :

```
Parent [] arr = new Child[5]; //valid
Object [] arr = new String[5]; //valid
```

But in generics the same type is not valid

```
List<Object> list = new ArrayList<Integer>(); //Invalid
List<Parent> mylist = new ArrayList<Child>(); //Invalid
List<Animal> animalList = new ArrayList<Dog>(); //Invalid
-----
import java.util.*;
public class Test10
{
public static void main(String [] args)
{
    /*ArrayList<Object> al = new ArrayList<String>(); [Compile time]
    ArrayList al = new ArrayList(); [Runtime, Type Erasure]
    al.add("Ravi");*/

    Object []obj = new String[3]; //valid with Array
    obj[0] = "Ravi";
    obj[1] = "hyd";
    obj[2] = 90; //java.lang.ArrayStoreException
    System.out.println(Arrays.toString(obj));
}
}
```

Note :- This Program will generate java.lang.ArrayStoreException because we are trying to insert 90 (integer value) into String array.

In Array we have an Exception called ArrayStoreException (Which protect us to assign some illegal value in the array at runtime) 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)

---

17-09-2024

---

WildCard (?)

---

<Dog> : Only We can assign Dog Object.

<Animal> : Only we can assign Animal Object

<?> : Many possibilities

<? super Lion> : It is known as Upper bound, Here we can assign Lion, any super of Lion like Animal, Here in future we can't add any more super class.

<? extends Animal> : It is known as lower bound, Here we can assign Animal and any sub class of Animal like Dog, Cat and so on, Here in future we have so many sub classes of Animal so chances of wrong Collection [Permit the sub classes by using declareing sealed ]

---

```
import java.util.*;
class Parent
{
}
class Child extends Parent
{
}

public class Test11
{
    public static void main(String [] args)
    {
        ArrayList<?> lp = new ArrayList<Child>(); //error
        ArrayList<Parent> lp1 = new ArrayList<Parent>();
        ArrayList<Child> lp2 = new ArrayList<>();
        System.out.println("Success");
    }
}
```

---

```
import java.util.*;
public class Test13
{
    public static void main(String[] args)
    {
        List<? extends Number> list1 = new ArrayList<Double>();
        List<? super String> list2 = new ArrayList<Object>();
        List<? super Gamma> list3 = new ArrayList<Alpha>();
        List list4 = new ArrayList();
        System.out.println("yes");
    }
}

class Alpha
```

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

-----  
How to create our own functional interfaces with multiple parameters :

```
package com.ravi.generic;  
  
@FunctionalInterface  
interface TriFunction<T,U,A,R>  
{  
    R apply(T x, U y, A z);  
}  
public class CustomFunctionalInterface  
{  
    public static void main(String[] args)  
    {  
        TriFunction<String, Integer, Integer, String> fn1 = (x, y, z) -> x + y +  
z;  
        System.out.println(fn1.apply("NIT", 100, 200));  
    }  
}  
-----  
class Test<T,U>  
{  
    private T r;  
    public void set(U a)  
    {  
        r = (T) a;  
    }  
    public T get()  
    {  
        return this.r;  
    }  
}  
public class Test15  
{  
    public static void main(String[] args)  
    {  
        Test<String, String> test = new Test();  
        test.set("NIT");  
        System.out.println(test.get());  
    }  
}
```

Note : Here T and U both are different types.

```
-----  
class MyClass<T>  
{  
    T obj;  
    public MyClass(T obj)          //Student obj  
    {  
        this.obj=obj;  
    }  
}
```

```

        T getObj()
    {
        return this.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> stdType = new MyClass<Student>(new
Student(111,"Scott"));
        System.out.println("Student object stored :"+stdType.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> //Element is Fruit Type
{
    private E element;
    public void setElement(E element) //Fruit element
    {
        this.element = element;
    }

    public E 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";
    }
}

//E stands for Element type
class Fruit
{
}

class Apple extends Fruit //Fruit is super, Apple is sub class
{
    @Override
    public String toString()
    {
        return "My Apple";
    }
}

class Basket<E> //E is Fruit type
{
    private E element;
    public void setElement(E element) //Fruit f = new Apple();
    {
        this.element = element;
    }

    public E getElement() //Method return type is Fruit
    {
        return this.element;
    }
}

public class Test1
{
    public static void main(String[] args)
    {
        Basket<Fruit> b = new Basket<>();
        b.setElement(new Apple());
        Apple apple = (Apple) b.getElement();
        System.out.println(apple);
    }
}

```

```

        b.setElement(new Mango());
        Mango mango = (Mango) b.getElement();
        System.out.println(mango);
    }
}

class Mango extends Fruit
{
    @Override
    public String toString()
    {
        return "My Mango";
    }
}

```

Note : There is no difference between element and type parameter, only they are represented by different types

E : Element Type

T : Type Parameter

-----  
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

`java.util.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);
            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.Iterator;  
import java.util.Vector;  
  
class Concurrent extends Thread  
{  
    private Vector<String> listOfFruits;  
  
    public Concurrent(Vector<String> listOfFruits)  
    {  
        super();  
        this.listOfFruits = listOfFruits;  
    }  
  
    @Override  
    public void run()  
    {  
        try  
        {  
            Thread.sleep(2000);  
        }  
        catch(InterruptedException e)  
        {  
        }  
        listOfFruits.add("POMOGRANATE");  
    }  
}  
  
}  
  
public class ConcurrentModificationDemo {  
    public static void main(String[] args) throws InterruptedException  
    {  
        Vector<String> fruits = new Vector<>();  
        fruits.add("Apple");  
        fruits.add("Orange");  
        fruits.add("Grapes");  
        fruits.add("Mango");  
        fruits.add("Guava");  
  
        Concurrent crm = new Concurrent(fruits);  
        crm.start();  
    }  
}
```

```
        Iterator<String> itr = fruits.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.

-----  
Working with Concurrent collection classes :

-----  
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 so it is immutable object , because whenever we make any changes the JVM creates a cloned copy of the array and add/update element to new object.

It is a thread-safe version of `ArrayList` as well as here `Iterator` is fail safe iterator.

\*`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 immutable so by default 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(500);
        }

        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 by eliminating duplicates 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.CopyOnWriteArraySet;
import java.util.concurrent.CopyOnWriteArraySet;

public class CopyOnWriteArraySetExample1
{
    public static void main(String[] args)
    {
        CopyOnWriteArraySet<String> set = new CopyOnWriteArraySet<>();

        set.add("Java");
        set.add("Python");
        set.add("C++");
        set.add("Java");

        Iterator itr = set.iterator();

        // Adding a new element
        set.add("JavaScript");

        for (String language : set)
        {
            System.out.println(language);
        }

        System.out.println(".....");
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
    }
}

-----  

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 : [Bucket Level Locking]
-----  

public class ConcurrentHashMap<K,V> extends AbstractMap<K,V> implements
java.util.concurrent.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/write 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);  
-----
```

19-09-2024

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.

```
//Converting HashMap to 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");
        }
    }
}
```

Stream API in java :

Stream API 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` package.

Package Information :

-----  
java.util -> Base package  
java.util.function -> Functional interfaces  
java.util.concurrent -> Multithreaded support  
java.util.stream -> Processing of Collection Object

Interfaces which contains 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 Stream.of(T ...values) 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.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");  
        //Collection to stream  
        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> streamOfData = items.stream();  
        streamOfData.forEach(System.out::println);  
    }  
}
```

```
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 numbers
        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 in Stream API :

-----  
We can perform two types of Operation in Stream API :

- 1) Intermediate Operation
- 2) Terminal Operation

-----  
Intermediate Operation :

-----  
Intermediate Operation will always produce another Stream, Here Streams are not in a closed position that means further we can apply any intermediate operation method.

In intermediate operation the method return type will always Stream because it is producing another Stream.

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/Map).

Collectors is a predefined final singleton class available in `java.util.stream` sub package which contains static method `toList()`, `toSet()`, `toMap()` to convert the data as a List/Set/Map i.e Collection object. The return type of this method is List/Set/Map interface.

```
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)
    {
        //Fetch all the even numbers from List without Stream
        List<Integer> listOfNumber = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

        ArrayList<Integer> evenNumbers = new ArrayList<>();

        for(Integer num : listOfNumber)
        {
            if(num%2==0)
            {
                evenNumbers.add(num);
            }
        }
        evenNumbers.forEach(System.out::println);

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

        listOfNumber.stream().filter(num ->
num%2==0).forEach(System.out::println);
    }
}
```

```
        List<Integer> asList = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,3,5,7);

        Set<Integer> collect = asList.stream().filter(num ->
num%2==1).collect(Collectors.toSet());

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

```
}
```

```
}
```

```
-----  
20-09-2024  
-----
```

```
//Filtering the name which starts with 'R' character with Stream API
package com.ravi.basic;
```

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo4
{
    public static void main(String[] args)
    {
        List<String> listOfName =
Arrays.asList("Raj","Rahul","Ankit","Roshan","Scott");

        List<String> namesStartWithR = listOfName.stream().filter(str
->str.startsWith("R")).collect(Collectors.toList());

        namesStartWithR.forEach(System.out::println);
    }
}
```

```
-----  
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 because It is an intermediate
operation.
It has an overloaded method which accept Comparator<T> as a parameter through
which we can provide user-defined sorting logic
```

```
//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().collect(Collectors.toList());

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

```

}

-----
package com.ravi.basic;

import java.util.ArrayList;
import java.util.List;

//Fetch all the Employees name whose salary is greater than 50k

record Employee(Integer employeeId, String employeeName, Double employeeSalary)
{
}

public class StreamDemo6
{
    public static void main(String[] args)
    {
        List<Employee> listOfEmployee = new ArrayList<>();
        listOfEmployee.add(new Employee(111, "Scott", 75000D));
        listOfEmployee.add(new Employee(222, "John", 45000D));
        listOfEmployee.add(new Employee(333, "Martin", 55000D));
        listOfEmployee.add(new Employee(444, "Smith", 65000D));
        listOfEmployee.add(new Employee(555, "Virat", 95000D));

        //Will print employee records
        listOfEmployee.stream().filter(emp -> emp.employeeSalary() > 50000).forEach(System.out::println);

        //Will print only employee names
        listOfEmployee.stream().filter(emp -> emp.employeeSalary() > 50000).forEach(e -> System.out.println(e.employeeName()));
    }
}
-----
```

**public Stream map(Function<? super T,? extends 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.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> listOfNumbers =
        Arrays.asList(11,12,13,14,15,16,17,18,19,20);
    }
}
```

```

        listOfNumbers.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> evenCubes = numbers.stream().filter(num ->
num%2==0).map(n -> n*n*n).collect(Collectors.toList());
        System.out.println(evenCubes);

        System.out.println(".....");
        //Find the length of the name
        Stream.of("Raj", "Scott", "subramanyam", "Rahul").map(str->
str.length()).forEach(System.out::println);

        //retrieve first character of all the given name
        List<String> list = List.of("James", "Aryan", "Virat", "Aakash");

        List<Character> collect = list.stream().map(str ->
str.charAt(0)).collect(Collectors.toList());
        System.out.println(collect);

    }
}

-----//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[])
    {
        //Get the name of the Player from Player Object
        List<Player> playerList = createPlayerList();

        Set<String> playerName = playerList.stream().map(player ->
player.playerName()).collect(Collectors.toSet());
        System.out.println(playerName);

    }

    public static List<Player> createPlayerList()
    {
        List<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, "Hardik"));

        return al;
    }
}

```

```

    }
}

record Player(Integer playerId, String playerName)
{
}

-----
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>> into Collection<T> [to make flat i.e converting Collections of collection into single collection or merging of all the collection]

```

package com.ravi.basic;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Test
{
    public static void main(String[] args)
    {
        List<Integer> list1 = Arrays.asList(1,2,3,4,5);
        List<Integer> list2 = Arrays.asList(11,12,13,14,15);
        List<Integer> list3 = Arrays.asList(21,22,23,24,25);

        List<List<Integer>> nestedStr = Arrays.asList(list1, list2, list3);
        System.out.println(nestedStr);

        List<Integer> collect = nestedStr.stream().flatMap(list ->
list.stream()).collect(Collectors.toList());
        System.out.println(collect);
    }
}

//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.Collector;
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>> nestedColl = Arrays.asList(list1, list2, list3);
    }
}

```

```

        System.out.println(nestedColl);

        List<String> flatColl = nestedColl.stream().flatMap(list ->
list.stream()).collect(Collectors.toList());
        System.out.println(flatColl);

    }
}

-----//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>> nestedColl =
List.of(primeNumbers,evenNumbers,oddNumbers);
        System.out.println(nestedColl);

        List<Integer> flatteningNumbers = nestedColl.stream().flatMap(list ->
list.stream()).collect(Collectors.toList());

        System.out.println(flatteningNumbers);

    }
}

-----//Fetching first character using flatMap()
package com.ravi.basic.flat_map;

import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class FlatMapDemo2
{
    public static void main(String[] args)
    {
        List<String> listOfNames =
List.of("Aman","Venkatesh","Raj","Scott","Smith");

        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.function.UnaryOperator;
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(product ->
            product.getListOfProducts().stream()).collect(Collectors.toList());

        System.out.println(collect);
    }
}

```

-----  
Working with Primitive Streams :

Streams works with collections of objects and not primitive types.

Now, to provide a way to work with the three most used primitive types - int, long and double, Java provides three primitive specialized implementations of Stream.

IntStream (represents sequence of primitive int elements)  
 LongStream (represents sequence of primitive long elements)  
 DoubleStream (represents sequence of primitive double elements)

Method of IntStream, LongStream and DoubleStream :

-----  
IntStream, LongStream and DoubleStream are the predefined interfaces available

in java.util.stream sub package.

These interfaces contain static method of(T ...values) through which we can create corresponding type of element.

Arrays which is a predefined class in java.util package provides a predefined method called stream() which will also convert corresponding array object into Stream type

```
public static IntStream stream(int [] array);
public static LongStream(long [] array);
public static DoubleStream(double [] array);
```

Note : By using above methods we can convert the array into corresponding Stream Type.

Program :

```
-----
package com.ravi.basic.flat_map;

import java.util.Arrays;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;

public class PrimitiveToStreamDemo1
{
    public static void main(String[] args)
    {
        IntStream intStream = IntStream.of(1,2,3,4,5,6,7,8);
        LongStream longStream = LongStream.of(1L,2L,3L,4L,5L);
        DoubleStream doubleStream = DoubleStream.of(1.1,1.2,1.3,1.4,1.5);
        intStream.forEach(System.out::print );
        System.out.println();
        longStream.forEach(System.out::print);
        System.out.println();
        doubleStream.forEach(System.out::print);

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

        int a[] = {1,2,3,4,5};
        IntStream intStream2 = Arrays.stream(a);

        long l[] = {1L, 2L, 3L, 4L};
        LongStream longStream2 = Arrays.stream(l);

        double d[] = {1.2, 2.6, 3.9, 8.9};
        DoubleStream doubleStream2 = Arrays.stream(d);

        intStream2.forEach(System.out::print );
        System.out.println();
        longStream2.forEach(System.out::print);
        System.out.println();
        doubleStream2.forEach(System.out::print);
    }
}

-----
```

IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper)

It is a predefined method of Stream interface which comes under flattening.

It allows us to transform each element of the stream into an IntStream (a stream of primitive int values) and then flattens these resulting streams into a single IntStream.

Note : IntStream is a specialized stream for working with int values available in java.util.stream sub package.

```
package com.ravi.basic.flat_map;

import java.util.Arrays;
import java.util.List;
import java.util.function.Supplier;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class FlatMapToIntDemo1 {

    public static void main(String[] args)
    {
        int []a1 = new int[] {1,2,3};
        int []a2 = new int[] {4,5,6};
        int []a3 = new int[] {7,8,9};

        List<int[]> listOfArrays = Arrays.asList(a1,a2,a3);

        IntStream flatMapToInt = listOfArrays.stream().flatMapToInt(array ->
IntStream.of(array));

        flatMapToInt.forEach(System.out::println);
    }
}
-----
```

LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper) :

It is a predefined method of Stream interface which comes under flattening.

It allows us to transform each element of the stream into a LongStream (a stream of primitive long values) and then flattens these resulting streams into a single LongStream.

