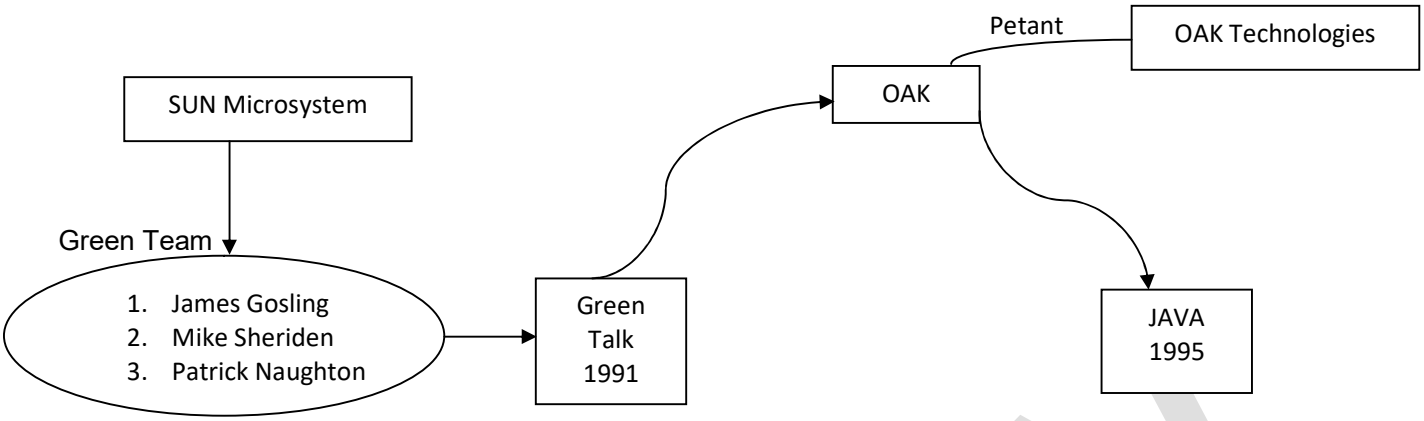
# Index

# ★ Introduction:-




**Computer:-**

- Computer is an electronic device which is made up of using hardware and software.

**Language:-**

- Language is a mode or medium of communication between two or more entities (object).
- Language is used to share information.

**Object:-**

- Object is a real world entity.
- E.g. Akshay, Tanmay, Vishwajeet, chair, duster, etc.

**Human Language:-**

- The language which is used by humans for communication is known as human language.
- E.g. Marathi, Hindi, etc.

**Internet:-**

- Internet is a global network of machines.
- Every machine is connected through internet.

**Programming Language:-**

- Language which is used to command a computer (machine) to perform a specific task is known as programming language.

**Programmer:-**

- A person who has knowledge of programming language is known as programmer.

**Binary Language:-**

- A language which is made up of two characters that are '0' and '1' is known as binary language.
- Typically this '0's and '1's are in set of 8 or 16 characters sequence.
- Binary language is a language of machine that's why its also known as machine understandable language.

**Source Code:-**

- Source code is a file which contains set of instructions written by a programmer using a programming language.
- Every source code contains its language extension.
- E.g. If we are creating a source code for java the extension while saving the file must be ".java"

**Programming:-**

- It is process of creating set of instructions using a programming language is known as programming.

## Compiler:-

- Compiler is an intermediate software which takes source code as an input.
- Compiler compiles (read and checks) the whole source code at once.
- It checks for syntactical mistakes, if found it throws a compile time error.
- Compiler displays all error at once.
- The process of compiling the source and finding the mistakes is known as compilation.
- If there is no syntactical mistake in the code compiler generates a new file i.e. known as byte code.

## Byte Code:-

- Byte code is an intermediate file generated by the compiler.
- Byte code name is same as class name and its extension is "**.**class".
- A byte code is given as a input to the JVM (interpreter).

## Interpreter:-

- Interpreter is used to interpret the byte code line by line.
- It checks for the logical mistake.
- It displays one error at a time.
- Error given by the interpreter is known as runtime error.

| | Compiler | | Interpreter |
|---|---|---|---|
| i. | Compiler converts high level language to low level language. | i. | Interpreter converts high level language to low level language using JIT (just in time) compiler. |
| ii. | It compiles the whole source code at once. | ii. | It interprets the byte code line by line. |
| iii. | Compiler checks for syntactical mistake. | iii. | Interpreter checks for logical mistake. |
| iv. | Compiler displays all error at once. | iv. | Interpreter displays one error at a time. |
| v. | Debugging is difficult. | v. | Debugging is easy. |
| vi. | Compiler faster in compilation. | vi. | Interpreter is slower. |
| vii. | Error thrown/given by the compiler is known as compile time error. | vii. | Error given by the interpreter is known as runtime error. |
| viii. | Compiler generates an intermediate file i.e. byte code. | viii. | Interpreter doesn't generate any file. |
| ix. | Compilation command:- javac filename.java | ix. | Interpreter/Execution command:- java Classname |

**Platform:-**

- A platform is a tool which is made up of hardware and software on which application runs.

**Software:-**

- Software is a set of instructions or block of code which is used to perform a specific task.

**Hardware :-**

- Hardware is a component which is physically present inside a platform such as CPU, Mouse, etc.

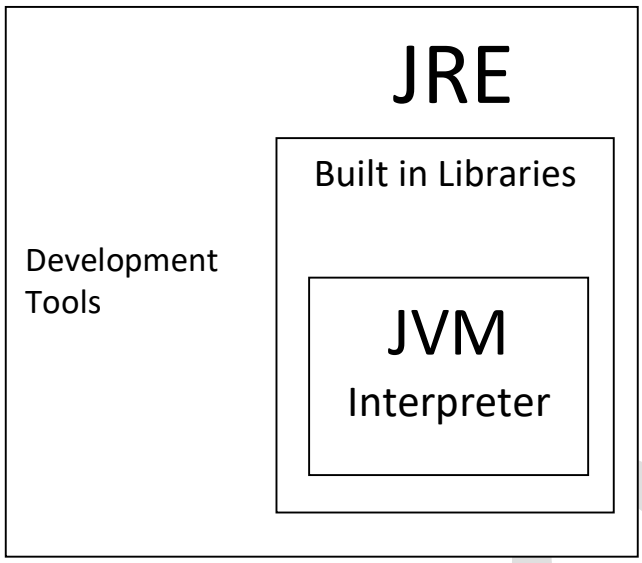
**Applications are divided into two types,**

1. Platform Dependent Application:-
2. Platform Independent Application:-

1. **Platform Dependent Application:-**
   - Application created on one specific platform can be executed (run) on that particular platform is known as platform dependent application.

2. **Platform Independent Application:-**
   - An application created on one platform can be executed on any other platform is known as platform independent application.

# ★ JDK Architecture:-

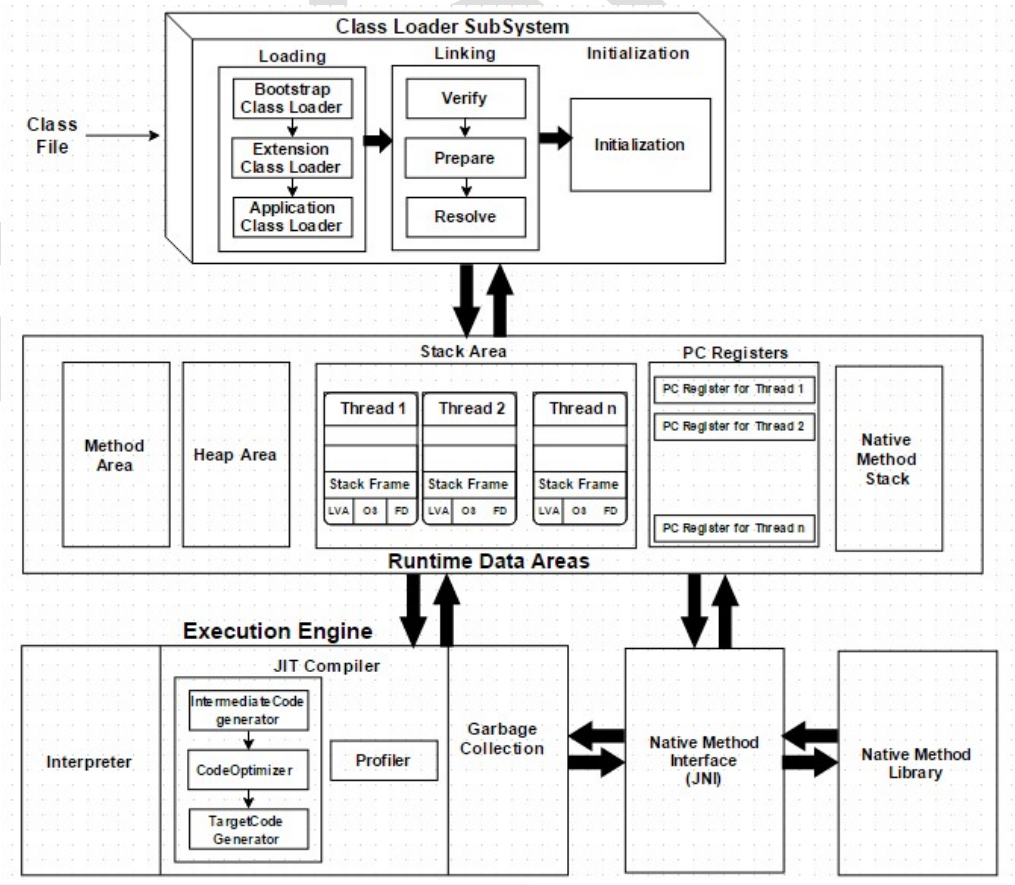



**JDK stands for Java Development Kit**

- It is a package provided by Oracle Corporation which is used to develop and execute a java application.
- It consists of development tools along with JRE.

**JRE stands for Java Runtime Environment**

- It provide us an integrated environment in which we can execute a java application.
- In case if we just want to execute a java application we can install **JRE**.
- JRE consists of built in libraries and **JVM** (**Java Virtual Machine**).



# ★ JVM architecture :-

- JVM is responsible for the execution of the byte code.
- JVM takes byte code as an input.
- It has three main components,
  1. Class loader subsystem
  2. JVM memory area
  3. Execution engine

**1. Class loader subsystem:-**

i. Class loader:-
- The class loader is responsible for loading compile class dynamically into the JVM memory area during runtime.

ii. Linking:-
- Linking is used to establish the connection between classes and interfaces.
- It has a subcomponent called as verifier which verifies the byte code has been compiled on a genuine compiler or not.

iii. Initialization:-
- In this step where JVM performs initialization logic for each loaded class or interface. (i.e. it invokes the constructor of class).

**2. JVM memory area:-**

i. Method area:-
- A method area stores the fields and method data in it.

ii. Heap Memory:-
- Objects in java are created at the runtime and they are stored in heap area.

iii. Stack:-
- In java the execution is done by a thread. For each thread JVM will create a runtime stack inside the stack area.

- For every single thread a **PC** register will be allocated and a native method stack.

**3. Execution engine:-**
- The execution engine consists of several of components,

i. Interpreter:-
- Interpreter is used to interpret a byte code line by line, but if we invoke a function n number of times the interpreter used to interpret the same function again and again which was making the execution slow.

ii. JIT:-
- JIT stands for 'Just In Time' compiler which was introduced in JDK1.1 to overcome the disadvantage of interpreter it is in association with a profiler.

iii. GC:-
- Garbage collector is a mechanism in java which is used to destroy an unwanted object from heap memory implicitly (automatically).
- Unwanted objects means the objects which does not have any reference.

**Q. Does java has destructor in it?**

→

- No, but it has a similar and better mechanism then that destructor i.e. known as garbage collector.
- Garbage collector is a mechanism in java which is used to destroy an unwanted object from heap memory implicitly.
- Destructor means it is used to destroy an object, in programming language like C++, a programmer has to invoke (call) the destructor to remove an object that makes the code more complex.
- So if we are not removing it, maybe we will be out of memory while execution.
- But in java the GC will remove the unwanted objects implicitly (automatically).

**Q. What is JAVA?**

→

- Java is a high level, object oriented programming language, which was developed to be platform independent.
- Java was created by James Gosling and his team (Green Team) at Sunmicrosystem in 1991.
- But currently owned by oracle corporation.
- The first version of java i.e. JDK 1.0 released in 1996.
- Popular version of java where major updates was introduced such as lambdas, stream API and many more features in the version 1.8 (2014).
- Current stable version available in the market is JDK 23.
- JAVA is statically type programming language and imperative programming paradigm.

**Features of JAVA:-**

1. **Platform Independent:-**
   - Java's byte code can be executed on any platform which has a JVM.

2. **Object oriented programming language:-**
   - Java follows the object oriented programming principles such as encapsulation, inheritance, polymorphism and abstraction which makes it object oriented.

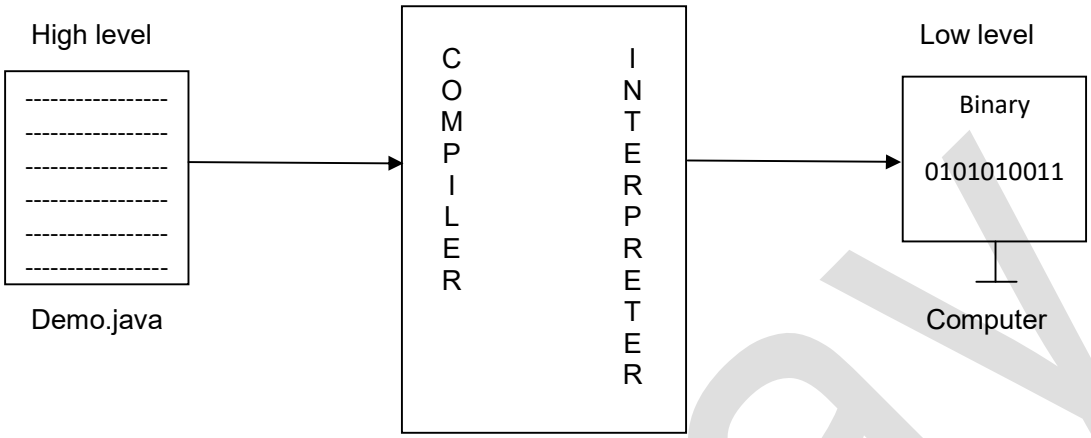
3. **Robust and Secure:-**
   - Java has features like memory management and strong type checking.
   - It has exception handling to ensure that a program doesn't get terminated in the middle of execution.

4. **Dynamic memory allocation:-**
   - Java supports dynamic memory allocation because it has a garbage collector in it.

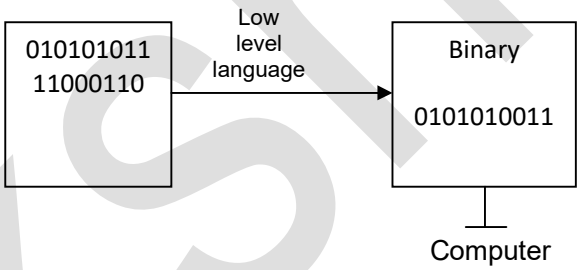
5. **Portable:-**
   - Java's "Write Once Run Anywhere" principle makes it highly portable.

## Languages can be divide based on their functionally such as,

1. **High level language:-**
   - A high level language is just easily understandable, readable and instructable by humans.
   - It is similar to human language.
   - e.g. **java**




2. **Low level language:-**
   - A language which easily understandable by machine is known as low level language.
   - Low level language is also known as machine understandable language.
   - e.g. **binary language**




3. **Mid level language:-**
   - A mid level language code is neither understandable by human or by a machine.
   - It contains some predefine set of words which are understandable by machine only through an intermediate software.
   - The predefined words are mnemonics.