Note : LongStream is a specialized stream for working with long values available in java.util.stream sub package.

```
package com.ravi.basic.flat_map;

import java.util.Arrays;
import java.util.List;
import java.util.stream.LongStream;

public class FlatMapToLongDemo1 {

    public static void main(String[] args)
    {
        long []arr1 = new long[] {23,33,43};
        long []arr2 = new long[] {53,63,73};
        long []arr3 = new long[] {83,93,103};

        List<long[]> longArray = Arrays.asList(arr1, arr2, arr3);
        LongStream flatMapToLong = longArray.stream().flatMapToLong(array ->
Arrays.stream(array));

        flatMapToLong.forEach(System.out::println);
    }
}
```

```
    }  
}  
-----  
DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper)
```

It is a predefined method of Stream interface which comes under flattening.

It allows us to transform each element of the stream into an DoubleStream (a stream of primitive double values) and then flattens these resulting streams into a single DoubleStream.

Note : DoubleStream is a specialized stream for working with double values available in java.util.stream sub package.

```
-----  
package com.ravi.basic.flat_map;
```

```
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.DoubleStream;  
  
public class FlatMapToDoubleDemo1  
{  
    public static void main(String[] args)  
    {  
        double d1[] = new double[]{1.1, 1.2, 1.3};  
        double d2[] = new double[]{2.1, 2.2, 2.3};  
        double d3[] = new double[]{3.1, 3.2, 3.3};  
  
        List<double[]> listOfDoubleArrays = Arrays.asList(d1,d2,d3);  
  
        DoubleStream doubleStream = listOfDoubleArrays.stream()  
            .flatMapToDouble(array -> DoubleStream.of(array));  
  
        // Print each double value in the flattened stream  
        doubleStream.forEach(System.out::println);  
    }  
}
```

```
-----  
**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.

It has an overloaded method which accept Comparator<T> as a parameter through which we can provide user-defined sorting logic

```
-----  
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.List;
import java.util.stream.Stream;

public class StreamDemo10
{
    public static void main(String[] args)
    {
        //Print the numbers in ascending order
        List<Integer> listOfNum = List.of(89,67,56,45,23,15);

listOfNum.stream().sorted(Integer::compareTo).forEach(System.out::println);
        System.out.println("=====");

        //Print the numbers in descending order
        List<Integer> listOfNumber = List.of(89,67,56,45,23,15);
        listOfNumber.stream().sorted((i1,i2)->
i2.compareTo(i1)).forEach(System.out::println);
        System.out.println("=====");

        //Print the names in Ascending order
        Stream<String> strOfName =
Stream.of("Ankit","Scott","Smith","James");
        strOfName.sorted(String::compareTo).forEach(System.out::println);

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

        //Print the names in Descending order
        Stream<String> strmOfName =
Stream.of("Ankit","Scott","Smith","James");
        strmOfName.sorted((s1,s2)->
s2.compareTo(s1)).forEach(System.out::println);

        System.out.println(".....");
        Stream<String> s = Stream.of("Virat", "Rohit", "Dhoni", "Virat",
"Rohit", "Aswin", "Bumrah");
        s.distinct().sorted((s1,s2)->
s2.compareTo(s1)).forEach(System.out::println);
    }
}

```

-----  
24-09-2024  
-----

```

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,
3, 4, 5);

        numbers.distinct().forEach(System.out::println);
    }
}

```

-----  
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.

```
-----  
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(2).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", "pomogranate");  
  
        List<Integer> fruitsLength = listOffruits  
            .peek(str -> System.out.println("Peeking from Original: " +  
str.toUpperCase()))  
            .map(fruit -> fruit.length())  
            .collect(Collectors.toList());  
        System.out.println("-----");  
        System.out.println(fruitsLength);  
    }  
}
```

```
}
```

Note :- peek(Consumer<T> cons) will not modify the Original Source.

```
-----  
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(12,2,10,3,4,5,6,7,8,9);  
  
        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 < 7).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);
    }
}
```

-----  
Optional<T> class in Java :

-----  
It is a predefined final and immutable class available in java.util package from java 1.8v.

It is a container object which is used to represent an object (Optional object) that may or may not contain a non-null value.

If the value is available in the container, isPresent() method will return true and get() method will return the actual value.

It is very useful in industry to avoid NullPointerException.

Methods of Optional<T> class :

1) public static Optional<T> ofNullable(T x) :

-----  
It will return the object of Optional class with specified value. If the specified value is null then this method will return an empty object of the optional class.

2) public boolean isPresent() :

-----  
It will return true, if the value is available in the container otherwise it will return false.

3) public T get() :

-----  
It will get/fetch the value from the container, if the value is not available then it will throw java.util.NoSuchElementException.

4) public T orElse(T defaultValue) :

-----  
It will return the value, if available otherwise it will return the specified default value.

5) public static Optional<T> of (T value) :

-----  
It will return the optional object with the specified value that is non- null value.

6) public static Optional<T> empty() :

-----  
It will return an empty Optional Object.

```

7) public java.util.stream.Stream stream()
-----
It will Convert optional to Stream.

8) public void ifPresent(Consumer<T> cons) :
-----
It Used to consume/accept the value from optional container if the value is not
null.
-----
package com.ravi.optional_demo;

import java.util.Optional;

public class OptionalDemo1
{
    public static void main(String[] args)
    {
        String str = null;

        Optional<String> optional = Optional.ofNullable(str);

        String orElse = optional.orElse("No value in container");
        System.out.println("Value by orElse :" + orElse);

        //Optional is containing value or not?
        if(optional.isPresent())
        {
            System.out.println("Value by get :" + optional.get());
        }
        else
        {
            System.err.println("No value is available in the container");
        }

    }
}

-----
package com.ravi.optional_demo;

import java.util.Optional;

class Employee
{
    private Integer empId;
    private String empName;

    public Employee() {}

    public Employee(Integer empId, String empName)
    {
        super();
        this.empId = empId;
        this.empName = empName;
    }

    //Changing the style of writing getter method

    public Optional<Integer> getEmpId()
    {
        return Optional.ofNullable(empId);
    }
}

```

```

    }

    public Optional<String> getEmpName()
    {
        return Optional.ofNullable(empName);
    }
}

public class OptionalDemo2
{
    public static void main(String[] args)
    {
        Employee emp = new Employee(111, "Ravi");
        //Employee emp = new Employee();

        Optional<Integer> empId = emp.getEmpId();
        if(empId.isPresent())
        {
            System.out.println(empId.get());
        }
        else
        {
            System.err.println("No id value ");
        }

        Optional<String> empName = emp.getEmpName();
        String name = empName.orElse("name is not available");
        System.out.println(name);
    }
}
-----
package com.ravi.optional_demo;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class OptionalDemo3
{
    public static void main(String[] args)
    {
        List<Optional<String>> optionalList = new ArrayList<>();

        optionalList.add(Optional.of("Hyderabad"));
        optionalList.add(Optional.of("Pune"));
        optionalList.add(Optional.of("Kolkata"));
        optionalList.add(Optional.of("BBSR"));
        optionalList.add(Optional.of("Mumbai"));
        optionalList.add(Optional.empty());

        for(Optional<String> str : optionalList)
        {
            if(str.isPresent())
            {
                System.out.println(str.get());
            }
            else
            {
                System.err.println("Value not available");
            }
        }
    }
}

```

```
 }

}

Note : If we take null then we will get NPE so instead of null we should use
Optional.empty() to represent an empty optional.
```

-----  
25-09-2024  
-----

```
package com.ravi.basic;

import java.util.Optional;

public class OptionalDemo4
{
    public static void main(String[] args)
    {

        Optional<String> optl = Optional.of("India");
        System.out.println(optl.hashCode());

        Optional<String> newOptnl = modifyOptional(optl);
        System.out.println(newOptnl.hashCode());

        // Check if the original Optional is still the same
        System.out.println("Address is :" + (optl == newOptnl));

    }

    public static Optional<String> modifyOptional(Optional<String> optional)
    {

        if (optional.isPresent())
        {
            return Optional.of("Modified: " + optional.get());
        }
        else
        {
            return Optional.empty();
        }
    }
}
```

Note : From the above program It is clear that Optional is an immutable class.

-----  
Method Reference in java :

-----  
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 refere 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)
- 

Example to make a difference between lambda body and Method Reference.

```
package com.ravi.method_reference;

@FunctionalInterface
interface Worker
{
    void work();
}

public class MethodReferenceDemo1 {

    public static void main(String[] args)
    {
        //Lambda Expression
        Worker w = ()-> System.out.println("Worker is working");
        w.work();

        //Method Reference
        Worker w1 = new Employee()::work;
        w1.work();

    }
}

class Employee
{
    public void work()
    {
        System.out.println("Employee is working");
    }
}

-----  
package com.ravi.method_reference;

@FunctionalInterface
interface Worker
{
    void work();
}

public class MethodReferenceDemo1 {

    public static void main(String[] args)
    {
        Worker w = Employee::salary;
        w.work();

    }
}
```

```

class Employee
{
    public static void salary()
    {
        System.out.println("Employee is working for Salary");
    }
}

-----
package com.ravi.method_reference;

@FunctionalInterface
interface Worker
{
    void getSalary(double salary);
}

public class MethodReferenceDemo1 {

    public static void main(String[] args)
    {
        Worker w = new Employee()::getSalary;
        w.getSalary(12000);
    }
}

class Employee
{
    public void getSalary(double sal)
    {
        System.out.println("Employee Salary is :" + sal);
    }
}

-----
Working with static Method Reference :
-----
package com.ravi.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);
    }
}

```

```

//By using Lambda Expression
numbers.forEach(num -> EvenOrOdd.isEven(num));

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

//By Using Method Reference
numbers.forEach(EvenOrOdd::isEven);
}

-----
Working with instance method Reference :

-----
package com.ravi.method_reference;

@FunctionalInterface
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 Reference :

-----
package com.ravi.method_reference;

@FunctionalInterface
interface A
{
    Foo createObject();
}

class Foo
{
    public Foo()
    {
        System.out.println("Test class Constructor invoked");
    }
}

public class ConstructorReferenceDemo1
{

```

```
public static void main(String[] args)
{
    //Lambda Expression
    A a1 = ()->
    {
        return new Foo();
    };
    a1.createObject();

    System.out.println(".....");
    //Method Reference
    A a2 = Foo::new;
    a2.createObject();
}
-----
```

```
package com.ravi.method_reference;

import java.util.function.Function;

class Test
{
    private int x;

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

    public int getX()
    {
        return this.x;
    }
}
```

```
public class ConstructorReferenceDemo
{
    public static void main(String[] args)
    {
        Function<Integer,Test> fn = Test::new;
        Test obj = fn.apply(12);
        System.out.println(obj.getX());
    }
}
```

-----  
26-09-2024  
-----

How to create an Object for Array by using Method Reference :

```
package com.ravi.method_ref;

import java.util.Arrays;
import java.util.function.Function;

class Person
{
    private String name;

    public Person(String name)
    {
        super();
        this.name = name;
    }
```

```

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

public class ConstructorReferenceDemo2
{
    public static void main(String[] args)
    {
        //Creating Person Array

        Function<Integer,Person[]> fn1 = Person[]::new;
        Person[] persons = fn1.apply(3);

        persons[0] = new Person("Raj");
        persons[1] = new Person("Ankit");
        persons[2] = new Person("Rahul");

        System.out.println(Arrays.toString(persons));
    }
}

-----
package com.ravi.method_ref;

import java.util.Scanner;
import java.util.function.Function;

class Student
{
    private int studentId;
    private String studentName;

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

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

public class ConstructorReferenceDemo3
{
    public static void main(String[] args)
    {
        Function<Integer,Student[]> fn2 = Student[]::new;

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of array :");
        int size = sc.nextInt();

        Student[] students = fn2.apply(size);

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

```

```

{
    System.out.print("Enter Student Id :");
    int id = sc.nextInt();
    System.out.print("Enter Student Name :");
    String name = sc.nextLine();
    name = sc.nextLine();

    students[i] = new Student(id, name);

}
System.out.println("Print Student Data");
for(Student student : students)
{
    System.out.println(student);
}
}

}
-----
```

#### 4) Arbitrary Reference : [ClassName::instanceMethodName]

It is a technique through which we can call non static methods with the help of class name.

Here at the time of calling the non static method, Internally JVM passes this keyword.

It will work as a method parameter only because this keyword is available in the method parameter only.

```

package com.ravi.method_ref;

import java.util.Arrays;

public class ArbitraryDemo1
{
    public static void main(String[] args)
    {
        //Comparator using Lambda
        String []names = {"Virat", "Rohit", "Aswin", "Zaheer"};
        Arrays.sort(names, (s1,s2)-> s1.compareTo(s2));
        System.out.println(Arrays.toString(names));

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

        //Comparator using Method Reference
        String []playerNames = {"Virat", "Rohit", "Aswin", "Zaheer"};

        Arrays.sort(playerNames, String::compareTo);
        System.out.println(Arrays.toString(playerNames));
    }
}
```

Note : Comparator<T> is a predefined functional interface available in java.util package.

```

@FunctionalInterface
interface Comparator<T>
{
```

```

        public int compare(T x, T y);
    }

Arrays.sort(T[]obj, Comparator<T>), sort is an overloaded method which is
accepting Object array and Comparator as a parameter so we write
Lambda OR Method reference for Comparator.
-----
package com.ravi.method_ref;

import java.util.Arrays;

class Customer
{
    String name;

    public Customer(String name)
    {
        this.name = name;
    }

    public int CustomerInstanceMethod(Customer cust)
    {
        return this.name.compareTo(cust.name);
    }

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

public class ArbitraryReferenceDemo2
{
    public static void main (String[] args) throws Exception
    {
        Customer customers[] = {new Customer("Zuber"),new Customer("Rohit"), new
Customer("Ankit")};

        Arrays.sort(customers,Customer::CustomerInstanceMethod);

        for(Customer customer : customers)
        {
            System.out.println(customer);
        }
    }
}
-----
```

New Date and Time API :

LocalDate :

It is a predefined final class which represents only Date. The java.util.Date class is providing Date and Time both so, only to get the Date we need to use LocalDate class available in java.time package.

```
    LocalDate d = LocalDate.now();
```

Here now is a static method of LocalDate class and its return type is LocalDate class. (Factory Method)

LocalTime :

-----  
It is also a final class which will provide only time.  
LocalTime d = LocalTime.now();

Here now is a static method of LocalTime class and its return type is LocalTime  
(Factory Method).

LocalDateTime :

-----  
It is also a final class which will provide Date and Time both without a time  
zone. It is a combination of LocalDate and LocalTime class.

LocalDateTime d = LocalDateTime.now();

-----  
package com.ravi.date\_and\_time;

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class Demo1
{
    public static void main(String[] args)
    {
        LocalDate d = LocalDate.now();
        System.out.println(d);

        LocalTime t = LocalTime.now();
        System.out.println(t);

        LocalDateTime dt = LocalDateTime.now();
        System.out.println(dt);
    }
}
```

Now, based on our application requirement we can take only date OR only time or  
Date and time both

-----  
ZonedDateTime : (Date and time + Time zone)

-----  
It is a final class available in java.time package.

It is also provides date and time along with time zone so, by using this class  
we can work with different time zone in a global way.

```
ZonedDateTime x = ZonedDateTime.now();
ZoneId zone = x.getZone();
```

getZone() is a predefined non static method of ZonedDateTime class which returns  
ZoneId class, this ZoneId class provides the different zones, by using  
getAvailableZoneIds() static method we can find out the total zone available  
using this ZoneId class.

-----  
package com.ravi.date\_and\_time;

```
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo2
{
    public static void main(String[] args)
    {
        ZonedDateTime z = ZonedDateTime.now();
        System.out.println(z);
```

```

        ZoneId zone = z.getZone();
        System.out.println(zone);

        System.out.println(ZoneId.getAvailableZoneIds());
    }
}

-----  

Different of() static method :  

-----
List.of();
Set.of();
Map.of();
Stream.of();
Optional.of();
ZoneId.of();

-----  

package com.ravi.date_and_time;

import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo3
{
    public static void main(String[] args)
    {
        ZoneId AusTimeZone = ZoneId.of("Australia/Hobart");
        ZonedDateTime aus = ZonedDateTime.now(AusTimeZone);
        System.out.println("Current Date and Time in Australia Time Zone: " +
aus);

        ZoneId londonTimeZone = ZoneId.of("Europe/London");
        ZonedDateTime lon = ZonedDateTime.now(londonTimeZone);
        System.out.println("Current Date and Time in London Time Zone: " + lon);

        ZoneId america = ZoneId.of("America/Cuiaba");
        ZonedDateTime americaCuiaba = ZonedDateTime.now(america);
        System.out.println("Current Date and Time in America Time Zone: " +
americaCuiaba);

    }
}

```

Note :- of(String zoneId) is a static method of ZoneId abstract class it will provide the zoneId of specified zone as a parameter.  
Now with ZonedDateTime we can find out the specified zoned time.

-----  
DateTimeFormatter :

-----  
It is a predefined final class available in java.time.format sub package.

It is mainly used to formatting and parsing date and time objects according to Java Date and Time API.

Method :

-----  
public static DateTimeFormatter ofPattern(String pattern) :

It is a static method of DateTimeFormatter class, It creates a formatter using user specified String pattern ("dd-MM-yyyy HH:mm:ss") and LocalDateTime class contains format method which takes

`DateTimeFormatter` as a parameter and returns the `String` value.

```
package com.ravi.date_and_time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Demo4
{
    public static void main(String[] args)
    {
        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println(localDateTime);

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-YYYY");

        String formattedDateTime = localDateTime.format(formatter);
        System.out.println("Formatted DateTime: " + formattedDateTime);
    }
}
```

-----  
Terminal Operation :

-----  
Terminal Operation in Stream in java:

-----  
In Java's Stream API, a terminal operation is an operation that produces a result or a side-effect operation.

Unlike intermediate operations, which returns a new stream, terminal operation consumes the elements of the stream.

Once a terminal operation is applied to a stream, the stream is considered consumed and cannot be reused.

It is a final operation.

Methods of Terminal Operation :

```
1) public long count()

2) public Optional<T> min(Comparator<? super T> comparator)

3) public Optional<T> max(Comparator<? super T> comparator)

4) public Optional<T> findAny()

5) public Optional<T> findFirst()

6) public boolean allMatch(Predicate<? super T> predicate)

7) public boolean anyMatch(Predicate<? super T> predicate)

8) public boolean noneMatch(Predicate<? super T> predicate)

9) public void forEach(Consumer<T> cons)

10) public Optional<T> reduce(BinaryOperator<Integer> accumulator)
```

```
11) public R collect(Collector<? super Integer,A,R> collect)
```

-----  
public long count() :

-----  
The count operation returns the number of elements available in the stream. It

is a terminal operation that terminates the stream after its execution. Basically it used to count the number of elements after filter() method.

```
package com.ravi.advanced.count_demo;

import java.util.stream.Stream;

public class CountDemo1
{
    public static void main(String[] args)
    {
        long count =
Stream.of("Ravi","Raj","Elina","Aryan","Sachin").count();
        System.out.println(count);
    }
}

-----
package com.ravi.advanced.count_demo;

//Count the name whose length is greater than 3

import java.util.List;

public class CountDemo2
{
    public static void main(String[] args) {
        List<String> listOfName =
List.of("Raj","Ravi","Virat","Rohit","Ram","Bumrah","Sachin");

        long names = listOfName.stream().filter(name ->
name.length()>3).count();
        System.out.println("Names whose length is > 3 are :" +names);
    }
}

-----
package com.ravi.advanced.count_demo;

//Count Unique elements by using Stream API
import java.util.List;

public class CountDemo3
{
    public static void main(String[] args) {
        List<String> listOfName = List.of("Raj","Raj","Ravi","Virat","Raj");

        long count = listOfName.stream()
            .distinct()
            .count();

        System.out.println("Count of unique elements: " + count);
    }
}

-----
package com.ravi.advanced.count_demo;

//Count the names which are containing the character A
import java.util.Arrays;
import java.util.List;

public class CountDemo4
{
    public static void main(String[] args) {
```

```

        List<String> list = Arrays.asList("Raj", "Ravi", "Rohit", "Virat", "Raj");

        long count = list.stream()
                        .map(String::toUpperCase)
                        .filter(s -> s.contains("A"))
                        .distinct()
                        .count();

        System.out.println("Count of distinct strings containing 'A': " +
count);
    }
}

```

-----  
Working with Predefined functional interfaces :

-----  
UnaryOperator<T> 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 on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

It has a single type parameter, T, which represents both the operand type and the result type.

```

@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,R>
{
    public abstract T apply(T x);
}

```

-----  
package com.ravi.advanced.count\_demo;

```

import java.util.function.UnaryOperator;

public class UnaryOperatorDemo1 {

    public static void main(String[] args)
    {

        UnaryOperator<String> fn1 = str -> str.concat(" World");
        String concat = fn1.apply("Hello");
        System.out.println(concat);

        UnaryOperator<Integer> square = x -> x * x;
        System.out.println(square.apply(5));
    }
}

```

-----  
BinaryOperator<T> 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 upon two operands of the same type, producing a result of the same type as the operands.

This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

It has two parameters of same type, T, which represents both the operand types and the result type.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,U,R>
{
    public abstract T apply(T x, T y);
}

import java.util.function.*;
public class Lambda16
{
    public static void main(String[] args)
    {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(3, 5));
    }
}

ToIntFunction<T> 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 takes an argument of type T and returns an int value. This is typically used in streams when you need to perform operations that result in primitive int value.

```
@FunctionalInterface
public interface ToIntFunction<T>
{
    int applyAsInt(T value);
}

import java.util.*;
import java.util.function.*;
import java.util.stream.*;

record Employee(String name, int experience)
{

public class Lambda17
{
    public static void main(String[] args)
    {
        ArrayList<Employee> listOfEmployee = new ArrayList<>();
        listOfEmployee.add(new Employee("Virat",12));
        listOfEmployee.add(new Employee("Rohit",12));
        listOfEmployee.add(new Employee("Bumrah",6));
        listOfEmployee.add(new Employee("Akshar",5));

        ToIntFunction<Employee> experienceFunction = employee ->
employee.experience();

        int totalYearsOfExperience = listOfEmployee.stream()
            .mapToInt(experienceFunction)
            .sum();

        System.out.println("Total years of experience: " +
totalYearsOfExperience);
    }
}
```

```

        }
}

-----  

Predefined Functional interface :  

-----  

1) Predicate<T>      boolean test(T x)  

2) Consumer<T>       void accept(T x)  

3) Function<T, R>    R apply(T x)  

4) Supplier<T>       T get()  

5) BiPredicate<T, U>  boolean test(T x, U u)  

6) BiConsumer<T, U>   void accept(T x, U u)  

7) BiFunction<T, U, R> R apply(T x, U u)  

8) UnaryOperator<T>   T apply(T x)  

9) BinaryOperator<T>  T apply(T x, T y);  

10)ToIntFunction<T>   int applyAsInt(T x)

```

```
-----  

public Optional<T> min(Comparator<? super T> comparator)
```

It is a predefined method of Stream interface ,It is used to find the minimum element of the stream according to the provided Comparator.

This method is useful when we need to find out the smallest element in a stream, based on a specific comparison criteria using Comparator.

```
-----  

package com.ravi.advanced.min_demo;  

  

import java.util.Arrays;  

import java.util.List;  

import java.util.Optional;  

  

public class MinDemo1  

{  

    public static void main(String[] args)  

    {  

        List<Integer> listOfNumbers = Arrays.asList(10, 20, 5, 40, 25, 1);  

  

        Optional<Integer> min = listOfNumbers.stream().min(Integer::compareTo);  

        min.ifPresent(System.out::println);  

    }  

}
```

Comparator.comparingInt(ToIntFunction<T> fn) :

min() method accepts Comparator as a parameter.Comparator interface provides one static method comparingInt(ToIntFunction<T> fn) functional interface as a parameter, which will accept any kind of data but it will always return int value.(int applyAsInt(T x))

```
-----  

package com.ravi.advanced.min_demo;  

  

import java.util.Comparator;  

import java.util.LinkedList;  

import java.util.Optional;  

import java.util.stream.Stream;  

  

//Finding the minimum age of Employee  

  

record Employee(Integer age, String name)  

{  

}
```

```

public class MinDemo2
{
    public static void main(String[] args)
    {

        Employee e1 = new Employee(23, "Scott");
        Employee e2 = new Employee(29, "Smith");
        Employee e3 = new Employee(21, "John");

        Stream<Employee> streamOfEmployee = Stream.of(e1,e2,e3);

        Optional<Employee> minAge =
streamOfEmployee.min(Comparator.comparingInt(Employee::age));

        if(minAge.isPresent())
        {
            System.out.println("Minimum Age of Employee
is :"+minAge.get());
        }
        else
        {
            System.err.println("No record available");
        }

    }

}

-----
package com.ravi.advanced.min_demo;

import java.util.Comparator;
import java.util.List;
import java.util.Optional;

//Finding the Cheapest Product

record Product(Integer productId, String productName, Double productPrice)
{



}

public class MinDemo3 {

    public static void main(String[] args)
    {
        var p1 = new Product(111, "Camera", 45000D);
        var p2 = new Product(222, "Watch", 23000D);
        var p3 = new Product(333, "HeadPhone", 2000D);

        List<Product> listOfProduct = List.of(p1,p2,p3);

        Optional<Product> min =
listOfProduct.stream().min(Comparator.comparingDouble(Product::productPrice));

        if(min.isPresent())
        {
            System.out.println("Cheapest product price is :" +min.get());
        }
        else
        {
            System.err.println("No record is available");
        }

    }

}

```

```
        }
    }
}
```

Note : Comparator interface has provided a static method comparingDouble(ToDoubleFunction<T> x) which will accept ToDoubleFunction as a parameter (which is basically any type of Input <T>) but return value must be double

```
-----  
public Optional<T> max(Comparator<? super T> comparator)  
-----
```

It is a predefined method of Stream interface ,It is used to find the maximum element of the stream according to the provided Comparator.

This method is useful when we need to find out the largest element in a stream, based on a specific comparison criteria using Comparator.

```
package com.ravi.advanced.max_demo;  
  
import java.util.Comparator;  
import java.util.List;  
import java.util.Optional;  
  
public class MaxDemo1 {  
  
    public static void main(String[] args)  
    {  
        List<String> listOfFruits =  
List.of("Apple", "Orange", "Mango", "Grapes", "Pomogranate");  
  
        Optional<String> max =  
listOfFruits.stream().max(Comparator.comparingInt(String::length));  
  
        max.ifPresent(System.out::println);  
    }  
}  
-----  
package com.ravi.advanced.max_demo;  
  
import java.util.Comparator;  
import java.util.Optional;  
import java.util.stream.Stream;  
  
//Finding the Employee with the Highest Salary  
  
record Employee(Integer employeeId, String employeeName, Double employeeSalary)  
{  
}  
  
public class MaxDemo2  
{  
    public static void main(String[] args)  
    {  
        Employee e1 = new Employee(111, "Aman", 23000D);  
        Employee e2 = new Employee(222, "Ramesh", 24000D);  
        Employee e3 = new Employee(333, "Suraj", 25000D);  
        Employee e4 = new Employee(444, "Raj", 26000D);  
        Employee e5 = new Employee(555, "Scott", 46000D);  
  
        Stream<Employee> streamOfEmployees = Stream.of(e1, e2, e3, e4, e5);  
    }  
}
```

```

        Optional<Employee> max =
streamOfEmployees.max(Comparator.comparingDouble(Employee::employeesSalary));

        if(max.isPresent())
{
    System.out.println("Employee Having Maximum Salary
is :"+max.get());
}
else
{
    System.out.println("No record Available");
}
}

-----
public Optional<T> findAny() :
-----
It is a predefined method of Stream interface ,It is used to return an Optional
which describes some element of the stream, or an empty Optional if the stream
is empty.

```

It's useful when we need any element from the stream but don't care which one (Randomly pick an element).

```

package com.ravi.advanced.find_any;

import java.util.List;
import java.util.Optional;

public class FindAnyDemo1
{
    public static void main(String[] args)
    {
        List<String> listOfNames = List.of("Raj", "Rahul", "Ankit");

        Optional<String> findAny = listOfNames.parallelStream().findAny();

        findAny.ifPresent(System.out::println);
    }
}

-----
package com.ravi.advanced.find_any;

import java.util.List;
import java.util.Optional;

public class FindAnyDemo2 {

    public static void main(String[] args)
    {
        List<String> listofName = List.of("Sachin", "Ankit", "Aman", "Rahul",
"Ravi");

        Optional<String> anyElement = listofName.parallelStream().filter(s ->
s.startsWith("R")).findAny();

        anyElement.ifPresent(System.out::println);
    }
}

```

Note : findAny() will select any random element but if we use parallelStream() then we will get the effect.

```
-----  
public Optional<T> findFirst()  
-----
```

It is a predefined method of Stream interface ,It is used to return an Optional which describes the first element of the stream, or an empty Optional if the stream is empty.

```
package com.ravi.advanced.find_first;  
  
import java.util.stream.Stream;  
  
public class FindFirstDemo1  
{  
    public static void main(String[] args)  
    {  
        Stream<String> playerName = Stream.of("Virat", "Rohit", "Raj",  
"Bumrah", "Arshdeep");  
  
        playerName.findFirst().ifPresent(System.out::println);  
    }  
}
```

Differences between findAny() and findFirst()

findFirst(): Always returns the first element of the stream, which is particularly useful for ordered streams.

findAny(): Returns any element from the stream, and is typically used in unordered streams where the order is not required. It may return elements faster because it does not have to maintain the order.

Note : Collection interface has provided parallelStream() method for fast execution [It uses multiple threads]

```
-----  
public boolean allMatch(Predicate<? super T> predicate)  
-----
```

It is a predefined method of Stream interface , It is used to check if all elements of the stream match a given predicate. This method is useful when we need to verify that every element in a stream satisfies a specific condition by using Predicate.

```
package com.ravi.advanced.match;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Predicate;  
import java.util.stream.Stream;  
  
public class AllMatchDemo1 {  
  
    public static void main(String[] args)  
    {  
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);  
  
        boolean allPositive = stream.allMatch(n -> n > 0);  
        System.out.println("All elements are positive: " + allPositive);  
  
        System.out.println(".....");  
  
        List<Integer> numbers = Arrays.asList(2, 4, 6, 8, 10, 11);
```

```

        Predicate<Integer> isEven = number -> number % 2 == 0;
        boolean allEven = numbers.stream().allMatch(isEven);

        if (allEven)
        {
            System.out.println("All numbers are even.");
        }
        else
        {
            System.out.println("Not all numbers are even.");
        }

    }

}
-----
```

```
public boolean anyMatch(Predicate<? super T> predicate)
```

-----  
It is a predefined method of Stream interface, It returns a boolean indicating whether any elements of the stream match the given predicate. It is useful when we need to verify if there exists at least one element in the stream that satisfies the provided condition.

```

package com.ravi.advanced.match;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class AnyMatchDemo1
{
    public static void main(String[] args)
    {
        List<String> listOfName =
List.of("Virat", "Rohit", "Bumrah", "Surya");

        boolean startsWithA = listOfName.stream().anyMatch(name ->
name.startsWith("A"));

        System.out.println("Any name starts with letter 'A' : " +
startsWithA);

        System.out.println("=====");
        List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 8);

        Predicate<Integer> isEven = number -> number % 2 == 0;

        boolean anyEven = numbers.stream().anyMatch(isEven);

        if (anyEven)
        {
            System.out.println("There is at least one even number.");
        }
        else
        {
            System.out.println("There are no even numbers.");
        }
    }

}
-----
```

```
public boolean noneMatch(Predicate<? super T> predicate)
```

-----

It is a predefined method of Stream interface, It is used to check if no elements of the stream match a given predicate. It returns a boolean value true if no elements match the predicate, and false if at least one element matches the predicate or if the stream is empty.

```
package com.ravi.advanced.match;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class NoneMatchDemo1
{
    public static void main(String[] args)
    {
        List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 9, 2, 8);

        Predicate<Integer> isEven = number -> number % 2 == 0;

        boolean noneEven = numbers.stream().noneMatch(isEven);

        if (noneEven)
        {
            System.out.println("There are no even numbers.");
        }
        else
        {
            System.out.println("There is at least one even number.");
        }
    }
}
-----
```

```
R collect(Collector<? super T, A, R> collector);
-----
```

It is a predefined method of Stream interface. It is used to transform the elements of a stream into a different form like collection i.e. List, Set, or Map.