**Structure of java program:-**

Demo.java ⟶ file name

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

- When we compile a source code we get a byte code which has a same name as a class name.
- Compiler generates n number of byte codes for n number of classes.
- We should always execute a byte code which has a main method in it.
- Because the JVM always searches for main method for the execution.
- We can create a source code and a class with a different name.

# ★ Tokens:-

- Tokens are the smallest unit of java programming language.
- There are six different types of tokens,
  1. Keywords
  2. Identifier
  3. Separator
  4. Operator
  5. Comments
  6. Literals

### 1. Keywords:-

- Keywords are the words which are aware by the compiler.
- These words are predefined and have specific meaning.
- We cannot alter (change) them.
- There are 52+ keywords in java and every keyword must be written in lowercase.
- Keywords,

| | | | | |
|---|---|---|---|---|
| abstract | default | goto | package | this |
| assert | do | if | private | throw |
| boolean | double | implements | protected | throws |
| break | else | import | public | transient |
| byte | enum | instanceof | return | true |
| case | extends | int | short | try |
| catch | false | interface | static | void |
| char | final | long | strictfp | volatile |
| class | finally | native | super | while |
| const | float | new | seitch | |
| continue | for | null | synchronized | |

**Note:-**

- Apart from keywords there are three reserve words in java that are true, false and null.
- true, false is a Boolean literal (data).
- null is used for non-primitive datatype.
- Reserve words are similar like keywords.

Program:-

```
class keywordExample
{
        public static void main(String[] args)
        {
                boolean     a    = true;
                String      str  = null;
                boolean     b    = false;
        }
}
```

## 2. Identifier:-

- Identifiers are the name given by a programmer to various java members such as class, method, variable, interface, package, etc.
- E.g.

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

- **Rules of identifier:-**
  1. It cannot start with a number.
  2. We cannot use keywords (reserve words) in place of identifier.
  3. Character spaces in between is not allowed.
  4. Only two special characters are allowed i.e. '$' and '_'.
  5. $ represents inner class / inner interface.

- e.g.
  - ✓ int num;
  - ✓ int num1;
  - X int n um1;
  - X int volatile;
  - X int true;
  - ✓ int fal_se;
  - ✓ int _num1;
  - ✓ int $num2;
  - ✓ int num$;
  - ✓ int num_;
  - X int 1num;

- **Program:-**

```
class Example
{
        public static void main(String[] args)
        {
                int  $ = 10;
                System.out.println($);   //10
                int  _ = 11;
                System.out.println(_);   //CTE
        }
}
```

**Note:-**

- '$' character is used to represent inner classes and inner interfaces in java.
- '_' it is used to represent a space.
- We cannot use just an underscore as an identifier name, because it will create an unnamed variable and also '_' is considered as a keyword from the release of JDK 1.9.

## ★ Industry Standards (Conventions):-

- Conventions are not mandatory but it is always recommended to use them.

**Convention for class:-**

- If a class name is single word then the first character must be in uppercase.
- If its multiword then every words first character must be in uppercase.
- The way of giving a class name is known as **'Pascal'** case.
- E.g. MyFirstJavaProgram, Demo, System, String, PrintStream

**Note:-**

- Convention for an **interface** is similar to a class.

**Convention for method:-**

- If a method name is single word all character must be in lowercase.
- If it is multiword from second word onwards every words first character must be in uppercase.
- The way of writing a method name is called as **'Camel'** case.
- E.g. addition, myMethod, isPrime, toUpperCase, toLowerCase, charAt, checkEvenOdd, println

**Note:-**

- Convention for a **variable** is same as a method.

### 3. Separator:-

- Separators are the special characters which are used to separate a block, statement or a java member.
- E.g. In java we have different separators such as,
  i. **{ }** curly braces
  ii. **( )** parenthesis
  iii. **: :** colons (method reference)
  iv. **;** semicolon (end of statement)
  v. **.** dot operator
  vi. **...** ellipsis (varargs)

- **Program:-**

```
class Example
{
    public static void main(String[] args)
    {
            int a, b, c;
            int [] arr = new int[10];
            System.out.println("Hello");
    }
}
```

### 4. Comments:-

- Comments are the non-executable statements.
- These statements are ignored by compiler as well as interpreter.
- They are most often used to explain logical part of a program.
- There are three types of comments,
  i. Single line comment
  ii. Multiline comment
  iii. Documentation comment

### i. Single line comment:-

- It is used to comment a single line of code.
- We use two forward slashes (//) prefix with the statement.
- E.g.
  //Akshay

**ii. Multiline comment:-**

- It is used to comment multiple lines of code.
- The statements which is needs to be commented must be placed in between ' **/*** ' and ' ***/** '
- E.g.

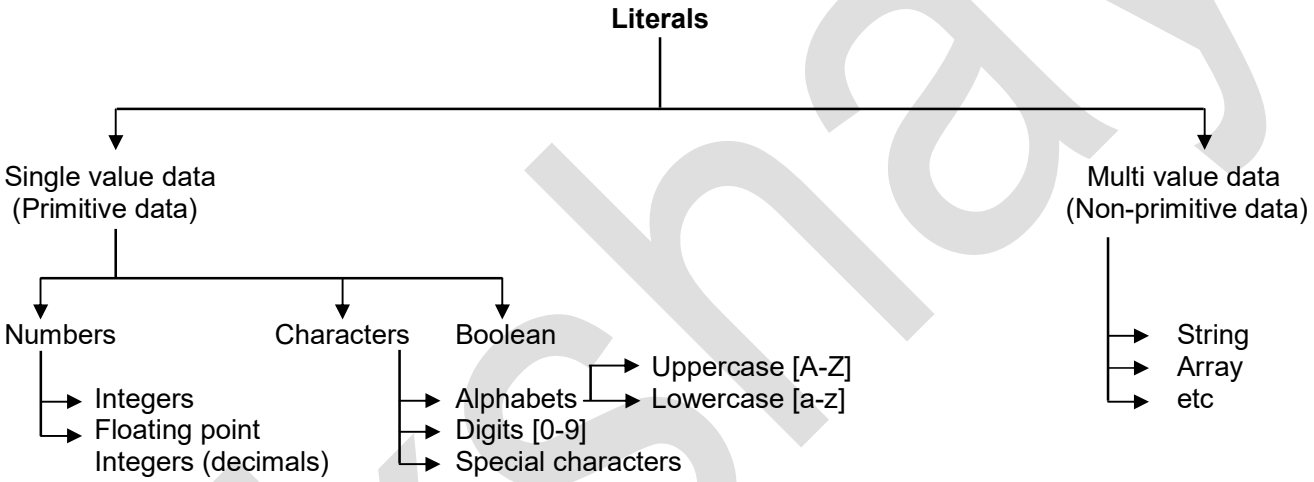
```
/* Akshay
Mali */
```

**iii. Documentation comment:-**

- The documentation comments are used for multiple lines to be commented.
- This is mostly used in creating API documents, defining a new class or an interface.
- E.g.

```
/**
*Here is block comment
*
*/
```

**5. Literals:-**




★ **Data types:-**

- Data type specifies the type of data and size of data which we want to store.

**OR**

- Data type helps the programmer for creating a container (variable).
- There are two types of data types,
  1. Primitive data type:-
  2. Non-primitive data type:-

**1. Primitive data type:-**

- The data type which is used to create a container (variable) to store single value data is known as primitive data type.
- E.g.

```
int num = 10;
boolean b = true;
```

- There are eight primitive datatypes,
  1. byte
  2. short
  3. int
  4. long
  5. float
  6. double
  7. char
  8. boolean

2. **Non-primitive data type:-**
   - The data type which is used to create a container to store multi value data is known as non-primitive data.

| Literal | Data type | Size (in bytes) | Range | Default value | Length |
|---|---|---|---|---|---|
| Integer | byte | 1 byte | -128 to 127 | 0 | |
| | short | 2 byte | -32768 to 32767 | 0 | |
| | int | 4 byte | -2147483648 to 2147483647 | 0 | |
| | long | 8 byte | | 0l / 0L | |
| Decimal | float | 4 byte | | 0.0f / 0.0F | |
| | double | 8 byte | | 0.0d / 0.0D | |
| Character | char | 2 byte | | /U0000 | |
| Boolean | boolean | 1 bit | | false | |

- The smallest unit of memory is 1 bit.
- In 1 bit we can store either 0 or 1.
- The memory is been distributed based on **2'** compliment.

2^0 = 1 1 bit = 0 and 1
2^1 = 2 1 byte = 8 bit
2^2 = 4 1 kilobyte = 1024 byte
2^3 = 8 1 megabyte = 1024 kilobyte
2^4 = 16 1 gigabyte = 1024 megabyte
2^5 = 32 1 terabyte = 1024 gigabyte
2^6 = 64 1 petabyte = 1024 terabyte
2^7 = 128 1 hexabyte = 1024 petabyte
2^8 = 256
2^9 = 512
2^10 = 1024

- **Types of numbers:-**
  i. Integers (decimal)(10):-
  Character sequence:- 0 1 2 3 4 5 6 7 8 9

  ii. Binary (2):-
  Character sequence:- 0 1

  iii. Hexadecimal(16):-
  Character sequence:- 0 1 2 3 4 5 6 7 8 9 A B C D E F

  iv. Octal(8):-
  Character sequence:- 0 1 2 3 4 5 6 7

## 1. byte:-

- byte is a primitive data type which is used to store integer literals.
- It is a keyword in java.
- Size of byte is 1 byte i.e. 8 bits.
- Range of byte is -128 to 127.
- The first bit is reserved for sign (0 for +ve ,1 for -ve).

**Program to find range of byte using wrapper class.**

```
class ByteExample
{
        public static void main(String[] args)
        {
                System.out.println("min="+Byte.MIN_VALUE);
                System.out.println("max="+Byte.MAX_VALUE);
        }
}
```

## ★ Variable:-

- Variables are the containers in which we store data.
- In java we can create variables with the help of datatype.

**Variable declaration statement: -**

- Statement which is used to declare a variable is called as variable declaration statement.
- Syntax :- datatype variablename ;
- E.g.
  short num;
  String name;
  long phno;
  char grade;

**Variable initialization statement: -**

- Statement which is used to store a value inside a variable.
- E.g
  num = 1000;
  name = "Akshay";
  phno = 8007929292;
  grade = 'z';

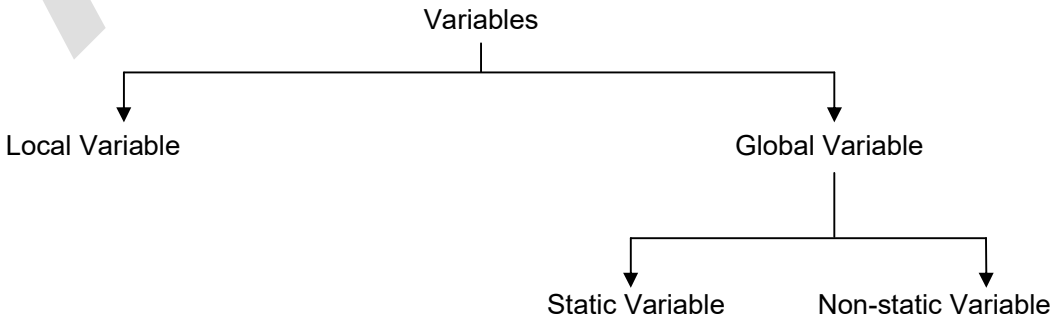
**Variable declaration and initialization statement: -**

- Declare a variable and store the value simultaneously.
- Syntax:- datatype variablename = value / expression ;
- E.g. :-
  String name = "Akshay";

**Note:-**

We cannot use a variable without declaring it.

- **Variables are classified into two types,**



1. **Global Variables:-**
   - The variables which are declared in the class block are known as global variables.
   - They are classified into two types,
     i. Static Variable
     ii. Non-static Variable

2. **Local Variables:-**
   - The variables which are declared in the method block or any other block other than class block is known as local variables.

★ **Characteristics of local variables:-**

i. A variable declared inside a block remains local (is only visible) within the same block and it cannot be accessed outside the block.
Program:-

```
class Local1
{
        public static void main(String[] args)
        {
                {
                        int a=100;
                }
                System.out.println(a);      //Compile time error
        }
}
```

ii. We cannot have more than one local variable of same name within the same block if we do it we get compile time error.
Program:-

```
class Local1
{
        public static void main(String[] args)
        {
                {
                        int a = 100 ;
                        System.out.println(a);
                        String a = “Akshay” ;
                        System.out.println(a); //Compile time error
                }
        }
}
```

iii. We can create more than one local variable of same name in two different blocks.
Program:-

```
class Local1
{
        public static void main(String[] args)
        {
                {
                        int a = 100 ;
                        System.out.println(a);
                }
                {
                        String a = “Akshay” ;
                        System.out.println(a);
                }
        }
}
```

iv. We cannot use a local variable without initialization because local variables are not assigned with default values by the compiler.
Program:-

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

# ★ Operators :-

- Operators are the symbols which are used to perform certain operations on the given operands (values).
- Operators are classified into two types,
  1. Based on the number of operand required,
     i. Unary Operator (1 operand)
     ii. Binary Operator (2 operand)
     iii. Ternary Operator (3 operand)
  2. On the basis of the type of operation it can perform,
     i. Arithmetic Operator
     ii. Assignment Operator
     iii. Relational Operator
     iv. Conditional Operator
     v. Compound Assignment Operator
     vi. Logical Operator
     vii. Bitwise Operator
     viii. Increment / Decrement Operator
     ix. Miscellaneous Operator

## i. Arithmetic Operator:-

- Arithmetic operators are binary operators.
- Arithmetic operators are used to perform basic mathematical calculations such as addition(+), subtraction(-), multiplication(*), division(/), modulus(%).