The collect() method takes an argument of type Collector, which is a functional interface that specifies how to perform the collection operation. The Collectors class has provided following static methods

- a) toList(): Collects the elements of the stream into a List.
  - b) toSet(): Collects the elements of the stream into a Set.
  - c) toMap(): Collects the elements of the stream into a Map.
  - d) joining(CharSequence ch): Concatenates the elements of the stream into a String.
  - e) groupingBy(Function<T,R> mapper): Groups the elements of the stream by a classifier function.
  - f) partitioningBy(Predicate<T> p): Partitions the elements of the stream into two groups based on a predicate.
- ```
-----
```

```
package com.ravi.project;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.stream.Collectors;

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

```

{
    Map<String, Double> products = new ConcurrentHashMap<>();
    products.put("Laptop", 50000.00);
    products.put("Smartphone", 40000.00);
    products.put("Tablet", 45000.00);
    products.put("Smartwatch", 5000.00);
    products.put("Headphones", 1000.00);

    //Filter the product whose price is >= 40000

    Map<String, Double> filteredProd =
products.entrySet().stream().filter(entry ->
entry.getValue()>40000).collect(Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));

    System.out.println(filteredProd);

    //Find the total sum of Filtered Product
    double productSum =
filteredProd.values().stream().mapToDouble(Double::doubleValue).sum();

    System.out.println(productSum);
}

```

01-10-2024

`Collectors.joining(CharSequence delimiter) :`  
It is used to join the elements using specified delimiter.

```

package com.ravi.advanced.collect;

//Join the elements by using joining() methods of Collectors class
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class CollectDemo1
{
    public static void main(String[] args)
    {
        List<String> list = Arrays.asList("a", "b", "c", "d");
        String result = list.stream().collect(Collectors.joining("@"));
        System.out.println(result);

    }
}

```

`public static Map<K,List<T>> groupingBy(Function<T,R> mapper)`  
`groupingBy(Function<T,R> mapper)` is a predefined static method  
of Collectors class which is group List of elements based  
on the specified criteria.The return type of this method is `Map<K,List<T>>`.

The criteria by which we will group the element that will become  
the key of the Map and that key will contain list of elements.

```

package com.ravi.advanced.collect;

//Group the city name according to length of the city name
import java.util.*;
import java.util.stream.Collectors;

```

```

public class CollectDemo2
{
    public static void main(String[] args)
    {
        List<String> items = Arrays.asList("Delhi", "Indore", "Kolkata", "Pune",
"Hyderabad", "Mumbai", "Chennai");

        Map<Integer, List<String>> groupedByLength =
items.stream().collect(Collectors.groupingBy(String::length));

        groupedByLength.forEach((length, cities) ->
        {
            System.out.println("Length " + length + ": " + cities);
        });
    }
}

-----  

package com.ravi.advanced.collect;

//Print the length of the country
import java.util.*;
import java.util.stream.Collectors;

public class CollectDemo3
{
    public static void main(String[] args)
    {
        List<String> listOfCountry =
List.of("India", "Australia", "USA", "China", "Japan");

        Map<String, Integer> map = listOfCountry.stream()
            .collect(Collectors.toMap(
                countryName -> countryName,
                countryName -> countryName.length()
            ));

        map.forEach((key, value) ->
        {
            System.out.println(key + ": " + value);
        });
    }
}

-----  

package com.ravi.advanced.collect;

//Based on the department, group all the employee

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

record Department(Integer deptId, String deptName)
{



record Employee(Integer empId, String empName, double salary, Department dept)
{
    //111 , "A", 23890.89, new Department(1,"IT");
}

```

```

public class CollectDemo4
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee(111, "Raj", 23789.89, new Department(1,
"IT"));
        Employee e2 = new Employee(222, "Rahul", 23789.89, new Department(1,
"IT"));
        Employee e3 = new Employee(333, "Scott", 23789.89, new Department(2,
"Sales"));
        Employee e4 = new Employee(444, "Smith", 23789.89, new Department(2,
"Sales"));
        Employee e5 = new Employee(333, "Virat", 23789.89, new Department(3,
"HR"));
        Employee e6 = new Employee(444, "Rohit", 23789.89, new Department(3,
"HR"));

        List<Employee> list = List.of(e1,e2,e3,e4,e5,e6);

        Map<Department, List<Employee>> collect =
list.stream().collect(Collectors.groupingBy(Employee::dept));

        collect.forEach((key, value) ->
        {
            System.out.println(key + ": " + value);
        });
    }
}
-----
```

`public static Map<Boolean, List<T>> partitioningBy(Predicate<T> p )`

It is a predefined static method of Collectors class.

It is used to partition the elements of a stream into two groups based on a given predicate.

The result is a Map with Boolean keys, where the true key corresponds to elements that satisfy the predicate, and the false key corresponds to elements that do not satisfy the predicate.

```

-----
package com.ravi.advanced.collect;

//Partition the given number based on the Predicate

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectDemo5
{
    public static void main(String[] args)
    {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        Map<Boolean, List<Integer>> partitioned = numbers.stream()
            .collect(Collectors.partitioningBy(n -> n % 2 == 0));
    }
}
```

```

        System.out.println(partitioned);
    }
}
-----
package com.ravi.advanced.collect;

//Partition the given number based on the Predicate and convert to Set

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class CollectDemo6
{
    public static void main(String[] args)
    {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 3);

        Map<Boolean, Set<Integer>> partitioned = numbers.stream()
            .collect(Collectors.partitioningBy(
                n -> n % 2 == 1,
                Collectors.toSet()));

        System.out.println(partitioned);

    }
}
-----
package com.ravi.advanced.collect;

//Count how many elements are partition based on given predicate

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectDemo7
{
    public static void main(String[] args)
    {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11);

        Map<Boolean, Long> partitioned = numbers.stream()
            .collect(Collectors.partitioningBy
            (
                n -> n % 2 == 1,
                Collectors.counting()));

        System.out.println(partitioned);

    }
}
Collectors.counting() to count the numbers.
-----
Optional<T> reduce(BinaryOperator<T> accumulator)
-----
T reduce(T identity, BinaryOperator<T> accumulator)
-----
It is a predefined method of Stream interface.

This method is useful for combining stream elements into a single result because
it accept BinaryOperator<T> as a parameter, such as computing the sum, product,
```

or concatenation of elements.

It performs a reduction on the elements of the stream, using an associative accumulation function, and returns an Optional.

```
package com.ravi.advanced.reduce;

import java.util.Optional;
import java.util.stream.Stream;

public class ReduceDemo1
{
    public static void main(String[] args)
    {

        Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
        Optional<Integer> reduce = integerStream.reduce(Integer::sum);

        reduce.ifPresent(System.out::println);

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

        integerStream = Stream.of(1, 2, 3, 4, 5);
        Integer sumWithIdentity = integerStream.reduce(2, Integer::sum);
        System.out.println(sumWithIdentity);

    }
}

-----  
package com.ravi.advanced.reduce;

//Finding the maximum number

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ReduceDemo2
{
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 9);

        Optional<Integer> max = numbers.stream()
            .reduce(Integer::max);

        max.ifPresent(System.out::println);
    }
}

-----  
package com.ravi.advanced.reduce;

//Finding the multiplication of all numbers

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ReduceDemo3
{
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```

        Optional<Integer> product = numbers.stream()
                                         .reduce((a, b) -> a * b);

        product.ifPresent(System.out::println);

        System.out.println("=====");
        Integer reduce = numbers.stream().reduce(1,(a,b)-> a*b);
        System.out.println(reduce);

    }

}

-----  

package com.ravi.advanced.reduce;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ReduceDemo4
{
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Java", "is", "Best");

        Optional<String> concatenated = words.stream()
                                         .reduce((a, b) -> a + " " + b);

        concatenated.ifPresent(System.out::println);
    }
}

-----  

package com.ravi.advanced.reduce;

//Finding the total sale amount

import java.util.stream.Stream;

record Sale(String item, Double amount)
{

}

public class ReduceDemo5
{
    public static void main(String[] args)
    {
        Stream<Sale> sales = Stream.of(
            new Sale("Camera", 10000.0),
            new Sale("Mobile", 50000.0),
            new Sale("Laptop", 80000.0),
            new Sale("LED", 20000.0)
        );

        Double totalSale = sales.reduce(0.0, (sum , sale)-> sum +
sale.amount(),Double::sum);

        System.out.println("Total Sale for today is :" +totalSale);

    }
}

```

```

public class Tester {
    private static final String RED = "\u033[1;31m"; // RED
    private static final String RESET = "\u033[0m"; // Text Reset
    public static void main(String[] args)
    {
        List<Employee> list = EmployeeAdder.addDetails();
        Set<Integer> set = new HashSet<>();
        Set<String> set2 = new HashSet<>();
        // 1. Filter Employees by Gender:
        // - Retrieve a list of all female employees.
        System.out.println(RED+"*****Retrieve a list of all female
employees*****"+RESET);
        list.stream()
            .filter(t -> t.getGender().equals("Female"))
            .forEach(System.out::println);
        System.out.println(RED+"*****Get a list of employees older than 30
years*****"+RESET);
        // 2. Filter Employees by Age:
        // - Get a list of employees older than 30 years.
        list.stream()
            .filter(employee->employee.getAge()>30)
            .forEach(System.out::println);
        System.out.println(RED+"*****Find employees with a salary greater
than $50,000*****"+RESET);
        // 3. Filter Employees by Salary:
        // - Find employees with a salary greater than $50,000.
        list.stream()
            .filter(employee->employee.getSalary()>50000)
            .forEach(System.out::println);

        // 4. Map Employee Names:
        // - Create a list of employee names (Strings).
        System.out.println(RED+"*****Create a list of employee names
(Strings)*****"+RESET);
        list.stream()
            .map(employee->employee.getName())
            .forEach(System.out::println);

        // 5. Calculate Average Salary:
        // - Calculate the average salary of all employees.
        System.out.println(RED+"*****Calculate the average salary of all
employees*****"+RESET);
        doubleorElseThrow = list.stream()
            .mapToDouble(Employee::getSalary)
            .average()
            .orElseThrow();
        System.out.println("Average : "+orElseThrow);

        // 6. Find Maximum Salary:
        // - Find the employee with the highest salary.
        System.out.println(RED+"*****Find the employee with the highest
salary*****"+RESET);
        double max = list.stream()
            .mapToDouble(Employee::getSalary)
            .max()
            .orElseThrow();
        System.out.println("Max Salary : "+max);

        // 7. Group Employees by Gender:
        // - Group employees by gender and return
        // a map with gender as the key and a list of employees as the
value.
        System.out.println(RED+"*****Group employees by gender and return a
map*****"+RESET);
    }
}

```

```

list.stream()
.collect(Collectors.groupingBy(Employee::getGender))
.forEach((key,value)->System.out.println(key+" - "+value));

// 8. Count Male Employees:
//     - Count the number of male employees.
System.out.println(RED+"*****Count the number of male
employees."+RESET);
long count = list.stream()
    .filter(employee -> employee.getGender().equals("Male"))
    .count();
System.out.println("Count : "+count);

// 9. Sum of All Salaries:
//     - Calculate the total sum of salaries for all employees.
System.out.println(RED+"*****Calculate the total sum of salaries
for all employees."+RESET);
double sum = list.stream()
    .mapToDouble(employee->employee.getSalary())
    .sum();
System.out.println("Sum : "+sum);

// 10. Sort Employees by Name:
//      - Sort the employees by their names in alphabetical order.
System.out.println(RED+"*****Sort the employees by their names in
alphabetical order."+RESET);
list.stream().sorted(Comparator.comparing(Employee::getName))
    .forEach(System.out::println);

// 11. Sort Employees by Age:
//      - Sort the employees by age in ascending order.
System.out.println(RED+"*****Sort the employees by age in
ascending order."+RESET);
list.stream()
    .sorted(Comparator.comparing(Employee::getAge))
    .forEach(System.out::println);

// 12. Sort Employees by Salary:
//      - Sort the employees by salary in descending order.
System.out.println(RED+"*****Sort the employees by salary in
descending order."+RESET);
list.stream()
    .sorted(Comparator.comparing(Employee::getAge).reversed())
    .forEach(System.out::println);

// 13. Find Oldest Employee:
//      - Find the oldest employee.
System.out.println(RED+"*****Find the oldest
employee."+RESET);
//      intorElseThrow2 =
list.stream().mapToInt(Employee::getAge).max().orElseThrow();
Employee employee = list.stream()
    .max((e1,e2)->(e1.getAge()-e2.getAge()))
    .get();
System.out.println(employee);
//      System.out.println("Max Age : "+orElseThrow2);

// 14. Group Employees by Age:
//      - Group employees into age groups (e.g., 20-30, 31-40, etc.)
//      and return a map with age group as the key and a list of
employees as the value.
System.out.println(RED+"****Group employees into age groups (e.g.,
20-30, 31-40, etc.)"+RESET);
list.stream()

```

```

.collect(Collectors.groupingBy((t) ->
    {int age = t.getAge();
     if(age>=20 && age<=30)
         return "20-30";
     else if (age>=31 && age<=40)
         return "31-40";
     else
         return "40+";
    })).forEach((key,value)->
System.out.println(key+ " - "+value));
// 15. Find Employees with a Specific Age:
//      - Find all employees who are exactly 35 years old.
System.out.println(RED+"*****" Find all employees who are
exactly 35 years old."*****"+RESET);
list.stream().filter(k->k.getAge()==35).forEach(System.out::println);

// 16. Calculate the Sum of Salaries by Gender:
//      - Calculate the sum of salaries for each gender (i.e., male and
female)
//      and return a map with gender as the key and the sum of salaries
as the value.
System.out.println(RED+"*****" Calculate the sum of salaries for
each gender."*****"+RESET);
list.stream()
.collect(Collectors.groupingBy(Employee::getGender ,
    Collectors.summingDouble(Employee::getSalary)))
.forEach((key,value)->System.out.println(key +" - "+value));

// 17. Find Employees with Names Starting with "J":
//      - Find all employees whose names start with the letter "E."
System.out.println(RED+"*****"Find all employees whose names
start with the letter J."*****"+RESET);
list.stream().filter(k->k.getName().startsWith("E")).forEach(System.out::println);

// 18. Calculate the Average Salary for Male and Female Employees:
//      - Calculate the average salary separately for male
//      and female employees and return a map with gender
//      as the key and the average salary as the value.
System.out.println(RED+"*****"Calculate the average salary
separately for male and female."*****"+RESET);
list.stream()
.collect(Collectors.groupingBy(Employee::getGender,
    Collectors.averagingDouble(Employee::getSalary)))
.forEach((key,value)->System.out.println(key+ " - "+value));

// 19. Find the Top N Highest Paid Employees:
//      - Find the top N employees with the highest salaries.

System.out.println(RED+"*****"Find the top N employees with the
highest salaries."*****"+RESET);
list.stream().sorted((o1, o2) -> -(int)(o1.getSalary()-
o2.getSalary()))
.limit(5).forEach(System.out::println);

list.stream()
.sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
.limit(3)
.forEach(System.out::println);

// 20. Retrieve Distinct Age Values:
//      - Get a list of distinct ages of all employees.

```

```

        System.out.println(RED+"*****Get a list of distinct ages of all
employees.*****"+RESET);

        list.stream().filter(k->!
set.add(k.getAge())).forEach(System.out::println);

//          21. Find the Three Lowest-Paid Employees:
//                  - Find and display the names of the three employees with
the lowest salaries.
        System.out.println(RED+"*****Find and display the names of the
three employees with the lowest salaries.*****"+RESET);
        list.stream()
.sorted((o1,o2)->(int)(o1.getSalary()-o2.getSalary()))
.map(k->k.getName())
.limit(3)
.forEach(System.out::println);

//          22. Sort Employees by Name Length:
//                  - Sort employees by the length of their names (shortest to
longest).
        System.out.println(RED+"*****Sort employees by the length of
their names (shortest to longest).*****"+RESET);
        list.stream()
.sorted((o1, o2) ->(o1.getName().length()-o2.getName().length()))
.forEach(System.out::println);

//          23. Group Employees by Age Range:
//                  - Group employees into custom
//                                age ranges (e.g., 20-29, 30-39, etc.) and
//                                return a map with the age range as the key and a list of
employees as the value.
        System.out.println(RED+"*****Group employees into custom age
ranges (e.g., 20-29, 30-39, etc.).*****"+RESET);
        list.stream()
.collect(Collectors.groupingBy((t)->
{
    int age = (t).getAge();
    if(age>=20 && age<=29)
        return "20-29";
    else if(age>=30 && age <= 39)
        return "30-39";
    else
        return "40+";
}))
.forEach((key,value)->System.out.println(key+" - "+value));

//          24. Find the Average Salary of Employees Aged 30 or Younger:
//                  - Calculate the average salary of employees aged 30 or
younger.