1. Addition Operator:-
   Addition operator used to finds out the sum of two operands (values).

| Operand 1 | Operator | Operand 2 | Output | Data type |
|---|---|---|---|---|
| 10 (byte) | + | 20 (byte) | 30 | int |
| 100 (short) | + | 200 (short) | 300 | int |
| ‘a’ (char) | + | ‘b’ (char) | 195 | int |
| 1000 (int) | + | 2000 (int) | 3000 | int |
| 5000l(long) | + | 1000l (long) | 6000 | long |
| 1.1f (float) | + | 2.2f (float) | 3.300002 | float |
| 1.1d (double) | + | 2.2d (double) | 3.30000000003 | double |
| true (boolean) | + | false (boolean) | Compile time error | - |
| “ABC” (String) | + | “EFG” (String) | ABCEFG | String |

2. Subtraction Operator:-
   Subtraction operator used to finds out the difference of two operands (values).

| Operand 1 | Operator | Operand 2 | Output | Data type |
|---|---|---|---|---|
| 10 (byte) | - | 20 (byte) | -10 | int |
| 100 (short) | - | 200 (short) | -100 | int |
| 'a' (char) | - | 'b' (char) | -1 | int |
| 1000 (int) | - | 2000 (int) | -1000 | int |
| 5000l(long) | - | 1000l (long) | 4000 | long |
| 1.1f (float) | - | 2.2f (float) | -1.1 | float |
| 1.1d (double) | - | 2.2d (double) | -1.1 | double |
| true (boolean) | - | false (boolean) | Compile time error | - |
| "ABC" (String) | - | "EFG" (String) | Compile time error | - |

3. Multiplication Operator:-
   Multiplication operator used to finds out product of two operands (values).

| Operand 1 | Operator | Operand 2 | Output | Data type |
|---|---|---|---|---|
| 10 (byte) | * | 20 (byte) | 200 | int |
| 100 (short) | * | 200 (short) | 20000 | int |
| 'a' (char) | * | 'b' (char) | 9506 | int |
| 1000 (int) | * | 2000 (int) | 2000000 | int |
| 5000l(long) | * | 1000l (long) | 5000000 | long |
| 1.1f (float) | * | 2.2f (float) | 2.42 | float |
| 1.1d (double) | * | 2.2d (double) | 2.4200000000004 | double |
| true (boolean) | * | false (boolean) | Compile time error | - |
| "ABC" (String) | * | "EFG" (String) | Compile time error | - |

4. Division Operator:-
   Division operator used to finds out division of two operands (values).

| Operand 1 | Operator | Operand 2 | Output | Data type |
|---|---|---|---|---|
| 10 (byte) | / | 20 (byte) | 0 | int |
| 100 (short) | / | 200 (short) | 0 | int |
| 'a' (char) | / | 'b' (char) | 0 | int |
| 1000 (int) | / | 2000 (int) | 0 | int |
| 5000l(long) | / | 1000l (long) | 5 | long |
| 1.1f (float) | / | 2.2f (float) | 0.5 | float |
| 1.1d (double) | / | 2.2d (double) | 0.5 | double |
| true (boolean) | / | false (boolean) | Compile time error | - |
| "ABC" (String) | / | "EFG" (String) | Compile time error | - |

5. Modulus Operator:-
   Modulus operator used to find out the division remainder of two operands (values).

| Operand 1 | Operator | Operand 2 | Output | Data type |
|---|---|---|---|---|
| 10 (byte) | % | 20 (byte) | 10 | int |
| 100 (short) | % | 200 (short) | 100 | int |
| 'a' (char) | % | 'b' (char) | 97 | int |
| 1000 (int) | % | 2000 (int) | 1000 | int |
| 5000l(long) | % | 1000l (long) | 0 | long |
| 1.1f (float) | % | 2.2f (float) | 1.1 | float |
| 1.1d (double) | % | 2.2d (double) | 1.1 | double |
| true (boolean) | % | false (boolean) | Compile time error | - |
| "ABC" (String) | % | "EFG" (String) | Compile time error | - |

# ii. Assignment Operator:-

- Assignment operator is a binary operator which is used to store value in the variable.

**Rule:-**

- The L.H.S of an assignment operator must be a variable.
- The R.H.S of an assignment operator can be value or expression or another variable.
- Syntax :- variable = value / expression / variable

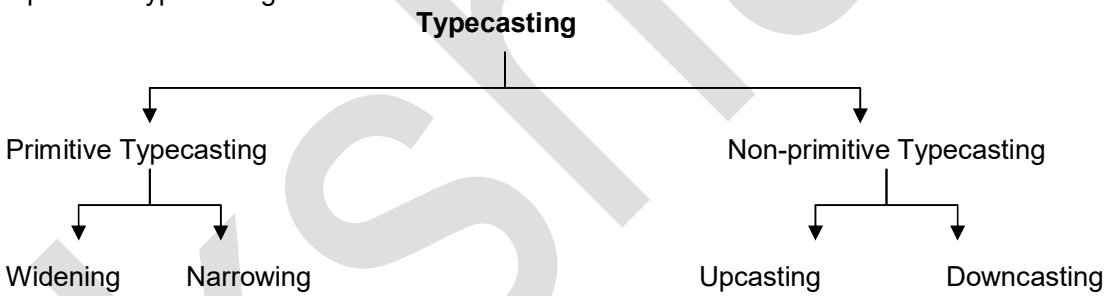
  ← (right-to-left)

  L.H.S R.H.S
- Assignment operator works from right to left (associativity) direction of working.
- 10 = 10 → wrong
  We cannot assign a value to a value.
  int var = 10 → correct
- The type of value to be stored must be same as the type of the given variable.
  String s = true; → wrong
  String s = "true"; → correct

**If the type of value is not same as the type of variable there are two possibilities.**

## ★ Typecasting for primitive datatype:-

- It is a process of converting or storing one type of data into another type.
- Typecasting is of two types,
  1. Primitive typecasting
  2. Non-primitive typecasting




### 1. Primitive Typecasting:-

- It is process of converting one type of primitive data into another primitive type.
- There are two types,
  i. Widening
  ii. Narrowing

### i. Widening:-

- Widening is a process of converting or storing smaller type data into larger type.
- Widening is performed implicitly performed by the compiler because when we perform widening there is no loss of data.
- **Program:-**

```
class Widening{
        public static void main(String[]  args){
                double d = 12;
                System.out.println(d);
                //int i = 12.12 ; CTE
                int o = 'p';
                System.out.println(o);
                //long l = 123.123f;      CTE
                float f = 1011;
                System.out.println(f);
                float f2 = 'x';
                System.out.println(f2);
        }
}
```

**Widening of byte datatype**

```
class WideningByte
{
	public static void main(String[] args)
	{
		byte a = 10 ;
		System.out.println(a);
		short s = a;
		System.out.println(s);
		//char c = a ; CTE
		//System.out.println(c);
		int i = a;
		System.out.println(i);
		long l =a;
		System.out.println(l);
		float f = a;
		System.out.println(f);
		double d = a;
		System.out.println(d);
	}
}
```

**Widening of short datatype**

```
class WideningByte
{
	public static void main(String[] args)
	{
		short s = 10;
		System.out.println(s);
		int i = s;
		System.out.println(i);
		long l = s;
		System.out.println(l);
		float f = s;
		System.out.println(f);
		double d = s;
		System.out.println(d);
	}
}
```

**Widening of int datatype**

```
class WideningByte
{
	public static void main(String[] args)
	{
		int i = 10;
		System.out.println(i);
		long l = i;
		System.out.println(l);
		float f = i;
		System.out.println(f);
		double d = i;
		System.out.println(d);
	}
}
```

**Widening of long datatype**

```
class WideningByte
{
	public static void main(String[] args)
	{
		long l = 10l;
		System.out.println(l);
		float f = l;
		System.out.println(f);
		double d = l;
		System.out.println(d);
	}
}
```

**Widening of float datatype**

```
class WideningByte
{
	public static void main(String[] args)
	{
		float f = 10.0f;
		System.out.println(f);
		double d = f;
		System.out.println(d);
	}
}
```

**Widening of char datatype**

```
class WideningChar
{
	public static void main(String[] args)
	{
		char c = 'a' ;
		System.out.println(c);
		int i = c;
		System.out.println(i);
		long  l = c;
		System.out.println(l);
		float f = c;
		System.out.println(f);
		double d = c;
		System.out.println(d);
	}
}
```

**ii. Narrowing :-**

- Narrowing is a process of converting or storing a larger type data into smaller type.
- Narrowing is not done implicitly by the compiler instead it has to be done explicitly by the programmer using **typecast** operator.

**Note: -**

Implicit narrowing is not possible because there is a possibility of loss of data.

**Syntax: -** ( datatype to converted ) value;

**E.g.: -**

```
int a = 12.23 ;		//CTE
int a = (int)12.23 ;	//No CTE
System.out.println(a);
```

**O/p: -** 12

**.**23 is lost after narrowing

**Program: -**

```
class Narrowing
{
        public static void main(String[] args)
        {
                double d = 123.2143343 ;
                float f = (float) d ;
                long l = (long) d ;
                int i = (int) d ;
                short s = (short) d ;
                byte b = (short) d;
                char c = (char) d ;
        }
}
```

**Assignment operator continued:-**

- If the type of value is not same as the type of variable there are two possibilities,
  1. If the type of value is smaller than the type of variable the compiler perform widening.
  2. If the type of value is bigger than the type of variable the compiler cannot perform narrowing implicitly hence we get compile time error.

```
class Narrow
{
        public static void main(String[] args)
        {
                double dist = 1530 ;
                double stepDist = 0.9982 ;
                int noOfSteps = (int) dist/stepDist ;
                System.out.println(noOfSteps+"needed to be complete"+dist+"meters");
        }
}
```

# iii. Compound Assignment Operator :-

- It is a binary operator, it is also known as update operator.
- They are,
  **+=**
  **-=**
  ***=**
  **/=**
  **%=**

- Compound assignment operators are used to assign a value to the variable by updating the old value in the variable.

**OR**

- A compound assignment operator performs the operations specified by the additional operators and then assigns the value / result to the variable (L.H.S.).

- Syntax:- var += value ;
  i.e. var = var + value ;

- In compound assignment operator the L.H.S must be a variable and R.H.S can be value or expression.

**Note:-**

Compound assignment operators can perform **implicit narrowing**.

**Program:-**

```
class UpdateOperator
{
	public static void main(String[] args)
	{
		int i = 10000 ;
		// i = i+1000;
		i += 1000;
		System.out.println(i);
	}
}
```

**Implicit Narrowing**

```
class Narrowing
{
	public static void main(String[] args)
	{
		int i = 100 ;
		// i = (int) (i+123.123) ;
		i += 123.123 ;
		System.out.println(i);
	}
}
```

# iv. Relational Operators:-

- Relational operators are binary operators which help us to compare two values and helps in decision making purpose.
- They are,

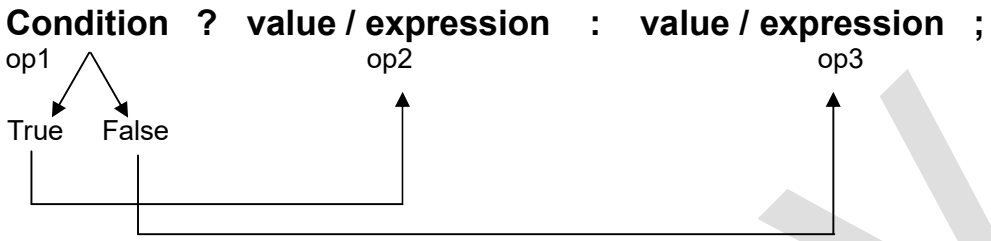
| | |
|---|---|
| == | Equality operator |
| != | Not equal operator |
| < | Lesser than operator |
| > | Greater than operator |
| <= | Less than equal to operator |
| >= | Greater than equal to operator |

```
class Relational
{
    public static void main(String[] args)
    {
        System.out.println(5>9);                //false
        System.out.println(5<9);                //true
        System.out.println(0==0.0);             //true
        System.out.println('a'==97);            //true
        System.out.println('a'!='A');           //true
        System.out.println(59.9999==60);        //false
        System.out.println(true>false);         //false
        System.out.println(true==false);        //false
        System.out.println(true==true);         //true
        System.out.println(1<=true);            //CTE
        System.out.println(100>=100);           //true
    }
}
```

# v. Conditional Operator:-

- It is a ternary operator.
- It is used for decision making purpose.

**Syntax:-**




- The condition must be of boolean type (op1).
- If op1 or condition evaluates to true then the control goes to op2, op2 gets evaluated and will be returned as output.
- If op1 or condition evaluates to false then the control goes to op3, op3 gets evaluated and will be returned as output.
- The result / output type of conditional operator depends on the type of value present in op2 and op3.

**Note:-**

We cannot execute (op2 and op3) at the same time.

E.g.

```
class PositiveNegative
{
    public static void main(String[] args)
    {
            int   num   =   51  ;

            String ans = num>0 ? num+"is positive" : num+"is negative" ;

            System.out.println(ans);
    }
}

class LargestOfTwo
{
    public static void main(String[] args)
    {
            int    n1  =  10  ;

            int    n2  =   20 ;

            String ans = n1>n2 ? n1+"is greater" : n2+"is greater" ;

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

```
class EvenOdd
{
    public static void main(String[] args)
    {
            int   num  =  234 ;

            String  ans  =  (num%2)==0 ? num+"is even" : num+"is odd" ;

            System.out.println(ans);
    }
}

class LargestOfThree
{
    public static void main(String[] args)
    {
            int   num1  =  10 ;

            int   num1  =  20 ;

            int   num1  =  30 ;

            int   ans  =  n1>n2?n1:n2;

            String  ans2  = ans>n3?ans+"is greater" : n3+"is greater" ;

            System.out.println(ans2);
    }
}

class LargestOfThree
{
    public static void main(String[] args)
    {
            int   num1  =  10 ;

            int   num1  =  20 ;

            int   num1  =  30 ;

            int   largest  =  n1>n2?(n1>n3?n1:n3):(n2>n3?n2:n3);

            System.out.println(largest+"is greater");
    }
}
```

Another ways to perform LargestOfThree

```
(n1>(n2>n3?n2:n3))?n1:(n2>n3?n2:n3) ;

(n1>n2?n1:n2)>n3?(n1>n2?n1:n2):n3;

int large;
int ans  =  (large=n1>n2?n1:n2)>n3?large:n3;
```

Largest of four:-

```
class LargestOfFour
{
    public static void main(String[] args)
    {
            int n1 = 10 ;
            int n2 = 20 ;
            int n3 = 30 ;
            int n4 = 40 ;
            int large = n1>n2?n1:n2;
            int larger = large>n3?large:n3;
            int largest = larger>n4?larger:n4;
            System.out.println(largest+“is largest number.”);
    }
}
```

Another way

```
int large ;
(n1>(large=n2>n3?n2:n3))?n1:(n4>large?n4:large);
```

Another way

```
(n1>(n2>n3?n2:n3))?n1:(n4>(n2>n3?n2:n3)?n4:(n2>n3?n2:n3));
```

# vi. Logical Operators :-

- Logical operators are used for logical decision making purpose.
- The input type of a logical operator must be Boolean.
- The output type of a logical operator will be Boolean.
- There are three logical operators,
  i. AND Operator (&&)
  ii. OR Operator (||)
  iii. NOT Operator (!)

### i. AND Operator (&&):-

- AND operator is a binary operator.
- Truth Table:-

| Operand 1 | Operand 2 | Output |
|---|---|---|
| true | false | false |
| false | true | false |
| false | false | false |
| true | true | true |

- AND operator returns false if any one of the operand is false.

### ii. OR Operator (||):-

- OR operator is a binary operator.
- Truth Table:-

| Operand 1 | Operand 2 | Output |
|---|---|---|
| true | false | true |
| false | true | true |
| false | false | false |
| true | true | true |

- OR operator returns true if any one of the operand is true.

### iii. NOT Operator (!):-

- NOT operator is a unary operator.
- Truth Table:-

| Operand 1 | Output |
|---|---|
| true | false |
| false | true |

- NOT Operator returns true if operand is false.

Program for logical operators:-

```
class Logical
{
	public static void main(String[] args)
	{
		System.out.println(true&&false);		//false
		System.out.println(true||false);		//true
		System.out.println(!true);			//false
		System.out.println(true&&0=0.0);		//true
		System.out.println(10%2==0&&10%2!=1);		//true
		System.out.println('a'<'1' && true);		//true
		System.out.println(0=='0'||0!='0');		//true
		System.out.println((!true) && (!false));	//false
		System.out.println(1.0==1.0f && 3.14==3.14f);	//false
		System.out.println(1.0==1.0f && 3.14==3.14f);	//true
		System.out.println(true && !(true&&false));	//true
	}
}
```

# vii. Increment / Decrement Operator:-

- They are unary operator.

1. **Increment Operator:-**
   They are of two types,
   i. Pre increment operator
   ii. Post increment operator

**i. Pre increment operator:-**

- If a variable is prefixed with increment operator it is known as pre increment operator.
- Syntax: - ++ variable;
- Working: -
  Here the variable is added with one value and then the updated value is substituted.

**ii. Post increment operator:-**

- If a variable is suffixed with increment operator is known as post increment operator.
- Syntax: - variable ++;
- Working: -
  Here the variable is substituted first and then immediately updated with an increase of one value.

**Note:-**

Increment and decrement operators can only be used with the variable not with value or expression.

**Program:-**