        System.out.println(RED+"*****Calculate the average salary of
employees aged 30 or younger.*****"+RESET);
        doubleorElseThrow2 = list.stream()
.filter(emp->emp.getAge()<=30).mapToDouble(k-
>k.getSalary()).average().orElseThrow();
        System.out.println(orElseThrow2);

//          25. Find the Names of Male Employees with Salaries Over $60,000:
//                  - Retrieve the names of male employees with salaries over
$60,000.

        System.out.println(RED+"*****Retrieve the names of male
employees with salaries over $53,000.*****"+RESET);
        list.stream()
.filter(e->e.getSalary()>=53000 && e.getGender().equals("Male"))

```

```

        .map(k->k.getName())
        .forEach(System.out::println);

//      26. Find the Youngest Female Employee:
//            - Find the youngest female employee.
System.out.println(RED+"*****Find the youngest female
employee."+RESET);
Employee employee2 = list.stream()
    .filter(k->k.getGender().equals("Female"))
    .collect(Collectors.minBy((o1, o2) -> o1.getAge()-o2.getAge())).get();
System.out.println(employee2);

//      27. Retrieve the Names of Employees in Reverse Order:
//            - Get a list of employee names in reverse order (from the
last employee to the first).
System.out.println(RED+"*****Get a list of employee names in
reverse order"+RESET);
List<String> collect = list.stream().map(k-
>k.getName()).collect(Collectors.toList());
Collections.reverse(collect);
System.out.println(collect);

//      28. Find the Highest Salary Among Female Employees:
//            - Find the highest salary among female employees.
System.out.println(RED+"*****Find the highest salary among female
employees."+RESET);
Employee employee3 = list.stream()
    .filter(k->k.getGender().equals("Female"))
    .collect(Collectors.maxBy((o1, o2) -> (int)(o1.getSalary()-o2.getSalary()))).get();
System.out.println(employee3);

//      29. Group Employees by Age and Gender:
//            - Group employees by both age and gender and return a
multi-level map.
System.out.println(RED+"*****Group employees by both age and
gender and return a multi-level map."+RESET);
Map<String, Map<Integer, List<Employee>>> collect2 = list.stream()
    .collect(Collectors.groupingBy(Employee::getGender, Collectors.groupi-
ngBy(Employee::getAge)));

        collect2.forEach((key,value)->
{
    value.forEach((k,v)->System.out.println(k+"-"+v));
    System.out.println(key+"-"+value);
});

//      30. Find the Sum of Salaries for Employees with Names Containing
"Smith":
//            - Calculate the sum of salaries for employees whose names
contain the substring "Smith."
System.out.println(RED+"***** Calculate the sum of salaries for
employees whose names contain the substring Smith***"+RESET);
double sum2 = list.stream().filter(k-
>k.getName().contains("Smith")).mapToDouble(k->k.getSalary()).sum();
System.out.println(sum2);

//      31. Find the Names of Employees Aged 30-40 with Salaries Between
$50,000 and $60,000:
//            - Retrieve the names of employees aged 30-40 with salaries
between $50,000 and $60,000.
System.out.println(RED+"*****Retrieve the names of employees aged
30-40 with salaries between $50,000 and $60,000."+RESET);

```

```

        list.stream()
            .filter(k->(k.getAge()>=30 &&
k.getAge()<=40)&&(k.getSalary()>=52000&&k.getSalary()<=60000))
            .forEach(System.out::println);

//          32. Calculate the Total Number of Employees:
//                  - Determine the total count of employees.

        System.out.println(RED+"***** Determine the total count of
employees."+"*****"+RESET);
        System.out.println("Total Count of employess :
"+list.stream().count());

//          33. Find the Most Common Age Among Employees:
//                  - Determine the age that is most common among the
employees.

        System.out.println(RED+"*****Determine the age that is most common
among the employees."+"**"+RESET);
        Integer orElseThrow3 = list.stream()
            .collect(Collectors.groupingBy(Employee::getAge,Collectors.counting(
))) // grouping ages and count
            .entrySet() // converting to set
            .stream()
            .max(Map.Entry.comparingByValue()) // finding max value in map
            .map(Map.Entry::getKey) // getting key of max value
            .orElseThrow(); // getting the key
        System.out.println(orElseThrow3);

//          34. Find the Median Salary:
//                  - Calculate the median salary of all employees.

        System.out.println(RED+"*****Calculate the median salary of all
employees."+"*****"+RESET);
        List<Employee> list2 = list.stream()
            .sorted(Comparator.comparingDouble(Employee::getSalary)).toList();
        int size = list2.size();
        if(size%2==0)
        {
            double s = list2.get(size/2-1).getSalary();
            double s1 = list2.get(size/2).getSalary();
            System.out.println((s+s1)/2.0);
        }
        else {
            System.out.println(list2.get(size/2).getSalary());
        }

//          35. Group Employees by Age and Count:
//                  - Group employees by age and count the number of employees in
each age group.

        System.out.println(RED+"*****Group employees by age and
count*****"+RESET);

list.stream().collect(Collectors.groupingBy(Employee::getAge,Collectors.counting(
)))
            .forEach((key,value)->System.out.println(key+" - "+value));

//          36. Find the Employee with the Longest Name:
//                  - Find the employee with the longest name.

        System.out.println(RED+"*****Find the employee with the longest
name."+"*****"+RESET);
        Employee employee4 = list.
            stream()

```

```

        .max((o1, o2) -> o1.getName().length())
- o2.getName().length())
        .get();
    System.out.println(employee4);

//      37. Calculate the Sum of Salaries for Each Age:
//              - Calculate the sum of salaries for each distinct age in a
map.
    System.out.println(RED+"*****Calculate the sum of salaries for
each distinct age in a map*****"+RESET);
    set.clear();
    list.stream()
        .filter(k->!set.add(k.getAge()))
        .collect(Collectors.groupingBy(Employee::getAge, Collectors.summingDo
uble(Employee::getSalary)))
        .forEach((key,value)->System.out.println(key+" - "+value));

//      38. Sort Employees by Age, Then by Salary:
//              - Sort employees first by age in ascending order, and then
by salary in descending order.
    System.out.println(RED+"*****Sort employees first by age in
ascending order, and then by salary in descending order*****"+RESET);
    list.stream()
        .sorted(Comparator.comparingInt(Employee::getAge)
            .thenComparing(Comparator.comparingDouble(Employee::getS
alary).reversed()))
        .forEach(System.out::println);

//      39. Find Employees Whose Names Contain More Than One Word:
//              - Retrieve employees whose names consist of more than one
word.
    System.out.println(RED+"*****Retrieve employees whose names
consist of more than one word*****"+RESET);
    list.stream()
        .filter(k->k.getName().length()>1)
        .forEach(System.out::println);

//      40. Find the Two Highest Paid Female Employees:
//              - Find and display the names of the two highest-paid female
employees.
    System.out.println(RED+"*****Find and display the names of the
two highest-paid female employees*****"+RESET);
    list.stream()
        .filter(k->k.getGender().equals("Female"))
        .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
        .limit(2)
        .forEach(System.out::println);

//      41. Find the Employee with the Highest Salary in Each Gender:
//              - Find the employee with the highest salary for each gender
(male and female).
    System.out.println(RED+"*****Find the employee with the highest
salary for each gender (male and female)*****"+RESET);
    list.stream()
        .collect(Collectors.toMap(Employee::getGender, k->k, (e1, e2) ->
e1.getSalary()>=e2.getSalary()?e1:e2))
        .forEach((key,value)->System.out.println(key+" - "+value));

//      42. Retrieve Employees with Unique Names:
//              - Find employees with unique names (no duplicate names in
the list).
    System.out.println(RED+"*****Find employees with unique names (no
duplicate names in the list)*****"+RESET);
    list.stream().filter(k->set2.add(k.getName())));

```

```

        .forEach(System.out::println);
        set2.clear();

//        43. Find the Names of Employees in Uppercase:
//              - Get a list of employee names in uppercase.
        System.out.println(RED+"*****Get a list of employee names in
uppercase.*****"+RESET);
        list.stream()
        .filter(k->k.getName().equals(k.getName().toUpperCase()))
        .forEach(System.out::println);

//        44. Calculate the Salary Range (Min-Max) for Each Age Group:
//              - Calculate the salary range (minimum and maximum) for each
distinct age group.
        System.out.println(RED+"*****Calculate the salary range (minimum
and maximum) for each distinct age group.*****"+RESET);
        list.stream()
        .collect(Collectors.
                    groupingBy(Employee::getAge,
Collectors.collectingAndThen(Collectors.toList(),
                                employees->
{
    double min = employees.stream()
                            .mapToDouble(Employee::getS
alary)
                            .min().orElseThrow();
    double maxs = employees.stream()
                            .mapToDouble(Employee::getS
alary)
                            .max().orElseThrow();
    Map<String,Double> map = new
HashMap<>();
    map.put("min", min);
    map.put("max", maxs);
    return map;
}))..
        forEach((age,salary)->
        System.out.println
        ("Age : "+age+ " - "+"Min Salary : "+salary.get("min")+" Max Salary :
"+salary.get("max")));

//        45. Filter Employees by First Name Initial:
//              - Retrieve employees whose first name starts with a
specific initial (e.g., 'A').
        System.out.println(RED+"*****Retrieve employees whose first name
starts with a specific initial (e.g., 'E').*****"+RESET);
        list.stream()
        .filter(k->k.getName().startsWith("E"))
        .forEach(System.out::println);

//        46. Group Employees by Age and List Only Unique Salaries:
//              - Group employees by age and display a list of unique
salaries for each age group.
        System.out.println(RED+"*****Group employees by age and display a
list of unique salaries for each age group.*****"+RESET);
        list.stream()
        .collect(Collectors.
                    groupingBy(Employee::getAge,
Collectors.mapping(Employee::getSalary,
Collectors.toSet())))
                    .forEach((key,value)->
System.out.println("Age : "+key + " Salary : "+value));

```

```

//      47. Find Employees with the Same Salary:
//          - Identify and display employees who have the same salary.
System.out.println(RED+"*****Identify and display employees who have
the same salary."+RESET);
list.stream()
.collect(Collectors.groupingBy(Employee::getSalary))
.entrySet()
.stream() // k is entry
.filter(k -> k.getValue().size()>1)
.forEach(entry -> // Map (Double , List<>)
{
    System.out.println(entry.getKey());
    entry.getValue().forEach(System.out::println);
});

//      48. Find the Employee with the Shortest Name Among Male Employees:
//          - Find the male employee with the shortest name.
System.out.println(RED+"*****Find the male employee with the
shortest name."+RESET);
Employee employee5 = list.stream()
.filter(k->k.getGender().equals("Male"))
.min((o1, o2) -> Integer.compare(o1.getName().length(),
o2.getName().length()))
.orElseThrow();
System.out.println(employee5);

//      49. Find the Most Common Salary Value:
//          - Determine the salary value that appears most frequently
among the employees.
System.out.println(RED+"*****Determine the salary value that
appears most frequently among the employees."+RESET);
Double orElseThrow4 = list.stream()
.collect(Collectors.groupingBy(Employee::getSalary ,
Collectors.counting()))
.entrySet()
.stream()
.max(Map.Entry.comparingByValue())
.map(Map.Entry::getKey).orElseThrow();
System.out.println(orElseThrow4);

//      50. Find the Oldest Employee with the Lowest Salary:
//          - Find the oldest employee with the lowest salary.
System.out.println(RED+"*****Find the oldest employee with the
lowest salary."+RESET);
Employee employee6 = list.stream()
.filter(k->
k.getAge()==list.stream().mapToInt(Employee::getAge).max().orElseThrow()
.min(Comparator.comparingDouble(Employee::getSalary)).get());
System.out.println(employee6);

//      51. Filter Employees by Gender:
//          - Retrieve a list of all female employees.
System.out.println(RED+"***** Retrieve a list of all female
employees."+RESET);
list
.stream()
.filter(k->k.getGender().equals("Female"))
.forEach(System.out::println);

//      52. Filter Employees by Age:
//          - Get a list of employees older than 30 years.
System.out.println(RED+"*****Get a list of employees older

```

```

than 30 years."+RESET);
        list
        .stream()
        .filter(k->k.getAge()>30)
        .forEach(System.out::println);

//      53. Filter Employees by Salary:
//          - Find employees with a salary greater than $50,000.
System.out.println(RED+"*****Find employees with a salary greater
than $50,000."+RESET);
        list.stream()
        .filter(k->k.getSalary(>50000)
        .forEach(System.out::println);

//      54. Map Employee Names:
//          - Create a list of employee names (Strings).
System.out.println(RED+"*****Create a list of employee names
(Strings)."+RESET);
        list.stream()
        .map(k->k.getName())
        .toList().forEach(System.out::println);

//      55. Calculate Average Salary:
//          - Calculate the average salary of all employees.
System.out.println(RED+"*****Calculate the average salary of all
employees."+RESET);
        Double collect3 = list.stream()
        .collect(Collectors.averagingDouble(Employee::getSalary));
        System.out.println(collect3);

//      56. Find Maximum Salary:
//          - Find the employee with the highest salary.
System.out.println(RED+"*****Find the employee with the highest
salary."+RESET);
        Employee employee7= list.stream()
        .collect(Collectors.maxBy(Comparator.comparingDouble(Employee::getSa
lary))).get();
        System.out.println(employee7);

//      57. Group Employees by Gender:
//          - Group employees by gender
//          and return a map with gender as the key and a list of
employees as the value.
System.out.println(RED+"*****return a map with gender as the key
and a list of employees."+RESET);
        list.stream()
        .collect(Collectors.groupingBy(Employee::getGender))
        .forEach((k,v)->System.out.println(k+" - "+v));

//      58. Count Male Employees:
//          - Count the number of male and female employees.
System.out.println(RED+"****Count the number of male and female
employees."+RESET);
        list.stream()
        .collect(Collectors.groupingBy(Employee::getGender ,
Collectors.counting()))
        .forEach((k,v)->System.out.println(k+" - "+v));

//      59. Sum of All Salaries:
//          - Calculate the total sum of salaries for all employees.
System.out.println(RED+"*****Calculate the total sum of salaries
for all employees."+RESET);

```

```

        Double collect4 = list.stream()
            .collect(Collectors.summingDouble(Employee::getSalary));
        System.out.println(collect4);

//      60. Sort Employees by Name:
//          - Sort the employees by their names in alphabetical order.
        System.out.println(RED+"***** Sort the employees by their names in
alphabetical order."+RESET);
        list.stream()
            .sorted(Comparator.comparing(Employee::getName)).forEach(System.out:
:println);

//      61. Find the Employee with the Highest Salary in Each Gender:
//          - Find the employee with the highest salary for each gender
(male and female).
        System.out.println(RED+"*****Find the employee with the highest
salary for each gender (male and female)."++RESET);
        list.stream()
            .collect(Collectors.toMap(Employee::getGender, t -> t,(t, u) ->
t.getSalary()>=u.getSalary() ? t : u))
            .forEach((k,v)->System.out.println(k+" - "+v));

//      62. Retrieve Employees with Unique Names:
//          - Find employees with unique names (no duplicate names in
the list).
        System.out.println(RED+"*****Find employees with unique names (no
duplicate names in the list)."++RESET);
        list
            .stream()
            .collect(Collectors.groupingBy(Employee::getName ,
Collectors.counting()))
            .entrySet()
            .stream()
            .filter(k->k.getValue()==1)
            .forEach(k->System.out.println(k.getKey()));