```
class Increment
{
    public static void main(String[] main)
    {
        int pocket = 100;
        System.out.println(++pocket);          //101
        System.out.println(++pocket);          //102
        System.out.println(pocket++);          //102
        System.out.println(pocket);            //103
        System.out.println(++pocket);          //104
        System.out.println(pocket++);          //104
        System.out.println(++pocket + pocket++);    //212
    }
}
```

2. **Decrement Operator:-**
   They are of two types
   i. Pre decrement operator
   ii. Post decrement operator

**i. Pre decrement operator :-**

- If a variable is prefixed with decrement operator it is known as pre decrement operator.
- Syntax: - - - variable;
- Working: -
  Here the variable is subtracted with one value and then the updated value is substituted.

**ii. Post decrement operator :-**

- If a variable is suffixed with decrement operator is known as post decrement operator.
- Syntax: - variable - -;
- Working:-
  Here the value is substituted first and then immediately updated with an decrease of one value.

# Operator Precedence in Java :-

Java has well-defined rules for evaluating expressions, including *operator precedence*, *operator associativity*, and *order of operand evalution*. We describe each of these three rules.

## Operator precedence :-

Operator precedence specifies the manner in which operands are grouped with operators. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 because the multiplication operator * has a higher precedence than the addition operator +. You can use parentheses to override the default operator precedence rules.

## Operator associativity :-

When an expression has two operators with the same precedence, the operators and operands are grouped according to their *associativity*. For example 72 / 2 / 3 is treated as (72 / 2) / 3 since the division operator is left-to-right associate. You can use parentheses to override the default operator associativity rules.

Most Java operators are left-to-right associative. One notable exception is the assignment operator, which is right-to-left associative. As a result, the expression x = y = z = 17 is treated as x = (y = (z = 17)), leaving all three variables with the value 17. Recall that an assignment statement evaluates to the value of its right-hand side. Associativity it not relevant for some operators. For example, x <= y <= z and x++-- are invalid expressions in Java.

## Precedence and associativity of Java operators :-

The table below shows all Java 11 operators from highest to lowest precedence, along with their associativity. The table also includes other Java constructs (such as new, [], and ::) that are not Java operators. Most programmers do not memorize them all, and, even those that do, use parentheses for clarity.

| Level | Operator | Description | Associativity |
|---|---|---|---|
| 16 | ( )<br>[ ]<br>new<br>.<br>:: | parentheses<br>array access<br>object creation<br>member access<br>method reference | left – to - right |
| 15 | + +<br>- - | unary post increment<br>unary post decrement | left – to – right |
| 14 | +<br>-<br>!<br>~<br>+ +<br>- - | unary plus<br>unary minus<br>unary logical NOT<br>unary bitwise NOT<br>unary pre increment<br>unary pre decrement | right – to – left |
| 13 | ( ) | cast | right – to – left |
| 12 | * / % | multiplicative | left – to – right |
| 11 | + -<br>+ | additive<br>concatenation | left – to – right<br>left – to – right |
| 10 | << >><br>>>> | shift | left – to – right |
| 9 | < <=<br>> >= | relational | left – to – right |
| 8 | = =<br>! = | equality | left – to – right |
| 7 | & | bitwise AND | left – to – right |
| 6 | ^ | bitwise XOR | left – to – right |
| 5 | \| | bitwise OR | left – to – right |
| 4 | && | logical AND | left – to – right |
| 3 | \|\| | logical OR | left – to – right |
| 2 | ? : | ternary | right – to – left |
| 1 | = += -=<br>*= /= %=<br>&= ^= \|=<br><<= >>= >>>= | assignment | right – to – left |
| 0 | -><br>-> | lambda expression<br>switch expression | right – to – left |

**Increment / Decrement Operator questions for assignment**

1.
```
int a=5,b=9,c=7,d;
d = c++ - a++ + ++b
c = a+b – d--;
a = ++a + --b;
b= --d – c--;
```

2.
```
int a=5,b=7,c=3,d;
d = b++ + ++c – a++;
a = --d + b++;
b = a+b - ++c;
c = ++d – c++ + --b;
```

3.
```
int a = -3,b=8,c=a+b,d;
d = -c++ + a--;
a = a++;
b = +a++ - --c;
c = c--;
```

# ★ Taking input from the user:-

- Using core java we can create standalone application i.e. console based application.
- There are different ways to take input from the user,
  Scanner class,
  Console class,
  Command line argument,
  To fetch the data from the front end we use JSP (Java Server Pages) / Servlets

## ★ Scanner class:-

- Reading input from the console enables the program to accept input from the user.
- To read data dynamically from the user we use a built-in class called as '**Scanner'**.
- Scanner class used to read data dynamically during execution of a program.
- Scanner is built-in class present in java.util package.
- Java uses System.out to refer to the standard output device and System.in to the standard input device.
- By default, the output device is the display monitor and the input device is the keyboard.
- To perform console output, you simply use the println method to display a primitive value or a String to the console.
- Console input is not directly supported in Java, but you can use the Scanner class to create an object to read input form System.in

**Scanner input = new Scanner(System.in);**

- The syntax new Scanner(System.in) creates an object of the Scanner type.
- The syntax Scanner input declares that **input** is a variable whose type is **Scanner**.
- The whole line **Scanner input = new Scanner(System.in)** creates a Scanner object and assigns its reference to the variable input.
- An object may invoke its methods.
- To invoke a method on an object is to ask the object to perform a task.

### Steps to use scanner class :-

**Step 1** Import Scanner class.
We can input class with the help of keyword called import java.util.Scanner;
Import statement must be the first statement of a java program.

**Step 2** Create the object of Scanner class with the help of scanner constructor.
Scanner input = new Scanner(System.in);
The Scanner object gives a scanner object reference which should be stored inside a scanner type variable(input).

**Step 3** With the help of scanner object reference use the method of scanner class to read data dynamically from the user.

### Methods for Scanner Objects:-

| Data-type | Method Name | Return Type |
|---|---|---|
| byte | nextByte(); | byte |
| short | nextShort(); | short |
| int | nextInt(); | int |
| long | nextLong(); | long |
| float | nextFloat(); | float |
| char | next().charAt(0); | char |
| boolean | nextBoolean(); | boolean |
| String | next(); | String |
| | nextLine(); | |

WAJP for all the data type input from the user.

```
import   java.util.Scanner;
class   ScannerExample
{
        public static void main(String[] agrs)
        {
                //create an object of scanner class
                //and store it in reference variable
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter a byte : ");
                byte   b   =   sc.nextByte();
                System.out.println("Byte b = "+b);
                System.out.println("Enter a short : ");
                short   s   =   sc.nextShort();
                System.out.println("Short s = "+s);
                System.out.println("Enter a integer : ");
                int   i    =   sc.nextInt();
                System.out.println("Integer i = "+i);
                System.out.println("Enter a long : ");
                long l   =   sc.nextLong();
                System.out.println("Long l = "+l);
                System.out.println("Enter a float : ");
                float f = sc.nextFloat();
                System.out.println("Float f = "+f);
                System.out.println("Enter a double : ");
                double d = sc.nextDouble();
                System.out.println("Double d = "+d);
                System.out.println("Enter a boolean : ");
                boolean g = sc.nextBoolean();
                System.out.println("Boolean b = "+g);
        }
}
```

WAJP string input using next() and nextLine().

```
import java.util.Scanner;
class ScannerExample2
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter a string : ");
                String str1 = sc.nextLine();
                System.out.println("Enter a Second String : ");
                String str2 = sc.next();
                System.out.println("String : "+str1);
                System.out.println("String : "+str2);
        }
}
```

WAJP to take name from user and check its first character is vowel or consonant.

```
import java.util.Scanner;
class ScannerExample3
{
	Public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println(“Enter a name : ”);
		char ch = next().ToLowerCase.charAt(0);
		System.out.println((ch==’a’ || ch==’i’ || ch==’o’ || ch==’e’ || ch==’u’)?ch+” is a vowel”:ch+” is consonant”);
	}
}
```

WAJP to check if user entered character is alphabet or not.

```
import java.util.Scanner;
class ScannerExample3
{
	Public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println(“Enter a character : ”);
		char ch = next().charAt(0);
		System.out.println((ch>=65&&ch<=90 || ch>=97&&ch<=122)?ch+” is a alphabet”:ch+” is not an alphabet”);
	}
}
```

WAJP to check if entered character is lowercase or uppercase or digit

```
import java.util.Scanner;
class ScannerExample3
{
	Public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println(“Enter a character : ”);
		char ch = sc.next().charAt(0);
		System.out.println((ch>=97&&ch<=122)?ch+” is lowercase.”:(ch>=65&&ch<=90)?ch+” is uppercase.”:(ch>=48&&ch<=57)?ch+” is digit”: ch+ “is not digit, no lowercase, no uppercase);
	}
}
```

WAJP to find whether the number is even or odd without using modulus operator

```
class EvenOddExample
{
	public static void main(String[] args)
	{
		int num = 7;
		String str = (num/2==0&&num/2.0==0?”It is even.”:”It is odd.”);
		System.out.println(str);
	}
}
```

WAJP to swap two numbers using a third variable

```
class SwapTwoNumbers
{
	public static void main(String[] str)
	{
		int a = 10;
		int b = 20;
		int temp;
		System.out.println(“Before Swapping”);
		System.out.println(“a = ”+a);
		System.out.println(“b = ”+b);
		temp = a;
		a =b;
		b = temp;
		System.out.println(“After Swapping”);
		System.out.println(“a = ”+a);
		System.out.println(“b = ”+b);
	}
}
```

WAJP to swap two numbers without using a third variable

```
class SwapTwoNumbers
{
	public static void main(String[] args)
	{
		int a =10;
		int b = 20;
		System.out.println(“Before Swapping”);
		System.out.println(“a = ”+a);
		System.out.println(“b = ”+b);
		a = a+b;
		b = a-b;
		a = a-b;
		System.out.println(“After Swapping”);
		System.out.println(“a = ”+a);
		System.out.println(“b = ”+b);

	}
}
```

# ★ CONTROL FLOW STATEMENTS :-

- The statement which helps the programmer to control the execution flow of a java program is known as control flow statement.
- There are three types of control flow statements,
  1. Decision Making Statements
  2. Loop Statements
  3. Branching Statements

## 1. Decision Making Statements:-

- The statement which helps the programmer in making decision whether to execute a block or not is known as decision making statement.
- There are different types of decision making statements,
  i. Simple if statement
  ii. if else statement
  iii. if else if (ladder) statement
  iv. switch statement

### i. if statement:-

- if is a conditional statement which helps the programmer to make decision whether to execute a block or not.
- Syntax:-


**Control flow of if :-**

- The statements inside the if block gets executed only if the condition evaluates to true.
- If the condition is false then if block is skipped or ignored.

E.g.

```
class IfExample1
{
    public static void main(String[] agrs)
    {
            System.out.println("Execution Starts");
            if(10>=10)
            {
                    System.out.println("Hello from if");
            }
            System.out.println("Execution Ends");
    }
}
```

O/p:-

```
Execution Starts
Hello from if
Execution Ends
```

```
class IfExample1
{
    public static void main(String[] agrs)
    {
            System.out.println("Execution Starts");
            if(!true)
            {
                    System.out.println("Hello from if");
            }
            System.out.println("Execution Ends");
    }
}
```

O/p:-

```
Execution Starts
Execution Ends
```

**Note:-**

- In control flow statements blocks ({}) are optional except **switch** statement.
- In if statement, if we omit the braces only one single statement is considered as part of if.

```
class IfExample1
{
    public static void main(String[] agrs)
    {
            System.out.println("Execution Starts");
            if(false)
                    System.out.println("Hello");
                    System.out.println("Hello from if");
            System.out.println("Execution Ends");
    }
}
```

O/p:-

```
Execution Starts
Hello from if
Execution Ends
```

- If we are omitting the braces that one statement cannot be a variable declaration because if we declare a variable inside if, we cannot utilize that variable that's why it creates a compile time error.

E.g.

```
class IfExample1
{
    public static void main(String[] agrs)
    {
            System.out.println("Execution Starts");
            if(true)
            int a = 10;       //CTE
            System.out.println("Execution Ends");
    }
}
```

O/p:-
Compile Time Error (variable declaration is not allowed here)

**Programs**

**1.**

```
class IfExample1
{
	public static void main(String[] agrs)
	{
		System.out.println("Execution Starts");
		int a = 10;
		if(true)
		{
			int b = 20;
			System.out.println(a);
			System.out.println(b);
		}

		System.out.println(a);
		System.out.println(b); //CTE cannot find symbol
		System.out.println("Execution Ends");
	}
}
```

**2.**

```
class IfExample1
{
	public static void main(String[] agrs)
	{
		System.out.println("Execution Starts");
		int a;
		if(true)
		{
			a = 20;
			System.out.println("if");
		}

		System.out.println(a);
		System.out.println("Execution Ends");
	}
}
```

**3.**

```
class IfExample1
{
	public static void main(String[] agrs)
	{
		System.out.println("Execution Starts");
		int a ;
		if(false)
		{
			a = 20;
			System.out.println("if");
		}
		System.out.println(a);
		System.out.println("Execution Ends");
	}
}
```

**4. WAJP to check if the number is multiple of 5 if yes display 'hi five', if the number is divisible by 2 display 'hi two', if the number is divisible by 5 and 2 display 'hi, five two'.**

```
import java.util.Scanner;
class IfExample1
{
	public static void main(String[] agrs)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Execution Starts");
		int a;
		System.out.println("Enter a number");
		a = sc.nextInt();
		if(a%5==0&&a%2!=0)
			System.out.println("Hi five");

		if(a%2==0&&a%5!=0)
			System.out.println("Hi two");

		if(a%5==0&&a%2==0)
			System.out.println("Hi five, two");

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

**Another way to write this program:-**

```
import java.util.Scanner;
class IfExample1
{
	public static void main(String[] agrs)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Execution Starts");
		int a;
		System.out.println("Enter a number");
		a = sc.nextInt();
		if(a%5==0&&a%2!=0)
		{
			System.out.println("Hi five");
			return;
		}
		if(a%2==0&&a%5!=0)
		{
			System.out.println("Hi two");
			return;
		}
		if(a%5==0&&a%2==0)
		{
			System.out.println("Hi five, two");
			return;
		}
		System.out.println("Execution Ends");
	}
}
```

WAJP an if statement that assigns 1 to x if y is greater than 0

```
class IfExample
{
	public static void main(String[] agrs)
	{
		int x = 1;
		int y = 10;
		if(y>0)
		x=1;
		System.out.println(x);
	}
}
```

WAJP an if statement that increases pay by 3% if score is greater than 90

```
class IfExample
{
	public static void main(String[] agrs)
	{
		int score = 91;
		float pay = 5000;
		if(score>90)
		pay = pay+((pay/100)*3);
		System.out.println(pay);
	}
}
```

WAJP to check if the user entered percentage is eligible for appearing the interview or not?

```
import java.util.Scanner;
class IfExample
{
	public static void main(String[] agrs)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println(“Execution starts.”);
		float perc = sc.nextFloat(System.in);
		if(perc>60)
			System.out.println(“Eligible for interview.”);
		if(perc<60)
			System.out.println(“Not eligible for interview.”);
		System.out.println(“Execution Ends.”);
	}
}
```

WAJP to check if the user entered number is even or odd

```
import java.util.Scanner;
class IfExample
{
	public static void main(String[] agrs)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println(“Enter a number : ”);
		int a = nextInt();
		if(a%2==0)
			System.out.println(“Even”);
		if(a%2!=0)
			System.out.println(“Odd”);
	}
}
```

WAJP to check if the user given number is positive number

```
import java.util.Scanner;
class IfExample
{
        public static void main(String[] agrs)
        {
                Scanner sc = new Scanner(System.in);
                int a = sc.nextInt();
                if(a>0)
                        System.out.println("Positive Number");
                if(a<0)
                        System.out.println("Negative number");
        }
}
```

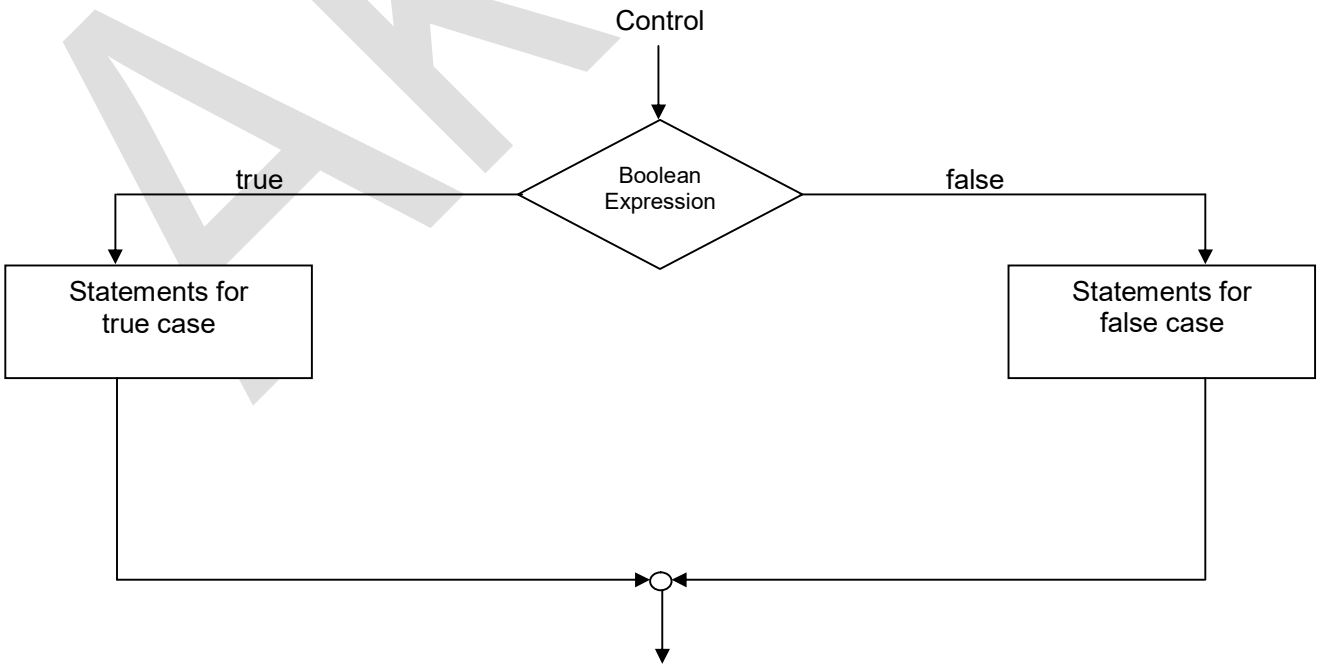
### ii. If else statement:-

- if else statement is a conditional statement which helps the programmer in making decision whether to execute a block or not.

- Syntax:-

```
            keyword
               ↓          ↘ no boolean it throws compile time error
            if (condition)
            {          ↘ boolean (true / false)
true  ──┐
        │           //statements
        └──
            }
            else
            {
false ──┐
        │           //statements
        └──
            }
```

- Flowchart of if else statement:-



## Control flow of if else:-

- The statement inside the if block gets executed only if the condition is true.
- If the condition evaluates to false then if block is skipped and else block gets executed.
- It's not possible in any case that both the blocks get executed at the same time.
- In this statement also blocks are optional and if we omit the braces only one statement is considered as a part of if and else.

e.g.

```
class IfElseExample
{
  public static void main(String[] agrs)
  {
        if(true)
      {
            System.out.println("If block");
        }
        //comment
        else
        {
            System.out.println("Else block");
        }
  }
}
```

o/p:-

```
If block
```

**Note:-**

- If you try to insert any statement between if and else it creates compile time error i.e. else without if.
- But we can only add comments between them as they are part of java.

```
class IfElseExample
{
  public static void main(String[] agrs)
  {
        System.out.println("Execution Starts");
        boolean a = false;
      if(!a)
            System.out.println("If block");
            System.out.print("Hello from if again.");
        //comment
        else
            System.out.println("Else block");
        System.out.println("Execution Starts");
      }
}
```

o/p:-

```
compile time error
```

### iii. if else if (ladder):-

- if else if ladder is a conditional statement which helps a programmer in making decision whether to execute a block or not.
- We use ladder when we want to check for multiple conditions.
- Syntax:-

```
if(condition 1)
{
        //statement if execute condition 1 is true
}
else if(condition 2)
{
        //statement if execute condition 2 is true
}
else if(condition 3)
{
        //statement if execute condition 3 is true
}
else
{
        //statement execute if all the conditions false
}
```

### Control flow of if else if ladder:-

- In if else if ladder we check for multiple conditions.
- Conditions are checked from top to bottom order.
- Whichever first condition evaluates to true that corresponding if block gets executes and rest of the blocks are skipped.
- If all the specified condition evaluates to false then only the else block gets executed.
- Specifying an else block is optional.

e.g.

```
class Ladder
{
  public static void main(String[] args)
  {
        if(10>10.0001)
        {
                System.out.println("Hello from if block 1");
        }
        else if('a'==99)
        {
                System.out.println("Hello from if block 2");
        }
        else if('a'>('b'-1))
        {
                System.out.println("Hello from if block 3");
        }
        else
        {
                System.out.println("Hello from else");
        }
  }
}
```

o/p:-

```
Hello form else
```

### iv. Switch statement:-

- Switch is a decision making statement which helps the programmer in making decision whether to execute a block or not.
- Syntax:-

```
switch( variable / expression / literal )
{
    case  label1 ( variable / expression / literal ) :
    {
            //statement
            //break;
    }
            case  label2 ( variable / expression / literal ) :
    {
            //statement
            //break;
    }
            case  label3 ( variable / expression / literal ) :
    {
            //statement
            //break;
    }
    default :
    {
            //statement
    }
}
```

- e.g.

```
class SwitchExample
{
    public static void main(String[] args)
    {
            int a = 3;
            switch(a)
            {
                    case 1:
                    {
                            System.out.println(“hello from case1”);
                            break;
                    }
                    case 2:
                    {
                            System.out.println(“hello form case 2”);
                            break;
                    }
                    case 3:
                    {
                            System.out.println(“hello form case 3”);
                            break;
                    }
                    default:
                    {
                            System.out.println(“hello form default block”);
                    }
            }
    }
}
```

## Project :- Simple Hotel Bill Management

```
import java.util.Scanner;
class JavaKaDhaba
{
	public static void main (String[] args)
	{
		Scanner sc = new Scanner(System.in);
		String foodItems = "";
		double totalBill = 0 ;
		System.out.println("                    *** MELOCME ****");
		System.out.println("                    JAVA KA DHABA ");
		System.out.println("");
		System.out.println("");
		// INFINTE LOOP FOR MAIN MENU
		for ( ; ; )
		{
			System.out.println();
			System.out.println("MENU 1. VEG 2. NON-VEG 3. Checkout 4. LOGOUT");
			System.out.print("Enter your option : ");
			int option = sc.nextInt();
			System. out.println();
			// this else if block is for veg menu
			if(option==1)
			{
				// INFINE Loop for veg menu Loop will break if user enters 5
				for( ; ; )
				{
					System.out.println("*** VEG MENU *** ");
					System.out.println("1.PANEER TIKKA 2.KAJU CARRY
							3.VEG- BIRYANI 4. VEG-KOFTA 5.MAIN MENU");
					System.out.println();
					System.out.println("Enter yout option : ");
					int option1 = sc.nextInt();
					if(option1==1)
					{
						foodItems += "PANEER TIKKA : 250";
						totalBill += 250;
						System.out.println("your order has been added to cart.");
					}
					else if(option1==2)
					{
						foodItems += "KAJU CURRY : 280";
						totalBill += 280;
						System.out.println("your order has been added to cart.");
					}
					else if(option1==3)
					{
						foodItems += "VEG-BIRYANI : 180";
						totalBill += 180;
						System.out.println("your order has been added to cart.");
					}
					else if(option1==4)
					{
						foodItems += "VEG-KOFTA : 210";
						totalBill += 210;
						System.out.println("your order has been added to cart.");
					}
					else if(option1==5)
					{
```

```
                        break;
                    }
                }
            }
            else if(option==2)
            {
                System.out.println("*** NON VEG MENU *** ");
                // INFINE Loop for non veg menu Loop will break if user enters 5
                for( ; ; )
                {
                    System.out.println("1.BUTTER CHICKEN 2.CHICKEN
                            MASALA 3.CHICKEN-BIRYANI 4.FISH 5.MAIN MENU");
                    System.out.println();
                    System.out.println("Enter yout option : ");
                    int option1 = sc.nextInt();
                    if(option==1)
                    {
                        foodItems += "BUTTER CHICKEN : 300";
                        totalBill += 300;
                        System.out.println("your order has been added to cart.");
                    }
                    else if(option1==2)
                    {
                        foodItems += "CHICKEN MASALA : 270";
                        totalBill += 270;
                        System.out.println("your order has been added to cart.");
                    }
                    else if(option1==3)
                    {
                        foodItems += "CHICKEN-BIRYANI : 200";
                        totalBill += 200;
                        System.out.println("your order has been added to cart.");
                    }
                    else if(option1==4)
                    {
                        foodItems += "FISH : 400";
                        totalBill += 400;
                        System.out.println("your order has been added to cart.");
                    }
                    else if(option1==5)
                    {
                        break;
                    }
                }
            }
            else if(option==3)
            {
                System.out.println("*** YOUR TOTAL BILL ***");
                System.out.println();
                System.out.println("Your purchase items below: ");
                System.out.println(foodItems);
                System.out.println("Total bill : "+totalBill+" rs");
                if(totalBill>=2000)
                {
                    System.out.println("You got as discount of 20%");
                    System.out.println("Your bill is : "+(totalBill-((totalBill/100)*20)));
                }
                else if(totalBill>=1000)
                {
                    System.out.println("You got as discount of 10%");
```

```
                                        System.out.println("Your bill is : "+(totalBill-((totalBill/100)*10)));
                                }
                                else if(totalBill>=500)
                                {
                                        System.out.println("You got as discount of 5%");
                                        System.out.println("Your bill is : "+(totalBill-((totalBill/100)*5)));
                                }
                        }
                        else if(option==4)
                        {
                                System.out.println("Thank you Visit again");
                                break;
                        }
                        System.out.println();
                }
        }
}
```

### Characteristics of switch:-

- We can pass byte, short, int, char and String (non-primitive datatype) type of data in switch selector.
- We cannot pass boolean type as a switch selector because boolean can have only two possible values switch is used for multiple conditions.
- We cant pass long, float and double as they have a bigger range that much number of case labels are not required to be created.
- Inside switch selector we can pass a variable / expression / value.
- We can create multiple case labels and blocks for them are optional.
- But we cannot create duplicate case label as well as default.
- default block in switch is optional.

e.g. 1

```
class SwitchExample
{
  public static void main(String[] args)
  {
        byte a = 120;
        switch(a)
        {
                case 1: System.out.println(“Hello from case 1”);break;
                case 2: System.out.println(“Hello form case 2”);break;
                case 127: System.out.println(“Hello form case 127”);break;
                case 128: System.out.println(“Hello form case 128”);break;
                default: System.out.println(“Hello from default.”);break;
        }
  }
}
```

O/p:-
Compile time error

**Number of case labels in switch depends upon the type of selector.**
In the above example selector type is byte and we have created a case label as 128 so it creates a compile time error.

e.g.2
```
class SwitchExample
{
	public static void main(String[] args)
	{
		byte a = 120;
		Switch(a+0)
		{
			case 1 : System.out.println(“Hello from case 1”);break;
			case 2 : System.out.println(“Hello from case 2”);break;
			case 120 : System.out.println(“Hello from case 120”);break;
			case 128 : System.out.println(“Hello from case 128”);break;
			default : System.out.println(“Hello from default”);break;
		}
	}
}
```

o/p:-
Hello from case 120
In the above example the switch selector has an expression in which we are adding a byte data with ‘0’ (zero) (datatype of zero is int) so byte+int the resultant output will be in form of int so that’s why the selector type is int.

e.g.3
```
class SwitchExample
{
	public static void main(String[] args)
	{
		char a = 1;
		Switch(a)
		{
			case 1 : System.out.println(“Hello from case 1”);break;
			case ‘a’ : System.out.println(“Hello from case a”);break;
			case 97 : System.out.println(“Hello from case 97”);break;
			case 98 : System.out.println(“Hello from case 98”);break;
		}
	}
}
```

o/p:-
compile time error : duplicate case label
For faster execution in switch java allow us to create unique case labels but, in the above which creates ambiguity (confusion) which case block must be executed.

e.g.4
```
class SwitchExample
{
	public static void main(String[] args)
	{
		boolean b = true;
		Switch(b)
		{
			case 1 : System.out.println(“Hello from case 1”);break;
			case 2 : System.out.println(“Hello from case 2”);break;
		}
	}
}
```
o/p:- Compile time error : boolean type is not allowed as a switch selector.

## default block:-

- default is a keyword.
- default block in switch is executes at the end when none of the case label matches with the selector.
- default block is optional.
- We can declare a default block anywhere inside a switch i.e. at the start, end or somewhere in middle of cases.

e.g.

```
class Example
{
        public static void main(String[] main)
        {
                int a = 3;
                Switch(a)
                {
                        case 1 : System.out.println(“case 1”);break;
                        default : System.out.println(“default”);break;
                        case 2 : System.out.println(“case 2”);break;
                        case 3 : System.out.println(“case 3”);break;
                }
        }
}
```

o/p:- case 3

e.g.

```
class Example
{
        public static void main(String[] main)
        {
                int a = 5;
                Switch(a)
                {
                        default : System.out.println(“default”);break;
                        case 1 : System.out.println(“case 1”);break;
                        case 2 : System.out.println(“case 2”);break;
                        case 3 : System.out.println(“case 3”);break;
                }
        }
}
```

o/p:- default

**Note:-**

- We can not create more than one default block in a Switch it creates duplicate default i.e. compile time error.

# 2. Branching Statement:-

- The branching statements alter the flow of execution in a program.
- The statement allow us to jump to different part of code.
- That's why they are also called as control transfer statements.
- There are several control transfer statements in java such as,
  break;
  continue;
  return;
  goto; //deprecated (cannot use)

## 1. break:-

- break is a keyword and control transfer statement in java.
- We can use break in switch and loop (Other than this we cannot use it anywhere).
- break in loop is used to terminate the execution.
- break in switch is used with cases to terminated the execution of switch.
- e.g.

```
class Vowel
{
    public static void main(String[] args)
    {
        System.out.println("Execution starts");
        System.out.print("Enter a character : ");
        char ch = sc.next().charAt(0);
        Switch(ch)
        {
            case 'a' : System.out.println(ch+"it is vowel.");break;
            case 'e' : System.out.println(ch+"it is vowel.");break;
            case 'i' : System.out.println(ch+"it is vowel.");break;
            case 'o' : System.out.println(ch+"it is vowel.");break;
            case 'u' : System.out.println(ch+"it is vowel.");break;
            default : System.out.println(ch+"it is consonant.");break;
        }
        System.out.println("Execution ends");
    }
}
```

### Fall through condition in switch:-

- This condition occurs in switch statement when there is no break statement at the end of case block.
- If any of the case labels gets matched with the selector it continues the execution into the subsequent case labels until there is a break statement or end of Switch.