//      63. Find the Names of Employees in Uppercase:
//          - Get a list of employee names in uppercase.
        System.out.println(RED+"*****Get a list of employee names in
uppercase."++RESET);
        list
            .stream()
            .filter(k->k.getName().equals(k.getName().toUpperCase()))
            .forEach(System.out::println);

//      64. Calculate the Salary Range (Min-Max) for Each Age Group:
//          - Calculate the salary range (minimum and maximum) for each
distinct age group.
        System.out.println(RED+"*****Calculate the salary range (minimum
and maximum) for each distinct age group."++RESET);
        Map<Integer,Map<String,Double>> collect5 = list.stream()
            .collect(Collectors.groupingBy(Employee::getAge,
                Collectors.collectingAndThen(Collectors.toList(), k ->
{
            Double maxx = k.stream().mapToDouble(k2-
>k2.getSalary()).max().getAsDouble();
            Double min = k.stream().mapToDouble(k1-
>k1.getSalary()).min().getAsDouble();
            Map<String, Double> map = new HashMap<>();
            map.put("max", maxx);
            map.put("min", min);
            return map;
})));
    
```

```

collect5.forEach((age, map)->
{
    System.out.print("Age : "+age+ " - ");
    System.out.println("Minimum : "+map.get("min")+" Maximum :
"+map.get("max"));
});

//      65. Filter Employees by First Name Initial:
//              - Retrieve employees whose first name starts with a
specific initial (e.g., 'E').
System.out.println(RED+"*****Retrieve employees whose first
name starts with a specific initial *****"+RESET);
list.stream()
.filter(k->k.getName().startsWith("E"))
.forEach(System.out::println);

//      66. Group Employees by Age and List Only Unique Salaries:
//              - Group employees by age and display a list of unique
salaries for each age group.
System.out.println(RED+"*****Group employees by age and display a
list of unique salaries for each age group.*****"+RESET);
list.stream()
.collect(Collectors.groupingBy(Employee::getAge,
                    Collectors.mapping(Employee::getSalary,
Collectors.toSet())))
.forEach((k,v)->System.out.println(k+ " - "+v));

//      67. Find Employees with the Same Salary:
//              - Identify and display employees who have the same salary.
System.out.println(RED+"*****Identify and display employees who
have the same salary.*****"+RESET);
list.stream()
.collect(Collectors.groupingBy(Employee::getSalary))
.entrySet()
.stream()
.filter(k->k.getValue().size()>1)
.forEach(t ->
{
    System.out.println(t.getKey() + " " + t.getValue());
});

//      68. Find the Employee with the Shortest Name Among Male Employees:
//              - Find the male employee with the shortest name.
System.out.println(RED+"*****Find the male employee with the
shortest name.*****"+RESET);
Employee employee8 = list.stream().filter(k-
>k.getGender().equals("Male"))
.min(Comparator.comparing(Employee::getName)).get();
System.out.println(employee8);

//      69. Find the Most Common Salary Value:
//              - Determine the salary value that appears most frequently
among the employees.
System.out.println(RED+"*****Determine the salary value that
appears most frequently among the employees.*****"+RESET);
Double double1 = list.stream()
.collect(Collectors.groupingBy(Employee::getSalary ,
Collectors.counting()))
.entrySet()
.stream()
.max(Map.Entry.comparingByValue())
.map(Map.Entry::getKey).get();
System.out.println(double1);

```

```

//      70. Find the Oldest Employee with the Lowest Salary:
//          - Find the oldest employee with the lowest salary.
System.out.println(RED+"*****Find the oldest employee with the
lowest salary."+RESET);
Employee employee9 = list.stream()
.filter(k-
>k.getAge()==list.stream().mapToInt(Employee::getAge).max().getAsInt())
.min(Comparator.comparingDouble(Employee::getSalary)).get();
System.out.println(employee9);

//      71. Find the Most Common Age Among Employees:
//          - Determine the age that is most common among the
employees.
System.out.println(RED+"*****Determine the age that is most common
among the employees."+RESET);
Integer integer = list.stream()
.collect(Collectors.groupingBy(Employee::getAge ,
Collectors.counting()))
.entrySet()
.stream()
.max(Map.Entry.comparingByValue())
.map(Map.Entry::getKey)
.get();
System.out.println(integer);

//      72. Find the Employee with the Longest Name:
//          - Find the employee with the longest name.
System.out.println(RED+"*****Find the employee with the longest
name."+RESET);
Employee employee10 = list.stream()
.max(Comparator.comparingInt(value ->
value.getName().length())).get();
System.out.println(employee10);

//      73. Find Employees with Palindromic Names:
//          - Retrieve employees whose names are palindromes (e.g.,
"Anna" or "Bob").
System.out.println(RED+"*****Retrieve employees whose names are
palindromes"+RESET);
list.stream()
.filter(k->
{
    String mainString = k.getName().toLowerCase();
    StringBuffer sr = new StringBuffer(mainString);
    return mainString.contentEquals(sr.reverse());
}).forEach(System.out::println);

//      74. Calculate the Sum of Salaries for Employees with Odd Ages:
//          - Calculate the sum of salaries for employees with odd
ages.
System.out.println(RED+"*****the sum of salaries for employees
with odd ages."+RESET);
double sum3 = list.stream()
.filter(k->k.getAge()%2!=0)
.mapToDouble(Employee::getSalary)
.sum();
System.out.println(sum3);

//      75. Find the Employee with the Highest Salary Whose Name Contains
"Smith":
//          - Find the employee with the highest salary whose name
contains the word "Smith."
System.out.println(RED+"*****Find the employee with the highest
salary whose name contains the word \"Smith.\\"+RESET);

```

```

Employee employee11 = list.stream()
    .filter(k->k.getName().contains("Smith"))
    .max(Comparator.comparingDouble(Employee::getSalary)).get();
System.out.println(employee11);

// 76. Group Employees by the First Letter of Their Names:
//       - Group employees by the first letter
//             of their names and return a map with
//                   the letter as the key and a list of employees
// as the value.
System.out.println(RED+"*****Group employees by the first letter of
their names****"+RESET);
list.stream()
    .collect(Collectors.groupingBy(t -> t.getName().charAt(0)))
    .forEach((k,v)->System.out.println(k+" - "+v));

// 77. Find the Employee with the Shortest Name:
//       - Find the employee with the shortest name.
System.out.println(RED+"*****Find the employee with the shortest
name.*****"+RESET);
Employee employee12 = list.stream()
    .min(Comparator.comparingInt(value -> value.getName().length()))
    .get();
System.out.println(employee12);

// 78. Calculate the Average Salary of Employees Whose Names Start with
"E":
//       - Calculate the average salary of employees whose names
start with the letter "E."
System.out.println(RED+"*****Calculate the average salary of
employees whose names start with the letter \"E.\\""+RESET);
Double collect6 = list.stream()
    .filter(k->k.getName().startsWith("E"))
    .collect(Collectors.averagingDouble(Employee::getSalary));
System.out.println(collect6);

// 79. Filter Employees by Age Range:
//       - Filter employees
//             based on a custom age range (e.g., between 25 and 35 years
old).
System.out.println(RED+"*****based on a custom age range (e.g.,
between 25 and 35 years old)*****"+RESET);
list.stream()
    .filter(k->k.getAge()>=25 && k.getAge()<=35)
    .forEach(System.out::println);

// 80. Group Employees by the First Two Letters of Their Names:
//       - Group employees by the first two letters
//             of their names and
//                   return a map with the letters as the key and a list of
employees as the value.
System.out.println(RED+"*****Group employees by the first two
letters*****"+RESET);
list.stream()
    .collect(Collectors.groupingBy(k->k.getName().substring(0, 2)))
    .forEach((k,v)->System.out.println(k+" - "+v));

// 81. Find the Employee with the Longest Name Whose Salary Is Below
$70,000:
//       - Find the employee with the longest name whose salary is
below $70,000.
System.out.println(RED+"*****Find the employee with the longest
name whose salary is below $70,000.**"+RESET);
Employee employee13 = list.stream().filter(k->k.getSalary()<70000)

```

```

    .max(Comparator.comparingInt(k->k.getName().length())).get();
    System.out.println(employee13);

//          82. Calculate the Average Salary of Employees Whose Names End with
// "son":
//                  - Calculate the average salary of employees whose names end
// with the suffix "son."
//                  System.out.println(RED+"*****Calculate the average salary of
employees whose names end with the suffix \"son.\\""+RESET);
//                  Double collect7 = list.stream()
//                      .filter(k->k.getName()
//                          .endsWith("na")).collect(Collectors.averagingDouble(Empl
oyee::getSalary));
//                  System.out.println(collect7);

//          83. Group employees by the number of
// words in their names (e.g., one-word names, two-word names, etc.)
// and return a map with the count as the key and a list of employees
as the value.
//                  list.stream()
//                      .collect(Collectors.groupingBy(k->k.getName().length()))
//                      .forEach((k,v)->System.out.println(k+" - "+v));

//          84. Calculate the Average Salary of Employees Whose Names Contain
Both "A" and "E":
//                  - Calculate the average salary of employees whose names
contain both the letters "A" and "E."
//                  System.out.println(RED+"*****the average salary of employees
whose names contain both the letters \"A\" and \"E.\\"*****"+RESET);
//                  double asDouble = list.stream()
//                      .filter(k->k.getName().contains("A")&&k.getName().contains("E"))
//                      .mapToDouble(Employee::getSalary).average().orElse(0.0);
//                  System.out.println(asDouble);
}

-----  

Multithreading Notes :  

-----  

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 under main group.

Program that describes that main is a Thread :

Whenever we define main method then JVM will create main thread internally under main group, the purpose of this main thread to execute the entire main method.

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;

public class ThreadDemo
{
    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("Name of the current thread is :" + name);
    }
}
```

-----  
How to create user-defined thread ?

-----  
We can create user-defined thread by using the following two packages

- 1) By using java.lang package
- 2) By using java.util.concurrent package

-----  
Creating user-defind Thread by using java.lang package :

-----  
By using java.lang package we can create user-defined thread by using anyone of the following two approaches :

- 1) By extending java.lang.Thread class
- 2) By implementing java.lang.Runnable interface

Note :- Thread is a predefined class available in java.lang package where as Runnable is a predefined interface available in java.lang Package.

-----  
10-07-2024  
-----

How to create user thread by using extending Thread class approach :

```
-----  
package com.ravi.multi_threading;  
  
class MyThread extends Thread  
{  
    @Override  
    public void run()  
    {  
        System.out.println("Child Thread is running!!!");  
    }  
}  
  
public class UserThreadDemo  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main Thread started....");  
        MyThread mt = new MyThread();  
        mt.start();  
        System.out.println("Main Thread ended.....");  
    }  
}
```

Here we have two threads, main thread which is responsible to execute main method and Thread-0 thread which is responsible to execute run() method. [10-JULY-24]

In entire Multithreading start() is the only method which is responsible to create a new thread.

-----  
public synchronized void start() :

start() is a predefined non static method of Thread class which internally performs the following two tasks :

- 1) It will make a request to the O.S to assign a new thread for concurrent execution.
- 2) It will implicitly call run() method.

-----  
public boolean isAlive() :-

It is a predefined non static method of Thread class through which we can find out whether a thread has started or not ?

As we know a new thread is created/started after calling start() method so if we use isAlive() method before start() method, it will return false but if the same isAlive() method if we invoke after the start() method, it will return true.

We can't restart a thread in java if we try to restart then It will generate an exception i.e java.lang.IllegalThreadStateException

-----  
package com.ravi.basic;

```

class Foo extends Thread
{
    @Override
    public void run()
    {
        System.out.println("Child thread is running...");
        System.out.println("It is running with separate stack");
    }
}
public class IsAlive
{
    public static void main(String[] args)
    {
        System.out.println("Main Thread is started..");
        Foo f = new Foo();
        System.out.println("Thread has not started yet so :"+f.isAlive());

        f.start(); //new Thread has created

        System.out.println("Thread has started so :"+f.isAlive());
        f.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 value is :" + i + " by " + name + " thread");
    }
}

public class ThreadLoop
{
    public static void main(String[] args)
    {
        new Sample().start();

        String name = Thread.currentThread().getName();

        for(int i=1; i<=10; i++)
        {
            System.out.println("i value is :" + i + " by " + name + " thread");
        }

        int x = 1;
        do
        {
            System.out.println("India");
            x++;
        }
        while(x<=10);
    }
}

```

Note : Here processor is frequently switching from main thread to Thread-0 thread so output is un-predictable

-----  
How to set and get the name of the Thread :

-----  
Whenever we create a userdefined Thread in java then by default JVM assigns the name of thread is Thread-0, Thread-1, Thread-2 and so on.

If a user wants to assign some user defined name of the Thread, then Thread class has provided a predefined method called `setName(String name)` to set the name of the Thread.

On the other hand we want to get the name of the Thread then Thread class has provided a predefined method called `getName()`.

```

public final void setName(String name) //setter
public final String getName() //getter
-----
package com.ravi.basic;
class DoStuff extends Thread
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " thread is running Here!!!!");
    }
}

```

```

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

        t1.start();
        t2.start();

        System.out.println(Thread.currentThread().getName()+" thread is
running.....");
    }
}

```

Note :- If we don't provide our user-defined name for the thread then by default the name would be Thread-0, Thread-1, Thread-2 and so on.

---

```

package com.ravi.basic;
class Demo extends Thread
{
    @Override
    public void run()
    {
        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 Here..");
    }
}

```

In the above program we have assigned user-defined name to the Thread so it will take the name given by the user.

---

11-07-2024

---

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.

It is a static method of Thread class.

```
Thread.sleep(1000); //Thread will wait for 1 second
```

It is throwing a checked Exception i.e InterruptedException because there may be chance that this sleeping thread may be interrupted by a thread so provide either try-catch or declare the method as throws.

```
-----  
package com.ravi.basic;  
  
class Sleep extends Thread  
{  
    @Override  
    public void run()  
    {  
        for(int i=1; i<=10; i++)  
        {  
            System.out.println("i value is :" + i);  
            try  
            {  
                Thread.sleep(1000);  
            }  
            catch(InterruptedException e)  
            {  
                System.err.println(e);  
            }  
        }  
    }  
}
```

```
}  
public class SleepDemo  
{  
    public static void main(String[] args)  
    {  
        Sleep s = new Sleep();  
        s.start();  
    }  
}
```

```
-----  
package com.ravi.basic;  
  
class MyTest extends Thread  
{  
    @Override  
    public void run()  
    {  
        Thread thread = Thread.currentThread();  
        System.out.println("Child Id is " + thread.getId());  
  
        for(int i=1; i<=5; i++) //11 22 33 44 55  
        {  
            System.out.println("i value is :" + i);  
            try  
            {  
                Thread.sleep(1000);  
            }  
            catch(InterruptedException e)  
            {  
                System.err.println("Thread has Interrupted");  
            }  
        }  
    }  
}
```

```
public class SleepDemo1
{
    public static void main(String[] args)
    {
        Thread thread = Thread.currentThread();
        System.out.println("Id Of Main Thread "+thread.getId());

        MyTest m1 = new MyTest();
        MyTest m2 = new MyTest();

        m1.start();
        m2.start();
    }
}
```

Assignment :

Thread.sleep(long millis, int nanos)

Thread life cycle :

As we know a thread is well known for Independent execution and it contains a life cycle which internally contains 5 states (Phases).

During the life cycle of a thread, It can pass from thses 5 states. At a time a thread can reside to only one state of the given 5 states.

- 1) NEW State (Born state)
- 2) RUNNABLE state (Ready to Run state) [Thread Pool]
- 3) RUNNING state
- 4) WAITING / BLOCKED state
- 5) EXIT/Dead state

New State :-

Whenever we create a thread instance(Thread Object) a thread comes to new state OR born state. New state does not mean that the Thread has started yet only the object or instance of Thread has been created.

Runnable state :-

Whenever we call start() method on thread object, A thread moves to Runnable state i.e Ready to run state. Here Thread 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 ?

As we know, In order to create a thread, Thread class must be super class so instead of making Thread class as a super class. We can use Anonymous inner class concept.

1) Anonymous inner class with reference variable :

```
package anonymous_demo_wth_reference;

public class AnonymousWithRef {

    public static void main(String[] args)
    {
        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println("Thread name is :" + name);
            }
        };

        t1.start();
        String name = Thread.currentThread().getName();
        System.out.println("Current Thread name is :" + name);
    }
}
```

2) Anonymous inner class without reference variable :

```
package anonymous_demo_wth_reference;

public class AnonymousWithoutRef {

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

```

        new Thread()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println("Thread name is :" + name);
            }
        }.start();
    }

}

```

---

12-07-2024

join() method of Thread class :

The main purpose of join() method to put the current thread into waiting state until the other thread finish its execution.