**Note:-**

Fall through condition has its own advantages and disadvantages and it totally depends on type of operation we want to perform in our program.

**Program:-**

```
class Vowel
{
    public static void main(String[] args)
    {
        System.out.println(“Execution starts”);
        System.out.print(“Enter a character : “);
        char ch = sc.next().charAt(0);
        Switch(ch)
        {
            case ‘a’ :
            case ‘e’ :
            case ‘i’ :
            case ‘o’ :
            case ‘u’ : System.out.println(ch+”it is vowel.”);break;
            default : System.out.println(ch+”it is consonant.”);break;
        }
        System.out.println(“Execution ends”);
    }
}
```

**Program:-**

```
class Example
{
    public static void main(String[] args)
    {
        System.out.println(“Execution starts”);
        int i = 5;
        Switch(i)
        {
            default : System.out.println(“default”);
            case 1 : System.out.println(“Case 1”);
            case 2 : System.out.println(“Case 2”);
            case 3 : System.out.println(“Case 3”);
            case 4 : System.out.println(“Case 4”);
        }
        System.out.println(“Execution ends”);
    }
}
```

Output:-

```
Execution starts
default
Case 1
Case 2
Case 3
Case 4
Execution ends
```

**Program:-**

```
class Example
{
	public static void main(String[] args)
	{
		System.out.println(“Execution starts”);
		int i = 5;
		Switch(i)
		{
			case 1 : System.out.println(“Case 1”);
			case 2 : System.out.println(“Case 2”);
			case 3 : System.out.println(“Case 3”);
			case 4 : System.out.println(“Case 4”);
			default : System.out.println(“default”);
		}
		System.out.println(“Execution ends”);
	}
}
```

Output:-

```
Execution starts
default
Execution ends
```

**VotingSystem.java**

```
import java.util.Scanner;
class VotingSystem
{
	static int abc,xyz,mno,pqr,ghi,nota;
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in)	;
		System.out.println("Enter the population : ");
		int pop = sc.nextInt();
		for(int i=1;i<=pop;i++)
		{
			System.out.println();
			System.out.println("***WELCOME***");
			System.out.println();
			System.out.println();
			System.out.println("1.ABC 2.XYZ 3.MNO 4.PQR 5.GHI 6.NOTA");
			System.out.println();
			System.out.println("Enter an option : ");
			int option = sc.nextInt();
			switch(option)
			{
				case 1 :System.out.println("you voted for ABC ");abc++;break;
				case 2 :System.out.println("you voted for XYZ ");xyz++;break;
				case 3 :System.out.println("you voted for MNO ");mno++;break;
				case 4 :System.out.println("you voted for PQR ");pqr++;break;
				case 5 :System.out.println("you voted for GHI ");ghi++;break;
				case 6 :System.out.println("You are educated.");nota++;break;
				default :System.out.println("Invalid Option. Vote again.");i--;break;
			}
		}
		int count1 = (mno>(abc>xyz?abc:xyz))?mno: (abc>xyz?abc:xyz);
		int count2 = (nota>(pqr>ghi?pqr:ghi))?nota: (pqr>ghi?pqr:ghi);
		int finalCount = count1>count2?count1:count2;
		if(abc==finalCount){
			System.out.println("Winner is ABC with "+abc+" votes.");
		}
		else if(xyz==finalCount){
			System.out.println("Winner is XYZ with "+xyz+" votes.");
		}
		else if(mno==finalCount){
			System.out.println("Winner is MNO with "+mno+" votes.");
		}
		else if(pqr==finalCount){
			System.out.println("Winner is PQR with "+pqr+" votes.");
		}
		else if(ghi==finalCount){
			System.out.println("Winner is GHI with "+ghi+" votes.");
		}
		else if(nota==finalCount){
			System.out.println("No winner. Nota = "+nota+" votes.");
		}
	}
}
```

★ **Difference between if else if ladder and Switch**

| if else if ladder | Switch |
|---|---|
| i. Use when you need to evaluate complex conditions. | i. Best used when comparing a single variable against a list of possible constant values. |
| ii. Works with all datatypes. | ii. Only works with int, char, byte, short, String, enum. |
| iii. Evaluates ‘ if ‘ or ‘ else if ’ condition sequentially from top to bottom order. | iii. Directly jumps to the matching ‘ case ‘. |
| iv. Stop evaluate once a ‘ true ‘ condition is found and execute its block. | iv. if ‘ break ‘ is omitted, the code will continue executing the following cases (fall through). |
| v. if else if ladder is slower. | v. Switch is faster. |

# 3. Loop statement :-

- Loop statements are used to execute some set of instructions repeatedly.
- We can iterate (cycle) a loop desired number of time based of our requirement.
- Loops are used to make java program more consized (small) and increases readability of our code.
- There are four types of loops in java,
  1. for loop
  2. while loop
  3. do while loop
  4. for each loop / enhanced loop / advance for loop

- Loops have three parts,
  1. Initialization statement
  2. Condition statement
  3. Updation / Update statement

1. **Initialization statement :-**
   - The initialization statement is the starting point of a loop.
   - Usually this statement consists of variable declaration and initialization.

2. **Condition statement :-**
   - Condition is used to control the loop.
   - If the condition evaluates to true loop body is executed.
   - If the condition evaluates to false loop body is terminated.

3. **Updation / Update statement :-**
   - This statement should be executed in every iteration (cycle) of loop.
   - Update statement helps the condtion to evaluate to false after desired number of iteration.

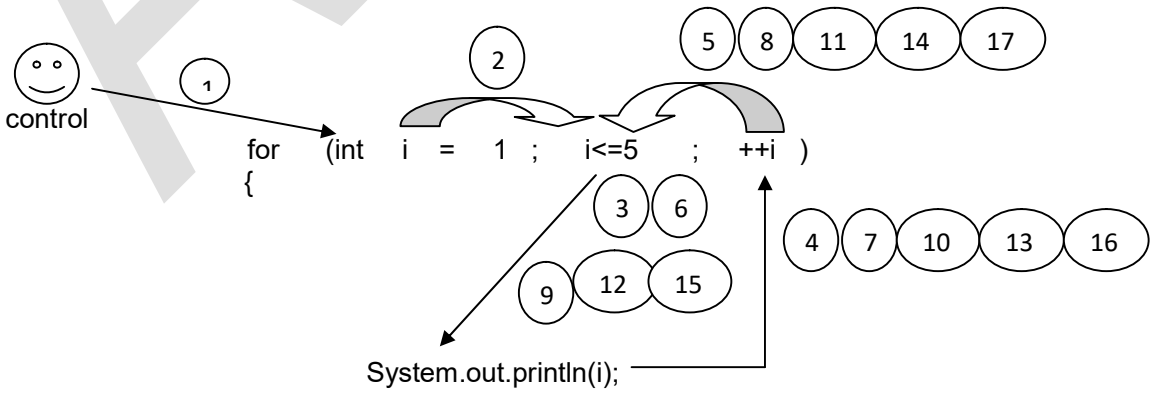
**Note:-**

The above three parts of loop is not applicable to for each loop.

1. **for loop :-**
   - for is loop statement in java which is used to execute some set of instructions repeatedly.
   - Syntax:-

```
for(initialization ; condition ; updation)
{
        //statements (block)
}
```

   - Control flow of for loop (working):-



**Example:-**

```
class ForLoopExample{
	public static void main(String[] agrs){
		//	i=1	1<=5 (true)	2
		//		2<=5 (true)	3
		//		3<=5 (true)	4
		//		4<=5 (true)	5
		//		5<=5 (true)	6
		//		6<=5 (false)	loop gets terminated
		for(int i = 1; i<=5; ++i){
			System.out.println(i);
		//1.  Prints – 1
		//2.  Prints – 2
		//3.  Prints – 3
		//4.  Prints – 4
		//5.  Prints – 5
		}
	}
}
```

**WAJP to print 1 to 20.**

```
class ForLoopExample{
	public static void main(String[] args){
		for(int i = 1; i<=20; i++)
			System.out.println(i);
	}
}
```

**WAJP to print 20 to 1.**

```
class ForLoopExample{
	public static void main(String[] args){
		for(int i = 20; i>=1; i--)
			System.out.print(i+" ");
	}
}
```

**WAJP to print alphabets a to z.**

```
class ForLoopExample{
	public static void main(String[] args){
		for(char i = 'a' ; i<='z'; i++)
			System.out.print(i+" ");
	}
}
```

**WAJP to print uppercase alphabets A to Z.**

```
class ForLoopExample{
	public static void main(String[] args){
		for(char i = 'A'; i<='Z'; i++)
			System.out.print(i+" ");
	}
}
```

**WAJP to print alphabets and their ascii values.**

```
class ForLoopExample
{
	public static void main(String[] args)
	{
		for(char i = ‘A’; i<=’Z’; i++)
			System.out.print(i+” : ”+(i+0)+” “);
	}
}
```

**WAJP to print the given series (A,D,G,J,M,…)**

```
class ForLoopExample
{
	public static void main(String[] args)
	{
		for(int i = ‘A’; i<=’Z’; i+=3)
			System.out.println(i+” “);
	}
}
```

**WAJP to print digits 0 to 9.**

```
class ForLoopExample
{
	public static void main(String[] args)
	{
		for(char i = ‘0’; i<=’9’; i++)
			System.out.print(i+” ”);
	}
}
```

**WAJP to print z to a (lowercase).**

```
class ForLoopExample
{
	public static void main(String[] args)
	{
		for(char i = ‘z’; i>=’a’; i--)
			System.out.print(i+” ”);
	}
}
```

**WAJP to print all even numbers between 1 to 100.**

```
class ForLoopExample
{
	public static void main(String[] args)
	{
		for(int i = 1; i<=100; i++)
		{
			if(i%2==0)
			{
				System.out.print(i+” ”);
			}
		}
	}
}
```

**WAJP to print all odd numbers between 100 to 1.**

```
class ForLoopExample{
        public static void main(String[] args){
                for(int i = 100; i>=1; i--){
                        if(i%2!=0){
                                System.out.print(i+" ");
                        }
                }
        }
}
```

**WAJP to print sum of 1 to 20.**

```
class ForLoopExample
{
        public static void main(String[] args)
        {
                int sum = 0;
                for(char i = 1; i<=20; i++)
                        sum+=I;
                System.out.print("Sum of 1 to 20 : "+sum);
        }
}
```

**Note:-**

- Blocks are optional in loops if we are omitting the blocks only one single statement is considered as a part of loop.
- If we are not specifying a condition for for loop compiler provides with a default condition i.e. true.
- E.g.

```
for(int i = 1;       ; i++)
{
        System.out.print(i+" ");
}
```

**(true provided by compiler)**

Output:-
It creates infinite loop.

- So after an infinite loop the statements become unreachable.
- E.g

```
for(int i = 1; false ; i++)
{
        System.out.print(i+" ");          //unreachable statement error
}
System.out.println(Execution ends");
```

- In the above loop we had specified a constant condition i.e. false.
- Because of which control cannot execute the block and loop gets terminated.
- So compiler throws a compiler time error i.e. unreachable statement.

**If we want to create infinite loop there are multiple ways,**

1. 
```
for(   ;   ;   )
{
  System.out.println(“Hello”);
}
```
2. 
```
for(int i =1 ;i<=5;   )
{
  System.out.println(“Hello”);
}
```
3. 
```
for( int i =1 ; true ; i++ )
{
  System.out.println(“Hello”);
}
```

- In for loop we can initialize the counter variable before starting for a loop or inside the loop.
- e.g.
```
int i = 1;
for(    ;  i <=5  ; i++  )
{
  System.out.println(i);
}
```

- Updation statement can be declared in the for loop body or with the initialization and condition.
- In for loop the initialization, condition and updation is declared inside the same statement.
  e.g.
```
int i = 1;
for(    ; i<10 ;   )
{
  System.out.println(i);
  i++;
}
```

**WAJP to find factors of user entered number.**

```
import java.util.Scanner;
class FactorsOfANumber
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a number : ");
		int num = sc.nextInt();
		System.out.println("The factors of "+num+" is : ");
		for(int i=1;i<=num;i++)
		{
			if(num%i==0)
				System.out.print(i+" ");
		}
	}
}
```

**WAJP to find user entered number is prime or not.**

```
import java.util.Scanner;
class PrimeNumber
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a number : ");
		int num = sc.nextInt();
		boolean flag = true;
		for (int i=2;i<num ;i++ )
		{
			if(num%i==0)
			{
				flag=false;
				break;
			}
		}
		System.out.println((flag)?(num+" is a prime."):(num+ " is not prime."));
	}
}
```

**WAJP to print sum of factors of a user entered number.**

```
import java.util.Scanner;
class SumOfFactors
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a number : ");
		int num = sc.nextInt();
		// int temp = num;
		int sum = 0;
		System.out.println("The factors of "+num+" is : ");
		for(int i=1;i<=num;i++)
			if(num%i==0)
			{
				System.out.print(i+" ");
				sum=sum+i;
			}
		System.out.println();
		System.out.println("The sum of factors of no. "+num+" is : "+sum);
	}
}
```

**WAJP to print '0' to '9' digit, 'A' to 'Z' and 'a' to 'z'**

```
class ForLoopExample
{
	public static void main(String[] args)
	{
		for(char i =1 ; i<=128  ;i++)
		{
			if ( (i>='0'&&i<='9') || (i>='a'&&i<='z') || (i>='A'&&i<='Z') )
				System.out.print(i+" ");
		}
	}
}
```

## 2. while loop:-

- while loop is a control flow statement (looping statement) which helps the programmer to execute some set of instructions repeatedly.
- We use while loop when we don't know the number of iterations.
- Syntax:-

```
Initialization;
while ( condition )
{
        //statements
        Updation;
}
```

- e.g. printing 'a' to 'z'

```
class WhileLoopExample
{
        public static void main(String[] args)
        {
                char i = 'a';
                while(i<='z')
                {
                        System.out.println(i+" ");
                        i++;
                }
        }
}
```

**Note:-**

- In while loop initialization should be done before starting of the loop.
- Updation is declared inside the loop body.
- If we are not specifying any update statement it will create an infinite loop.
- In while loop there is no default condition provided by the compiler if a programmer fails to add, it throws a compile time error.

**WAJP to convert number into words**

```
import java.util.Scanner;
class ConvertNumberToWord
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in)	;
		System.out.println("Enter a number : ");
		int num = sc.nextInt();
		String words = "";
		while(num>0)
		{
			int rem = num%10;
			switch(rem)
			{
				case 0 : words = "zero "+words;break;
				case 1 : words = "one "+words;break;
				case 2 : words = "two "+words;break;
				case 3 : words = "three "+words;break;
				case 4 : words = "four "+words;break;
				case 5 : words = "five "+words;break;
				case 6 : words = "six "+words;break;
				case 7 : words = "seven "+words;break;
				case 8 : words = "eight "+words;break;
				case 9 : words = "nine "+words;break;
			}
			num/=10;
		}
		System.out.println(words);
	}
}
```

**WAJP to reverse a number**

```
import java.util.Scanner;
class ReverseNumber
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter a number : ");
		int num = sc.nextInt() ;
		int dup = num ;
		int rev = 0 ;
		while(num!=0)
		{
			int rem = num%10 ;
			rev =(rev*10)+rem;
			num/=10;
		}
		System.out.println(dup+" in reverse is "+rev);
	}
}
```

**WAJP to check palindrome number**

```
import java.util.Scanner;
class PalindromeNumber
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter a number : ");
		int num = sc.nextInt() ;
		int dup = num ;
		int rev = 0 ;
		while(num!=0)
		{
			int rem = num%10 ;
			rev =(rev*10)+rem;
			num/=10;
		}
		System.out.println(dup==rev?dup+" is palindrome.":dup+" is not palindrome.");
	}
}
```

**WAJP to check palindrome name**

```
import java.util.Scanner;
class PalindromeName
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter a name : ");
		String name = sc.next() ;
		String rev = "" ;
		int i = 0 ;
		while(name.length()>i)
		{
			rev=name.charAt(i)+rev;
			i++;
		}
		System.out.println(rev.equals(name)?name+" is palindrome.":name+" is not palindrome.");
	}
}
```

**WAJP to print multiplication table**

```
import java.util.Scanner;
class MultiplicationTable
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter a number : ");
		int num = sc.nextInt();
		int i =1;
		while(i<=10)
		{
			System.out.println(num+" X "+i+" = "+(num*i));
			i++;
		}
	}
}
```

**WAJP to find factors of given number using while loop**

```
import java.util.Scanner;
class FactorsOfANumberInWhileLoop
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in) ;
		System.out.println("Enter a number : ");
		int num = sc.nextInt();
		int i = 1;
		while(i<=num)
		{
			if(num%i==0)
			System.out.print(i+" ");
			i++;
		}
	}
}
```

**WAJP to find even numbers between the range given by the user**

```
import java.util.Scanner;
class EvenNumbersBetweenRange
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in) ;
		System.out.println("Enter first value of your range : ");
		int num1 = sc.nextInt();
		System.out.println("Enter last value of your range : ");
		int num2 = sc.nextInt();
		System.out.println("Even numbers between "+num1+" and "+num2+" are :");
		while(num1<=num2)
		{
			if(num1%2==0)
			System.out.print(num1+" ");
			num1++;
		}
	}
}
```

★ **Difference between for and while loop:-**

| for loop | while loop |
|---|---|
| i. We use for loop when we know the number of iterations. | i. We use while loop when we don't know the number of iterations. |
| ii. Initialization, condition and updation are declared at the same line. | ii. Initialization, condition and updation are not declared at the same line. |
| iii. Initialization (counter variable) can be done before the loop or inside the loop. | iii. Initialization (counter variable) must be done before starting of the loop. |
| iv. The loop iterate infinite times when we are not specifying any condition (compiler provides default condition i.e. true). | iv. It throws a compile time error if we are not specifying the condition. |

### 3. do while loop :-

- It is a loop statement which helps the programmer to execute some set of instructions repeatedly.
- Whenever our first priority is to perform a task and then checking the condition we should use do while loop.
- Syntax:-

```
Initialization;
do
{
    //statements
    Updation;
} while(condition);
```

- e.g. lowercase 'a' to 'z' using do while loop

```
class DoWhileExample
{
    public static void main(String[] args)
    {
            char i = 'a';
            do
            {
                    System.out.print(i+"  ");
                    i++;
            }
    }while(i='z');
}
```

**Control flow of do while loop:-**

- First the control initialize the counter variable.
- Then it executes the do block where we have specified our updation statement for the loop and then control checks for the condition.
- If the condition is true control goes for the next iteration and executes the do block again.
- But if the condition is false the loop gets terminated.

**Note:-**

In do while loop even if the condition is false the number of iteration is one.

Program:-

```
class DoWhileExample
{
        public static void main(String[] args)
        {
                int i ;
                do
                {
                        i = 1;
                        System.out.print(i+"  ");
                        i++;
                }
        }while(i=10);
}
```

In the above program we are initializing the counter variable inside the do block because of which in every iteration the counter variable is getting reinitialize as 1, and it creates an infinite loop.

- Examples of do while loop :-
  exam, auto riksha fare, light bill, atm, password, java compiler, salary, payment.

## ★ Difference between while loop and do while loop

| while loop | do while loop |
|---|---|
| i. In while loop condition is checked first and then loop body gets executed. | i. In do while loop first loop body is executed and then condition is checked. |
| ii. Condition is checked at the entry that's why its also called as entry control loop. | ii. Condition is checked at the exit that's why its also called as exit control loop. |
| iii. No semicolon is used as a part of syntax. | iii. Semicolon is used mandatory at the end of the condition. |
| iv. It allows initialization of counter variable before entering the loop body. | iv. It allows initialization of counter variable before or after entering the loop body. |
| v. We use while loop when condition is more important. | v. We use do while loop when process (task) is more important. |
| vi. Minimum number of iteration is zero if the condition is false. | vi. Minimum number of iteration is one if the condition is false. |

**1.**
```
while(true)
{
        System.out.println("hello");
}
System.out.println("by");
```

**Output:-**
//compile time error unreachable statement after loop because condition is constant

**2.**
```
while(false)
{
        System.out.println("hello"); //compile time error unreachable statement after loop because
                                      //condition is constant
}
System.out.println("by");
```

**3.**
```
do
{
        System.out.println("Hello");
)while(true);
System.out.println("by");
```

**Output:-**
//compile time error unreachable statement after loop because condition is constant

**4.**
```
do
{
        System.out.println("Hello");
)while(false);
System.out.println("by");
```

**Output:-**
hello
by

**WAJP to check whether user entered password is correct or not**

```
import java.util.Scanner;
class Password
{
	public static void main(String[] args) throws InterruptedException
	{
		Scanner sc = new Scanner(System.in)	;
		String storedPassword = "akshay";
		int duration = 5000;

		//label for outer loop
		outerLoop:
		for (; ; ) //infinite loop it will break only
					//when user is entering correct pin
		{
			int attempt = 3; // initially user has 2 attempts in every cycle
			do
			{
				System.out.println();
				System.out.println("Enter your password : ");
				String userPassword = sc.next();
				//validate if store and user ented pass is correct
				if(storedPassword.equals(userPassword))
				{
					System.out.println("Phone has been unlocked. ");
					break outerLoop;
					//breaking the outer loop if password is correct
				}
				else
				{
				System.out.println("Wrong password entered "+(attempt-1)+" attempts left.");
				}
				attempt--;	//updating the attempts
			}while(attempt!=0);	//check the condition
			System.out.println();
			System.out.println("Phone is locked for "+(duration/1000)+" seconds.");
			//pause the execution for duration ms
			Thread.sleep(duration);
			duration*=2;	//updating the duration for next iteration
		}
	}
}
```

★ **Thread.sleep(int milliseconds) :-**

- In java the execution is done by thread JVM creates a “main” thread and that main thread is responsible for the execution.
- If we want to pause the execution of current thread we have to invoke “**sleep()**” method from **Thread** class using class name as a reference and we have to specify the argument in milliseconds.
- e.g. Thread.sleep(2000);
- The execution will be paused for 2 seconds i.e. 2000 milliseconds, but whenever thread is sleeping there are chances other threads might interrupt sleeping thread.
- That’s why we need to handle the exception using **throws**.
- e.g. public static void main(String[] args) throws **InterruptedException**

## 2. Continue:-

- continue is a keyword and a branching statement in java which is used to skip an iteration of a loop based on condition.
- Once the continue statement is executed control is transferred back at the top of loop i.e. for the next iteration.
- In simple words the statements after continue will be skipped and the control will go for the next iteration.
- e.g.

**WAJP to print 1 to 100 other than a number which is divisible by 5**

```
class Demo
{
    public static void main(String[] args)
    {
            for(int i = 1; i<=100 ; i++)
            {
                    if(i%5==0)
                            continue;
                    System.out.print(i+" ");
            }
    }
}
```

**Output:-**

1 2 3 4 6 7 8 9 11 …… 99

**WAJP to find vowels from 'A' to 'Z' using continue**

```
class Demo
{
    public static void main(String[] args)
    {
            for(char i = 'A'; i<='Z'; i++)
            {
                    if(!(i=='A'||i=='E'||i=='I'||i=='O'||i=='U'))
                            continue;
                    System.out.print(i+" ");
            }
    }
}
```

## ★ Label loops:-

- Label is an identifier which is used to provide a name for loops, basically it is a variable.
- Labels are used to transfer a control from one block to another it is always used with continue and break statement.
- To create a label we need to follow the rules of identifier as well as variable convention (camel case).
- Labels are placed at the top of a loop with colon at the end.
- e.g for loop (label)

```
class Demo{
        public static void main(String[] args){
                outerLoop:
                for(int i=1;i<=3;i++){
                        inneLoop:
                        for(int j =1;j<=3;j++){
                                if(i==2){
                                        break;
                                }
                                System.out.print("i = "+i+" j = "+j);
                        }
                }
        }
}
```

**e.g. while loop(label)**

```
class WhileLoopExample{
        public static void main(String[] args){
                outerLoop:{
                        int i = 1;
                        while(i<=10){
                                System.out.println(i+" ");
                                if(i==5){
                                        break outerLoop;
                                }
                                i++;
                        }
                }
        }
 }
```

**WAJP to print ODD digits from user entered number.**

```
import java.util.Scanner;
class Demo
{
        public static void main(String[] args)
        {
                System.out.println("Enter a number :");
                int num = new Scanner(System.in).nextInt();
                String op ="";
                while(num>0)
                {
                        int rem = num%10;
                        if(rem%2!=0)
                                op=rem+"  "+op;
                        num /= 10;
                }
                System.out.println(op);
        }
}
```

# ★ Methods:-

- It is a set of instructions or block of code which is used to perform specific task.
- Methods gets executed only when it is called.
- Syntax :-

```
access modifier   non-access modifier   return type   method name   (formal arguments)
{
        //statements
}
```

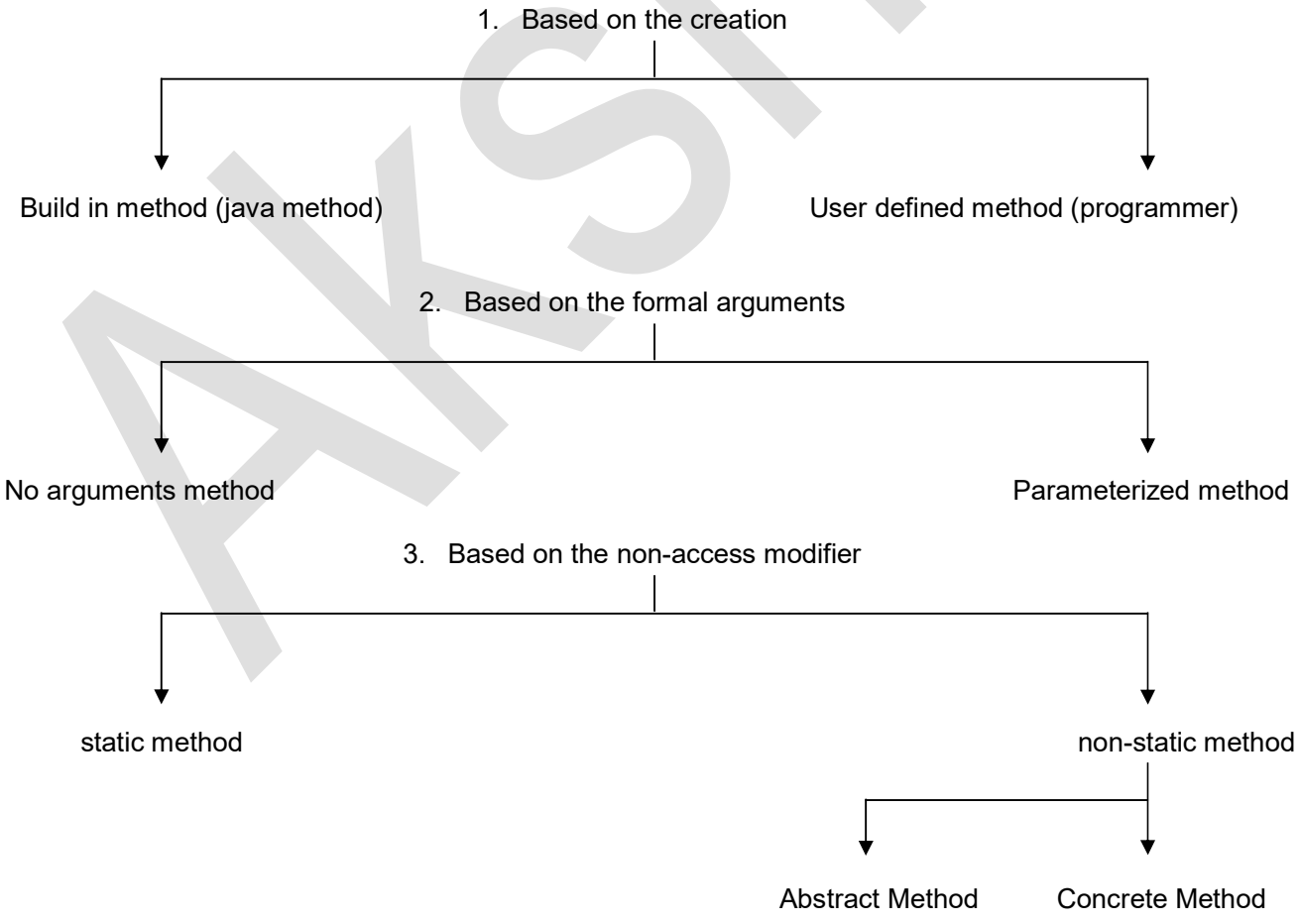
Parameters

- Create your own method:-

```
public static void akshayMali()
{
        //Statements
}
```

- **Advantages of methods:-**
  1. Code reusability (Define once and use multiple times).
  2. Code modularity (Dividing a bigger module into smaller sub module).
  3. Modification becomes easy.
  4. Increases code readability.

- **Method classification :-**



1. **Based on the creation:-**

   i. **Built-in method:-**
   - A method which is been defined by java is known as built in method.
   - e.g. charAt(), nextInt(), print(), println()

   ii. **User defined method:-**
   - A method which is been defined by a programmer (user) is known as user defined method.
   - e.g. isEvenOdd(), isPalindrome(), isPrime()

2. **Based on non-access modifier:-**

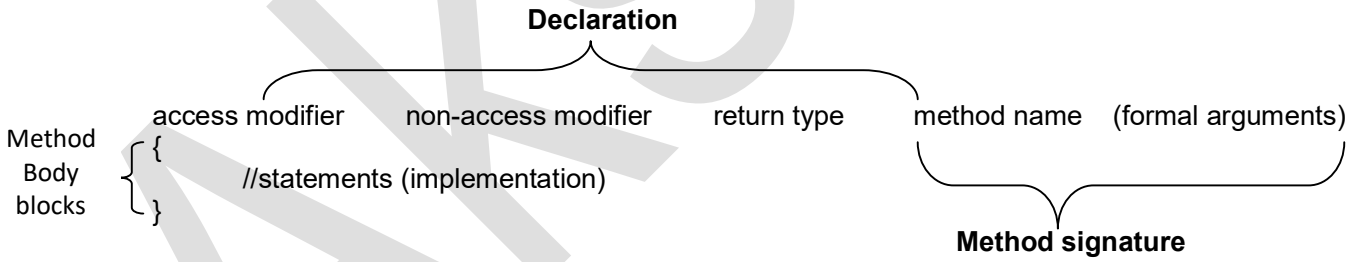
   i. **Static method:-**
   - A method which contains static modifier (keyword) in its declaration statement is known as static method.

   ii. **Non-static method:-**
   - A method which does not contain static modifier in its declaration statement is known as non static method.
   - Non-static methods are classified as:-
   - i. Abstract method ii.Concrete method

   **Abstract method:-**
   - A method which does not contains any implementations (i.e. method body) is known as abstract method.