Here the currently executing thread stops its execution and the thread goes into the waiting state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state.

It also throws checked exception i.e InterruptedException so better to use try catch or declare the method as throws.

It is a non static method so we can call this method with the help of Thread object reference.

```

package com.ravi.basic;

class Join extends Thread
{
    @Override
    public void run()
    {
        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 Ended");
    }

}

-----
package com.ravi.basic;

class Alpha extends Thread
{
    @Override
    public void run()
    {
        Thread t = Thread.currentThread();
        String name = t.getName();    //Alpha_Thread is current thread

        Beta b1 = new Beta();
        b1.setName("Beta_Thread");
        b1.start();
        try
        {
            b1.join(); //Alpha thread is waiting 4 Beta Thread to complete
            System.out.println("Alpha thread re-started");
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        for(int i=1; i<=10; i++)
        {
            System.out.println(i+" by "+name);
        }
    }
}

public class JoinDemo2
{
    public static void main(String[] args)
    {
        Alpha a1 = new Alpha();
        a1.setName("Alpha_Thread");
        a1.start();
    }
}

class Beta extends Thread
{
    @Override
    public void run()
    {
        Thread t = Thread.currentThread();
        String name = t.getName();
        for(int i=1; i<=15; i++)
        {
            System.out.println(i+" by "+name);
            try

```

```

        {
            Thread.sleep(500);
        }
    catch(InterruptedException e) {
        }
    }
    System.out.println("Beta Thread Ended");
}
-----
package com.ravi.basic;

public class JoinDemo1
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main Thread Started");

        Thread t = Thread.currentThread();
        String name = t.getName();

        for(int i=1; i<=10; i++)
        {
            System.out.println("i value is :" + i + " by " + name);
            Thread.sleep(500);
        }
        t.join(); //Main thread is waiting for main thread to complete
        // [Deadlock]

        System.out.println("Main Thread Ended");
    }
}

```

Here, It is a deadlock state because main thread is waiting for main thread to complete.

Assignment :

```

join(long millis)
join(long millis, long nanos)
-----
Assigning target by Runnable interface :
-----
package com.ravi.basic;

class Ravi implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();
        System.out.println(name + " thread is running Here");
    }
}
public class RunnableDemo
{
    public static void main(String [] args)
    {
        System.out.println("Main Thread is Running");

        Ravi r1 = new Ravi();
    }
}

```

```
    Thread thread = new Thread(r1,"Child1");
    thread.start();
}

-----
```

13-07-2024

-----

How to assign the target for different Threads :

-----

```
package com.ravi.runnable_demo;

class RegularBatch implements Runnable
{
    @Override
    public void run()
    {
        System.out.println("Regular Batch is going on");
    }
}

class PlacementBatch implements Runnable
{
    @Override
    public void run()
    {
        System.out.println("Placement Batch is going on");
    }
}

public class RunnableTarget {

    public static void main(String[] args) throws InterruptedException
    {
        Thread t1 = new Thread(new PlacementBatch());
        t1.start();

        Thread t2 = new Thread(new RegularBatch());
        t2.start();

    }
}
```

-----

Thread class constructor :

-----

We have total 9 constructors in the Thread class, The following are commonly used constructor in the Thread class

- 1) Thread t1 = new Thread();
- 2) Thread t2 = new Thread(String name);
- 3) Thread t3 = new Thread(Runnable target);
- 4) Thread t4 = new Thread(Runnable target, String name);
- 5) Thread t5 = new Thread(ThreadGroup tg, String name);
- 6) Thread t6 = new Thread(ThreadGroup tg, Runnable target);

```

7) Thread t7 = new Thread(ThreadGroup tg, Runnable target, String name);
-----
Case 1 :
-----
Anonymous inner class by using Runnable interface :
-----
package com.ravi.runnable_demo;

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("Current thread name is :" + name);

            }
        };
        Thread t1 = new Thread(r1);
        t1.start();
    }
}

Case 2 :
-----
Anonymous inner class as a parameter to Thread Constructor :
-----
package com.ravi.runnable_demo;

public class AnonymousToThreadConstructor
{
    public static void main(String[] args)
    {
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                System.out.println("Current thread name is :" + name);
            }
        }, "Child1").start();
    }
}

Case 3 :
-----
Runnable interface by using Lambda :
-----
package com.ravi.runnable_demo;

```

```

public class LambdaByRunnable
{
    public static void main(String[] args)
    {
        Runnable r1 = ()->
        {
            String name = Thread.currentThread().getName();
            System.out.println("Current thread name is :" + name);
        };

        new Thread(r1).start();
    }
}

-----
Case 4 :
-----
package com.ravi.runnable_demo;

public class AnonymousToThreadConstructor
{
    public static void main(String[] args)
    {
        new Thread(()->
System.out.println(Thread.currentThread().getName(), "Child1").start();

    }
}

-----
Program to assign different task to different Thread :
-----
package com.ravi.runnable_demo;

class GetCube
{
    public void getCubeOfTheNumber(int num)
    {
        System.out.println("Cube of the number is :" + (num * num * num));
    }
}

public class GettingCubeByMultipleThreads
{
    public static void main(String[] args)
    {
        GetCube cube = new GetCube();

        new Thread()
        {
            @Override
            public void run()
            {
                cube.getCubeOfTheNumber(5);
            }
        }.start();

        new Thread()
        {
            @Override
            public void run()
            {

```

```

        cube.getCubeOfTheNumber(9);
    }

    }.start();
}

}
-----
```

IQ :

---  
In between extends Thread and implements Runnable which one is better and why?

In between extends Thread and implements Runnable, implements Runnable is more better approach due to the following reasons.

- 1) In extend Thread class approach, We can't extend any other class further (Multiple Inheritance not possible by using class) but we can implement multiple interfaces. On the other hand in implements Runnable approach we can extend a class as well as implement multiple interfaces.
- 2) In extends Thread class all the Thread class properties are available to sub class so sub class is heavy weight, the same thing is not available while implementing Runnable interface.
- 3) In extends Thread class approach we have same target for different Threads but by using implements we can provide different target for different Threads.
- 4) In extends Thread class approach we don't have Lambda expression support but by using Runnable interface we can implement Lambda.

Conclusion :

Implements Runnable Advantage

- 1) extend a single class
- 2) Light weight
- 3) Assigning different targets (Loose coupling)
- 3) Implementing Lambda

-----  
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.(13-JULY)

```

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+" seat is not available");
    }

}

public class RailwayReservation
{
    public static void main(String[] args) throws InterruptedException
    {
        Customer cust = new Customer(1);

        Thread t1 = new Thread(cust,"Virat");
        Thread t2 = new Thread(cust,"Rohit");

        t1.start(); t2.start();

    }
}

```

Here both the Threads will get the ticket

---

```

package com.ravi.mt;

class Customer
{
    private double currentbalance = 20000;
    private double withdrawAmount;

    public Customer(double withdrawAmount)
    {
        super();
        this.withdrawAmount = withdrawAmount;
    }

    public void withdrawAmount()
    {
        String name = null;
        if(currentbalance >= withdrawAmount)
        {
            name = Thread.currentThread().getName();
            System.out.println(withdrawAmount+ " amount withdraw by
"+name);
            this.currentbalance = this.currentbalance - withdrawAmount;
        }
        else
    }
}

```

```

        {
            name = Thread.currentThread().getName();
            System.out.println("Sorry " +name+ " U have insufficient bal");
        }
    }

}

public class BankingApplication
{
    public static void main(String[] args)
    {
        Customer c1 = new Customer(20000);

        Runnable r1 = ()-> c1.withdrawAmount();

        Thread t1 = new Thread(r1,"Person1");
        Thread t2 = new Thread(r1,"Person2");

        t1.start(); t2.start();
    }
}

```

Here both the threads are getting the balance.

-----  
15-07-2024  
-----

\*\*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.(15-JUL-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 (15-JUL-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.

-----  
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 :  
-----  
MethodLevelSynchronization.java  
-----  
package com.ravi.multi_threaing;  
  
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
```

```

        Runnable r1 = ()-> obj.printTable(5);
        Runnable r2 = ()-> obj.printTable(10);

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        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();
    }
}

```

#### Limitation/Drawback of Object Level Synchronization :

From the given diagram it is clear that there is no interference between t1 and t2 thread because they are passing 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 15-JULY-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(8); //lock2
            }
        };

        Thread t4 = new Thread()
        {
            @Override
            public void run()
            {
                pt2.printTable(9); //lock2
            }
        };
        t1.start();      t2.start();  t3.start();  t4.start();
    }
}

```

Note : From the above program, It is clear that even our method is synchronized but due to multiple locks (multiple object) we are getting concurrency.

In order to avoid the drawback of Object level synchronization, static synchronization is introduced.

Static Synchronization :

If we make our synchronized method as a static method then it is called static synchronization.

Here, To call static synchronized method, object is not required.

The thread will take the lock from class but not object because we can call the static method with the help of class name.

Unlike Object, we cann't create multiple classes in the same package.

```
package com.ravi.advanced;
class MyTable
{
    public static synchronized void printTable(int n) //static
synchronization
    {
        for(int i=1; i<=10; i++)
        {
            try
            {
                Thread.sleep(100);
            }
            catch(InterruptedException e)
            {
                System.err.println("Thread is Interrupted...");
            }
            System.out.println(n+" X "+i+" = "+(n*i));
        }
        System.out.println("-----");
    }
}
public class StaticSynchronization
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread()
        {
            @Override
            public void run()
            {
                MyTable.printTable(5);
            }
        };

        Thread t2 = new Thread()
        {
            @Override
            public void run()
            {
                MyTable.printTable(10);
            }
        };

        Runnable r3 = ()-> MyTable.printTable(15);
    }
}
```

```
        Thread t3 = new Thread(r3);

        t1.start();
        t2.start(); t3.start();

    }
-----
16-07-2024
-----
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`.

-----  
Program describes that Child thread created as a part of main thread will get the same Priority because the child thread is created as a part of main thread.

```
package com.ravi.thread_priority;

public class PriorityDemo1
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println("Main Thread priority is :" + t.getPriority());

        Thread t1 = new Thread();
        System.out.println("Child Thread priority is :" + t1.getPriority());
    }
}
```

-----  
package com.ravi.thread\_priority;

```
public class PriorityDemo2
{
    public static void main(String[] args)
    {
```

```

        Thread t = Thread.currentThread();
        t.setPriority(Thread.MIN_PRIORITY);
        System.out.println("Main Thread Priority is :"+t.getPriority());

        new Thread(()-> System.out.println("Child Thread Priority
is :"+Thread.currentThread().getPriority())).start();

    }

}

```

Note : From the above program it is clear that child thread created as a part of main thread will acquire main thread priority.

---

```

package com.ravi.thread_priority;

public class PriorityDemo3 {

    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        t.setPriority(11); //java.lang.IllegalArgumentException
        System.out.println(t.getPriority());

    }

```

---

```
package com.ravi.thread_priority;
```

```

class Priority
{
    public void accept()
    {
        int count =0;
        for(int i=1; i<=100000; i++)
        {
            count++;
        }

        System.out.println("After completion name of the Thread
is :"+Thread.currentThread().getName());
        System.out.println("After completion Priority of the Thread
is :"+Thread.currentThread().getPriority());
    }
}

```

```
public class PriorityDemo4 {
```

```

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

        Runnable r1 = ()-> p.accept();

        Thread t1 = new Thread(r1,"LastThread");
        t1.setPriority(Thread.MIN_PRIORITY);

        Thread t2 = new Thread(r1,"FirstThread");
        t2.setPriority(Thread.MAX_PRIORITY);

        t1.start(); t2.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 : By using yield() we can make a request to thread scheduler to avoid over utilisation of CPU by current Thread.

-----  
ThreadGroup :

-----  
It is a predefined class available in java.lang Package.

By using ThreadGroup class we can put 'n' number of threads into a single group to perform some common operation.

By using ThreadGroup class constructor, we can assign the name of group under which all the thread will be executed.

```
ThreadGroup tg = new ThreadGroup(String groupName);
```

ThreadGroup class has provided the following method :

```
public String getName() : To get the name of the Group
```

```
public int activeCounts() : How many thread are running under that  
particular group.
```

Thread class has provided constructor to put the thread into particular group.

```
Thread t1 = new Thread(ThreadGroup tg, Runnable target, String name);
```

By using ThreadGroup class, multiple threads will be executed under single group.

-----  
package om.ravi.thread\_group;

```
class JavaClass implements Runnable  
{  
    @Override  
    public void run()  
    {  
        String name = Thread.currentThread().getName();  
        for(int i=1; i<=3; i++)  
        {  
            System.out.println(i+" by "+name);  
        }  
    }  
}
```

```
public class ThreadGroupDemo1  
{
```

```
    public static void main(String[] args)  
    {  
        ThreadGroup tg = new ThreadGroup("Java_Group");  
  
        Thread t1 = new Thread(tg, new JavaClass(), "Child 1" );  
        Thread t2 = new Thread(tg, new JavaClass(), "Child 2" );  
        t1.start();  
        t2.start();
```

```
        System.out.println("Thread group name is :" +tg.getName());  
        System.out.println("Total threads which are Running in this  
group :" +tg.activeCount());  
    }  
}
```

```
-----  
package om.ravi.thread_group;  
  
public class ThreadGroupDemo2 {  
  
    public static void main(String[] args)  
    {  
        Thread t = Thread.currentThread();  
        System.out.println(t.toString());  
    }  
}
```

Output : Thread[main, 5, main]

Here first main is the name of the Thread, 5 is the priority and last main represents group name.

Whenever we define a main method then internally, main group is created and under this main group main thread is executed.

```
-----  
package om.ravi.thread_group;  
  
class TatkalTicket implements Runnable  
{  
    @Override  
    public void run()  
    {  
        String name = Thread.currentThread().getName();  
        System.out.println("Tatkal ticket booked by :" + name);  
    }  
}  
  
class PremiumTatkal implements Runnable  
{  
    @Override  
    public void run()  
    {  
        String name = Thread.currentThread().getName();  
        System.out.println("Premium Tatkal ticket booked by :" + name);  
    }  
}  
  
public class ThreadGroupDemo3  
{  
    public static void main(String[] args)  
    {  
        ThreadGroup tg1 = new ThreadGroup("Tatkal Ticket");  
        ThreadGroup tg2 = new ThreadGroup("Premium Tatkal");  
  
        Thread t1 = new Thread(tg1, new TatkalTicket(), "Scott");  
        Thread t2 = new Thread(tg2, new PremiumTatkal(), "Smith");  
  
        t1.start(); t2.start();  
    }  
}  
-----  
** 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() method from the same object.

public native final void notify() :-

-----  
It will wake up the single thread that is waiting on the same object. It will not release the lock immediatly, once synchronized area is completed then only it will release the lock.

Once waiting thread will get the notification from the another thraed using notify()/notifyAll() method then the waiting thread will move from Waiting state to Runnable state(Ready to run state)

public native final void notifyAll() :-

-----  
It will wake up all the threads which are waiting on the same object. It will not release the lock immediatly, once synchronized area is completed then only it will release the lock.

\*Note :- wait(), notify() and notifyAll() methods are defined in Object class but not in Thread class because these methods are related to lock(because we can use these methods from the synchronized area ONLY) and Object has a lock so, all these methods are defined inside Object class.

The following program explains, if we don't use these methods from synchronized are then we will get java.lang.IllegalMonitorStateException

```
package com.ravi.inter_thread_communication;

public class Test
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Wait Method");
        Object obj = new Object();
        obj.wait();
    }
}
```

-----  
The following program explains, If we don't use co-ordination between two threads then we will get different output.

```

package com.ravi.inter_thread_communication;

class ThreadA extends Thread
{
    int val = 0;

    @Override
    public void run()
    {
        try
        {
            Thread.sleep(200);
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }

        for(int i=1; i<=10; i++)
        {
            val = val + i; // 1 3 6 10 15
        }
    }
}

public class ITCProblem
{
    public static void main(String[] args) throws InterruptedException
    {
        ThreadA a1 = new ThreadA();
        a1.start();

        Thread.sleep(200);

        System.out.println(a1.val);
    }
}

```

Here main thread is printing the value of val variable but var variable is frequently changing based on the loop iteration so may get some different output.

---

```

-----  

package com.ravi.inter_thread_communication;

class SecondThread extends Thread
{
    int val = 0;

    @Override
    public void run()
    {
        synchronized(this)
        {
            for(int i=1; i<=100; i++)
            {
                val = val + i;
            }
            System.out.println("Sending notification");
            notify();
        }
        //Here actually the lock will be released
    }
}

```

```

}

public class ITCDemo1
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main Thread started!!");

        SecondThread b = new SecondThread();
        b.start();

        synchronized(b)
        {
            System.out.println("Main Thread is waiting Here");
            b.wait();
            System.out.println("Main Thread Wake up");
            System.out.println(b.val);

        }
    }

}

```

Here main thread will wait after releasing the lock so child thread will get the lock, complete the task and send the notification by using notify() method [Here notify() method will not release the lock, the lock is available again for main thread after completing synchronized area of child thread], main thread will get the lock once again to complete its remaining task.

```

-----
package com.ravi.inter_thread_communication;

class Customer
{
    private double balance = 10000;

    public synchronized void withdraw(double amount)
    {
        System.out.println("Going To Withdraw!!!");
        if(amount > balance)
        {
            System.out.println("Less Amount, Waiting 4 deposit");
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {

            }
        }

        this.balance = this.balance - amount;
        System.out.println("Amount after Withdraw is :" +this.balance);

    }

    public synchronized void deposit(double amount)
    {
        System.out.println("Going to deposit!!!");
    }
}
```

```

        this.balance = this.balance + amount;
        System.out.println("Balance After deposit :" + this.balance);
        notify();
    }

}

public class ITCBalance
{
    public static void main(String[] args) throws InterruptedException
    {
        Customer c1 = new Customer();

        Thread son = new Thread()
        {
            @Override
            public void run()
            {
                c1.withdraw(15000);
            }
        };
        son.start();

        Thread.sleep(1000);

        Thread father = new Thread()
        {
            @Override
            public void run()
            {
                c1.deposit(10000);
            }
        };
        father.start();
    }
}
-----
```

18-07-2024

```

-----
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 made 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");

        t1.start();
        t2.start();
        t3.start();

        Thread setter = new Thread(() -> r1.setMethod(), "Setter_Thread");

        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        setter.start();
    }
}

```

#### interrupt Method of Thread class :

It is a predefined non static method of Thread class. The main purpose of this method to disturb the execution of the Thread, if the thread is in waiting or sleeping state.

Whenever a thread is 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 (Exception e)  
            {  
                System.err.println("Thread is Interrupted ");  
                e.printStackTrace();  
            }  
        }  
    }  
}  
public class InterruptThread  
{  
    public static void main(String[] args)  
    {  
        Interrupt it = new Interrupt();  
        System.out.println(it.getState()); //NEW  
        it.start();  
        it.interrupt(); //main thread is interrupting the child thread  
    }  
}
```

Note : Here main thread has interrupted child thread so child thread will execute the catch block only one time because sleep is interrupted only once.

-----

```
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)
    {
        Interrupt it = new Interrupt();
        it.start();
    }
}

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 by locking the resource");
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Thread interrupted gracefully.");
        }
        finally
        {
            System.out.println("Thread resource can be release here.");
        }
    }
}

```

Deadlock :

It is a situation where two or more than two threads are in blocked state forever, here threads are waiting to acquire another thread resource without releasing it's own resource.

This situation happens when multiple threads demands same resource without releasing its own attached resource so as a result we get Deadlock situation and our execution of the program will go to an infinite state as shown in the diagram. (18-JULY-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) which is a non static method Thread class.

```
-----  
public class DaemonThreadDemo1  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Main Thread Started...");  
  
        Thread daemonThread = new Thread(() ->  
        {  
            while (true)  
            {  
                System.out.println("Daemon Thread is running...");  
                try  
                {  
                    Thread.sleep(1000);  
                }  
                catch (InterruptedException e)  
                {  
                    e.printStackTrace();  
                }  
            }  
        });  
  
        daemonThread.setDaemon(true);  
        daemonThread.start();  
  
        Thread userThread = new Thread(() ->  
        {  
            for (int i = 1; i <= 9; i++)  
            {  
                System.out.println("User Thread: " + i);  
                try  
                {  
                    Thread.sleep(2000);  
                }  
                catch (InterruptedException e)  
                {  
                    e.printStackTrace();  
                }  
            }  
        });  
    }  
}
```

```
        }
    });

    userThread.start();

    System.out.println("Main Thread Ended...");
}
-----
```

Networking in java :

IP Address :

An IP address is a unique identification number allocated to each and every device connected through the network.

By using this IP address we can recognise a client in the network. IP address contains some bytes which identify the network and the actual computer inside the network.

Eg:- 192.168.100.09 (ipconfig is the command in command propmpt)

Finding the IP Address of a server :

It is possible to find out the IP address of any website on internet. In order to do the same, java.net package has provided a predefined class called InetAddress which contains a static method getByName(String host) through which we can find out the IP address of any website.

Ex :- InetAddress.getByName("www.google.com");

Here getByName() will return the IP address of Google.com website.

The following are the important methods of InetAddress class :

1) public static InetAddress getByName(String host) throws UnknownHostException :-

It will return the IP address of the specified host.

2) public static InetAddress getLocalHost() throws UnknownHostException :-

It will return both the IP address and name of the local system.

3) public String getHostName() :- It will return the name of the System

4) public String getHostAddress() :-It will the return the IP Address of the System

---

//Program to find out the local host IP address and Name.

```
import java.net.*;
public class Inet
{
    public static void main(String args[]) throws UnknownHostException
    {
        InetAddress ia = InetAddress.getLocalHost();
        System.out.println("Local Name and IP Address : "+ia);
```

```

        }
    }

    /**
     * Program to find out the name and address of local machine
     */
    public class Inet3
    {
        public static void main(String args[]) throws UnknownHostException
        {
            InetAddress ia = InetAddress.getLocalHost();
            String addr = ia.getHostAddress();
            String name = ia.getHostName();
            System.out.println("My host name is :" + name + " My address is : "
                    + addr);
        }
    }

    /**
     * Name and IP address both at the same time
     */
    class LocalHost
    {
        public static void main(String [] args) throws UnknownHostException
        {
            System.out.println(InetAddress.getLocalHost());
        }
    }

    /**
     * program to find out the IP address of the given host
     */
    import java.net.*;
    import java.io.*;
    class IpAddress
    {
        public static void main(String[] args ) throws IOException
        {
            BufferedReader br = new BufferedReader
                (new InputStreamReader(System.in));

            System.out.print("Enter the host name: ");
            String host = br.readLine();
            try
            {
                InetAddress ia = InetAddress.getByName(host);
                System.out.println("IP address of "+host+" is : " + ia);
            }
            catch(UnknownHostException e )
            {
                System.err.println(e);
            }
        }
    }

    /*
     *.com
     *.in
     *.ac.in
     *.edu
     *.gov.in
    */
    /**
     * URL class :
     */

```

-----  
It is a predefined class available in java.net package. It stands for Uniform Resource Locator. The URL class represents the address that we specify at browser URL to access some resource.

Example of URL:-

<https://www.gmail.com:25/index.jsp>

From this above URL

- 1) Protocol Name :- https (getProtocol())
  - 2) server name or host name :- www.gmail.com (getHost())
  - 3) port number :- 25 (-1 will be the value, if not supplied by the website) (getPort())
  - 4) File name :- index.jsp (getFile())
- 

```
import java.net.*;
public class URLInfo
{
public static void main(String[] args)
{
    try
    {
        URL url=new URL("https://www.gmail.com:110/index.jsp");
        System.out.println("Protocol: "+url.getProtocol());
        System.out.println("Host Name: "+url.getHost());
        System.out.println("Port Number: "+url.getPort());
        System.out.println("File Name: "+url.getFile());
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}
}
```

-----

--  
URLConnection class :

-----  
It is a predefined class available in java.net package. This class is useful to connect to a website or a resource in the network, it will fetch all the details of the specified web page as a part of URL class.

The URL class provides a method called openConnection(), this method will establish a connection with the specified web page and returns the URLConnection object, that is nothing but the complete details of the web page.

public URLConnection openConnection() throws IOException

-----

```
---
import java.io.*;
import java.net.*;
public class Details
{
public static void main(String[] args)
{
    try
    {
        URL url=new URL("https://www.python.org/downloads/");
}
```

```
//Establishing the connection from the Specified URL
URLConnection urlcon=url.openConnection();

InputStream stream = urlcon.getInputStream();

while(true)
{
    int i = stream.read();
    if(i==-1)
        break;
    System.out.print((char)i);

}
catch(Exception e)
{
    System.out.println(e);
}
}
```

---

25-Feb-23

-----

Socket :-

-----

It is possible to establish a logical connection point between the server and the client so that communication can be done through this point is called Socket.

Now, each socket is given an identification number which is called port number. The range of port number is 0-65535.

Establishing a communication between the client and server using socket is called Socket Programming.

Socket programming can be connection oriented or connection less. In java Socket (for client) and ServerSocket (for server) are the predefined classes available in java.net package for connection oriented Socket Programming.

The client socket programming must have two information.

- 1) Ip address of the server
- 2) port number

The following are the important methods of Socket class :

-----

- 1) public InputStream getInputStream();
- 2) public OutputStream getOutputStream();
- 3) public synchronized void close();

The following are the important methods of ServerSocket class :

-----

- 1) public Socket accept() :- It is used to put the server in wait mode till a client accept or establish the connection
- 2) public InputStream getInputStream();
- 3) public OutputStream getOutputStream();

```
4) public synchronized void close();
```

---

Creating a server that can send some data :

We can create a Socket that can be used to connect to a server and a client. Once the Socket is created, the Server can send the data to the client and client can receive the data.

All we have to do is just to send the data from server to socket. The socket will take care of whom to send the data on the network.

We should write the following java code to create a server that can send some data to the client.

1) At server side, create a server socket with some port number, This is done by using ServerSocket class

```
ServerSocket ss = new ServerSocket(7777);
```

2) We should make the server wait till a client accept connections. This can be done by using accept().

```
Socket s = ss.accept();
```

3) Now Attach some output stream to ServerSocket using getOutputStream() method. This method returns OutputStream object. This method is used to send the data from Socket to the client

```
OutputStream obj = s.getOutputStream();
```

4) Take another Stream like PrintStream or DataOutputStream to send the data till the Socket.

```
PrintStream ps = new PrintStream(obj);
```

5) Now to print the data which we are sending from Server to the client we can use print() or println() method available in PrintStream class.

```
ps.println(String data);
```

6) Now close all the connections.

```
ss.close();
s.close();
ps.close();
```

Creating a client that can receive some data :-

We can write client program that receives all the String sent from server to client machine. We should write the following java code

1) We should create a Socket at client side by using Socket class as

```
Socket s = new Socket("Ip address", port number);
```

Note :- If the client and server both are running in a single machine then it is called "localhost", we should write localhost instead of IP address.

2) We should add InputStream to the Socket so that the Socket will be able to

receive the data on the InputStream.

```
InputStream obj = s.getInputStream();
```

3) Now to read the data from Socket to the client machine we can take the help of BufferedReader as

```
BufferedReader br = new BufferedReader(new InputStreamReader(obj));
```

4) Now We can read the data from the BufferedReader as

```
String str = br.readLine();
```

5) close the connection

```
br.close();
s.close();
-----
```

-----  
Server1.java  
-----

```
//Program to send String to the client
import java.io.*;
import java.net.*;
class Server1
{
    public static void main(String args[]) throws IOException
    {
        ServerSocket ss = new ServerSocket(777);

        Socket s=ss.accept();

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

        OutputStream obj=s.getOutputStream();

        PrintStream ps = new PrintStream(obj);

        String str="Hello Client";
        ps.println(str);
        ps.println("Bye-Bye");
        ps.close();
        s.close();
        ss.close();
    }
}
```

-----  
Client1.java  
-----

```
import java.io.*;
import java.net.*;
class Client1
{
    public static void main(String args[]) throws Exception
    {
        Socket s = new Socket("localhost",777);

        InputStream obj = s.getInputStream();

        BufferedReader br = new BufferedReader(new InputStreamReader(obj));
```

```

        String str;
        while((str=br.readLine()) !=null)
        {
            System.out.println("From Server :"+str);
        }
        br.close();
        s.close();
    }
}

-----
-----  

Chating between Client and Server  

-----
Server2.java  

-----
//A server that receives and send the data
import java.io.*;
import java.net.*;
class Server2
{
    public static void main(String [] args) throws Exception
    {
        ServerSocket ss = new ServerSocket(888);
        Socket s = ss.accept();

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

        PrintStream ps = new PrintStream(s.getOutputStream());

        BufferedReader br = new BufferedReader
            (new InputStreamReader( s.getInputStream()));

        BufferedReader kb = new BufferedReader(new
InputStreamReader(System.in));

        while(true)
        {
            String recv,send; //send is for sending and recv is for
receiving the data

            while((recv=br.readLine()) !=null)
            {
                System.out.println(recv);
                send=kb.readLine();
                ps.println(send);
            }
            ps.close();
            br.close();
            kb.close();
            s.close();
            ss.close();
            System.exit(0);//Shutdown the JVM
        }
    }
}

-----
Client2.java  

-----
//A client that receives and send the data
import java.io.*;
import java.net.*;
class Client2
{

```

```

public static void main(String [] args) throws Exception
{
    Socket s = new Socket("localhost",888);
    DataOutputStream dos = new DataOutputStream(s.getOutputStream());
    BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
    BufferedReader kb = new BufferedReader(new
InputStreamReader(System.in));
    String send,recv; //send is for sending and recv is for receiving
the data
    while(! (send=kb.readLine()).equals("exit")) //Hello Server
    {
        dos.writeBytes(send+"\n");
        recv=br.readLine();
        System.out.println(recv);
    }
    dos.close();
    br.close();
    kb.close();
    s.close();
}
-----
```

//Program to download the content of a file from the Server, if the file is available at server.

FileServer.java

```

-----//A server that can send file content to the client
import java.io.*;
import java.net.*;
class FileServer
{
    public static void main(String [] args ) throws Exception
    {
        ServerSocket ss = new ServerSocket(8888);
        Socket s = ss.accept();
        System.out.println("Connection established...");
        BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        DataOutputStream out = new DataOutputStream(s.getOutputStream());
        String fname = in.readLine();      //fname = abc.txt
        boolean flag;
        //File is existing or not
        File f = new File(fname);
        if(f.exists())
            flag=true;
        else
            flag=false;
        //Sending the acknowledgement message to the client
        if(flag==true)
```

```

        out.writeBytes("yes"+'\n');
    else
        out.writeBytes("no"+'\n');

//Reading the File content and sending it to the client
if(flag==true)
{
    FileReader fr=new FileReader(fname);
    BufferedReader file = new BufferedReader(fr);

    String str;
    while((str=file.readLine()) !=null)
    {
        out.writeBytes(str+'\n');
    }
    file.close();
    fr.close();
    out.close();
    in.close();
    s.close();
    ss.close();
}
}

```

```

FileClient.java
-----
//A Client receiving a file content
import java.io.*;
import java.net.*;
class FileClient
{
    public static void main(String [] args ) throws Exception
    {
        Socket s = new Socket("localhost",8888);

        BufferedReader kb = new BufferedReader(new
InputStreamReader(System.in));

        System.out.print("Enter a file name :");

        String fname = kb.readLine(); //fname = abc.txt

        DataOutputStream out = new DataOutputStream(s.getOutputStream());

        out.writeBytes(fname+'\n'); //first of all client is sending file
name

        BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));

        String str = in.readLine();

        if(str.equalsIgnoreCase("yes"))
        {
            while((str=in.readLine()) !=null)
                System.out.println(str);
            kb.close();
            out.close();
            in.close();
            s.close();
        }
        else
    }
}

```

```
{  
    System.out.println("File not found at Server location..");  
}  
}  
=====
```