   **Concrete method:-**
   - A method which contains implementations (i.e. method body) is known as concrete method.

★ **Terminologies of methods:-**



## ★ Modifier:-

- Modifiers in java are the keywords which are used to alter (change) the behaviour of java members.
- There are two types of modifiers :-
  1. Access modifier / Access specifier
  2. Non-access modifier / Non-access specifier

### 1. Access modifiers:-

- Access modifiers are keywords which are used to control the accessibility (visibility) of fields (variables, methods, constructors, class, etc.).
- There are four access modifiers in java
  1. public
  2. protected
  3. default (package level modifier)
  4. private

| Modifiers (access modifier) | Same package (same class) | Same package (outside class) | Outside package (only child class) | Outside package (anywhere) |
|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | ✓ |
| protected | ✓ | ✓ | ✓ | X |
| default | ✓ | ✓ | X | X |
| private | ✓ | X | X | X |

### 1. public access modifier:-

- It can be used with various java members such as top level class, inner class, constructors, method, variable, interface, etc.
- This members can be accessed within same package anywhere as well as outside package.

### 2. protected access modifier:-

- Protected modifiers can be used with various java members such as inner class, variables, method, constructor, etc.
- A protected member can be accessed anywhere within the same package as well as outside package only within the child class.

### 3. default access modifier:-

- default access modifier can be used with various java members such as top level class, inner class, constructor, variable, method, interface, etc.
- default modifier is also known as package level modifier because default modifier can be only accessed within the same package anywhere.

### 4. private access modifier:-

- It can be used with various java members such as inner class, variable, method, constructor, etc.
- A private member cannot be access outside its class where it has been declared.

2. **Non access modifier:-**
   - Non access modifiers are keywords in java which are used to alter the behaviours of java members.
   - There are different types of non-access modifiers:-
     1. static
     2. final
     3. synchronized
     4. volatile
     5. transient
     6. strictfp
     7. native
     8. abstract

★ **Return type:-**

- In method declaration statement specifies the type of data needs to be returned or nothing after invoking a method.
- If a method doesn't want to return anything the return type of method should/must be void.
- If a method wants to return something then return type must be the specific data type what a method want to return.
- The return type data can be primitive or non-primitive.
- e.g.

```
class MethodExample
{
        public static void main(String[] args)
        {
                System.out.println("Execution starts");
                m1();
                System.out.println("Execution ends");
        }
        void m1()
        {
                System.out.println("LINE 1");
                System.out.println("LINE 2");
                System.out.println("LINE 3");
                int op = m2();
        }
        int m2()
        {
                int a = 10;
                int b = 20;
                int op = a+b;
                return op;
        }
        void result()
        {
                System.out.println(op);
        }
}
```

**Output:-**

```
Executions starts
LINE 1
LINE
LINE 3
30
Execution ends
```

## ★ Method name:-

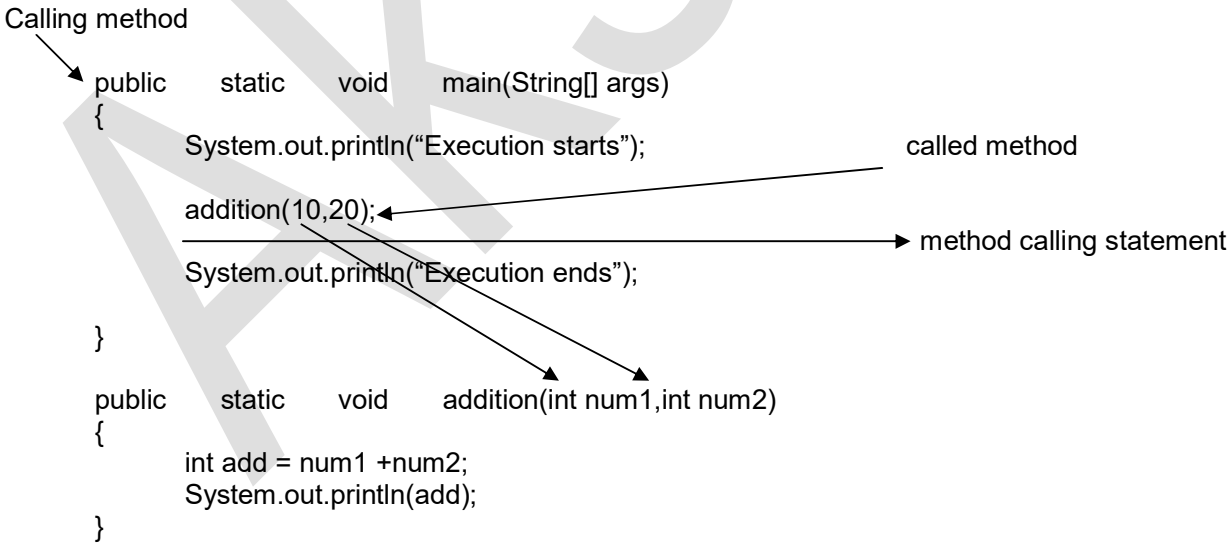
- It is an identifier which is used to invoke a method.
- Whenever we declare a method name we need to follow rules of identifier as well as method conventions (camel case).

## ★ Formal arguments:-

- Formal arguments are the variables created inside method declaration statements.
- Formal arguments (variables) are also known as parameters.
- They are the local variables of that method.
- Formal arguments are created when a method wants to accept any input data from the calling method at time of invoking.
- e.g.

```
class MethodExample
{
        public static void main(String[] args)
        {
                m1();
                m1(10);
                m1("hello");
                m1(10,20);
                byte a =1;
                m1(a);
        }
        void m1(int num)
        {
                System.out.println("Hello from m1()");
        }
}
```

- Length of the actual and formal argument must be same and the sequence of parameters must be same or the data must be compatible with a formal argument.



★ **Actual argument:-**
- Arguments (data) passed inside a method call statement are known as actual arguments.
- e.g. addition(10,20);
- These arguments are used to initialize the formal arguments (variable).
- If we are not invoking a method the formal arguments will not get initialized.

**Note:-**
An actual argument can be variable / literal / expression, it should not be a variable declaration.

★ **Method call statements:-**
- The statement which is used to invoke a method is called as method call statement.
- Method call statement consists of method name and actual argument.

★ **Calling method:-**
- A method which contains the actual call (method calling statement) is known as calling method.

★ **Called method:-**
- A method which is called by an another method is known as called method.

**3. No argument method:-**
- A method which does not contains any formal arguments is known as a no argument method.
- In no argument method we does not pass any argument (actual argument) at the time of calling a method.
- We create this method when a method does not want any input data from the calling method.
- e.g.

```
class Example
{
    public static Scanner sc = new Scanner(System.in);
    public static void main(String[] args)
    {
            evenOdd();
            reverseNumber();
    }
    public static void evenOdd()
    {
            System.out.println("Enter a number : ");
            int num = sc.nextInt();
            if(num%2==0)
                    System.out.println(num+"  is even.");
            else
                    System.out.println(num+"   is odd");
    }
    public static void reverseNumber(){
            System.out.println("Enter a number : ");
            int num = sc.nextInt();
            int rev = 0;
            for(num>0;num%10){
                    rev = rev*10+(num%10);
                    System.out.println(rev);
            }
    }
}
```

**Note:-**

- A method get executed only when it is called.
- If we are not calling a method still compiler will compile it and checks for syntactical mistakes a method must be invoked using a method call statement.
- We can call a method 'n' number of times.
- We can declare multiple variables with same name in different method blocks.
- We cannot invoke a local variable from one method to another method.

4. **Parameterized method:-**
   - A method which contains formal arguments is known as parameterized method.
   - We create formal arguments when a method wants to accept any input data from the calling method.
   - e.g.

```
class Demo
{
    public static void main(string[] args)
    {
            m1(byte(10));
            m1(10);
    }
    public static void m1(byte a)
    {
            System.out.println("hello from m1(byte a)");
    }
    public static void m1(int a)
    {
            System.out.println("hello from m1(int a)");
    }
}
```

   - When we invoke a method the actual argument are mapped with the formal arguments, this process of mapping the arguments done by the compiler that is known as **<u>binding</u>.**
   - The method call statements will be binded with method declaration at the compile time the compiler will decide which method must be invoked.

**break:-**

- It is a keyword and branching statement in java which is used to transfer the control back to the calling method.
- return statement stops the execution of a method and control transfer back.
- Once the execution of a method is completed control will transfer back if we are using return statement or not.
- return statement is most oftenly used when a method wants to return a data block to the calling method.

**Note:-**

i. Return statement must always be at the end of block.
ii. The statements after return will be unreachable.

**break** :- it terminates the execution of loop / switch.
**continue** :- it skips the current iteration of a loop.
**return** :- it terminates the execution of a method.

★ **Return type:-**

```
class Example
{
  public static void main(String[] args)
  {
        System.out.println("Execution starts");
        m1();
        m2();
        System.out.println("Execution ends");
  }
  public static int m1()
  {
        System.out.println("Line 1");
        System.out.println("Line 2");
        System.out.println("Line 3");
  }
      public static int m1()
  {
        System.out.println("Line m2() line");
        System.out.println("Line 2");
        System.out.println("Line 3");
        System.out.println("Line 4");
  }

}
```

- When the return type of method is void after execution of a method control is transfer back to the calling method and does not carry or return any data to the calling method.
- We can use the return statement at the end of a method block which is optional in this case.

e.g.

```
class Example
{
        public static void main(String[] args)
        {
                m1();
        }
        public static int m1()
        {
                System.out.println("Line 1");
                System.out.println("Line 2");
                System.out.println("Line 3");
                System.out.println("Line 4");
                System.out.println("Line 5");
        }

}
```

Output:-

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

**We cannot return a value (data) when the return type of method is void.**

```
class Example{
  public static void main(String[] args){
        m1();
  }
  public static int m1(){
        System.out.println("Line 1");
        return;
  }
}
```

**When the method wants to return some data inside the return statement we need to pass the value.**

```
class Example{
  public static void main(String[] args){
        m1();
        m2();
  }
  public static int m1(){
        System.out.println("m1()");
        return(byte) 10 ;
  }
  public static boolean m2(){
        System.out.println("m2()");
        return 10 ;
  }
}
```

**Methods return value and methods return type must be compatable.**

| return type | value |
|---|---|
| int | int / short / byte |
| boolean | boolean |
| String | String |
| double | byte / short / int / long / char / float / double |
| char | char |
| byte | byte |
| long | byte / short / int / long |

★ **Can we call a main() explicitly?**

- Yes, we can call main() method explicitly by passing the actual parameter as String[].
- In java the execution always starts from main() method and ends at main().
- JVM always searches for the main method at the runtime to start the execution.
- So if we call main() explicitly it will create a recursion.
- e.g.

```
class Example
{
  public static void main(String[] args)
  {
        System.out.println("Hello from main(String[] agrs)");
        m1();
  }
  public static void m1()
  {
        System.out.println("Hello from m1()");
        String[] arr = new String[1];
        main(arr);
  }
}
```

# ★ Method Recursion:-

- A method calling itself directly or indirectly is called as recursion.

**Note :-**

Direct calling means a calling method and called method are same.

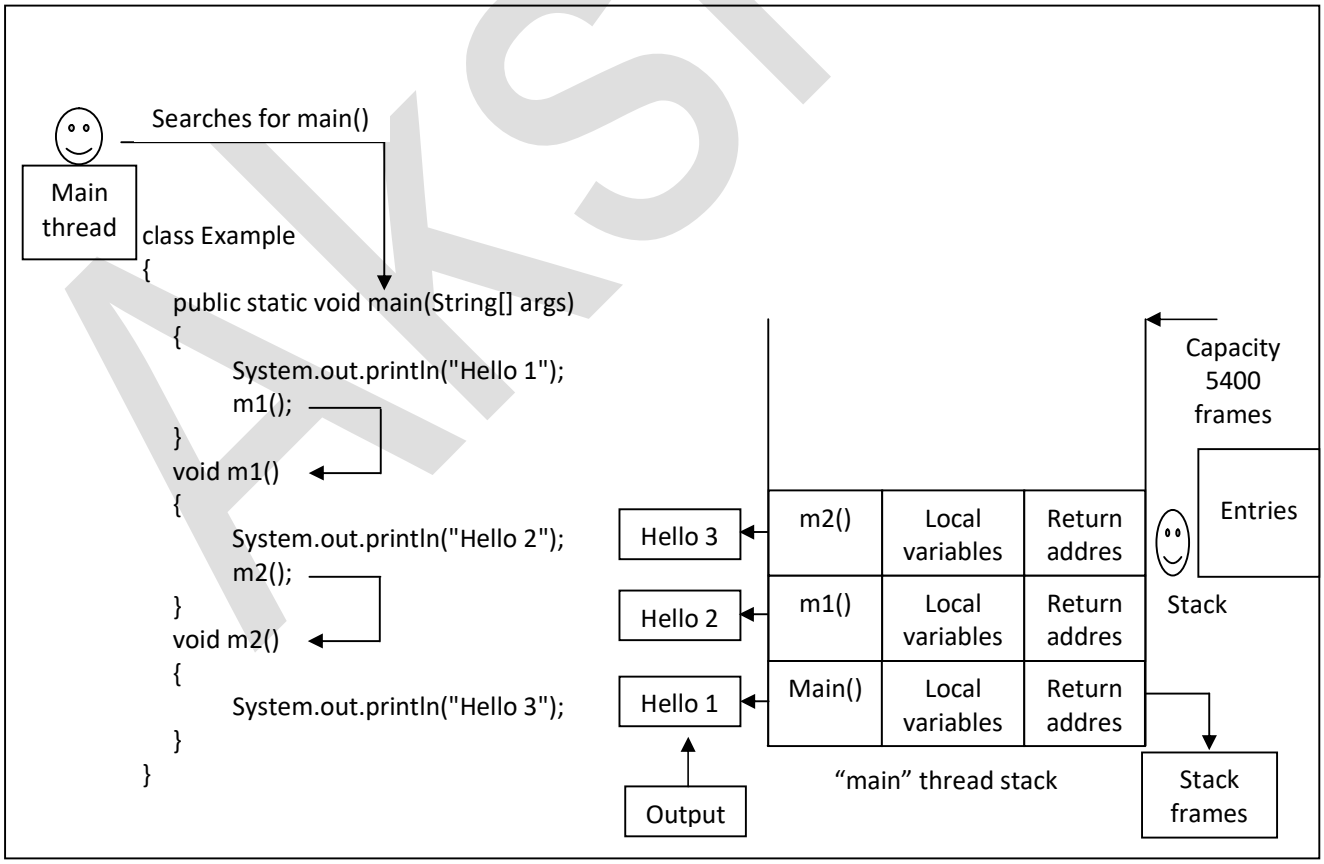
**e.g. for direct calling**

```
class Example
{
        public static void main(String[] args)
        {
                System.out.println("Hello form main(String[] args)");
                m1();
        }
        public static void m1()//calling method
        {
                System.out.println("Hello from m1()");
                m1();   //called method
        }
}
```

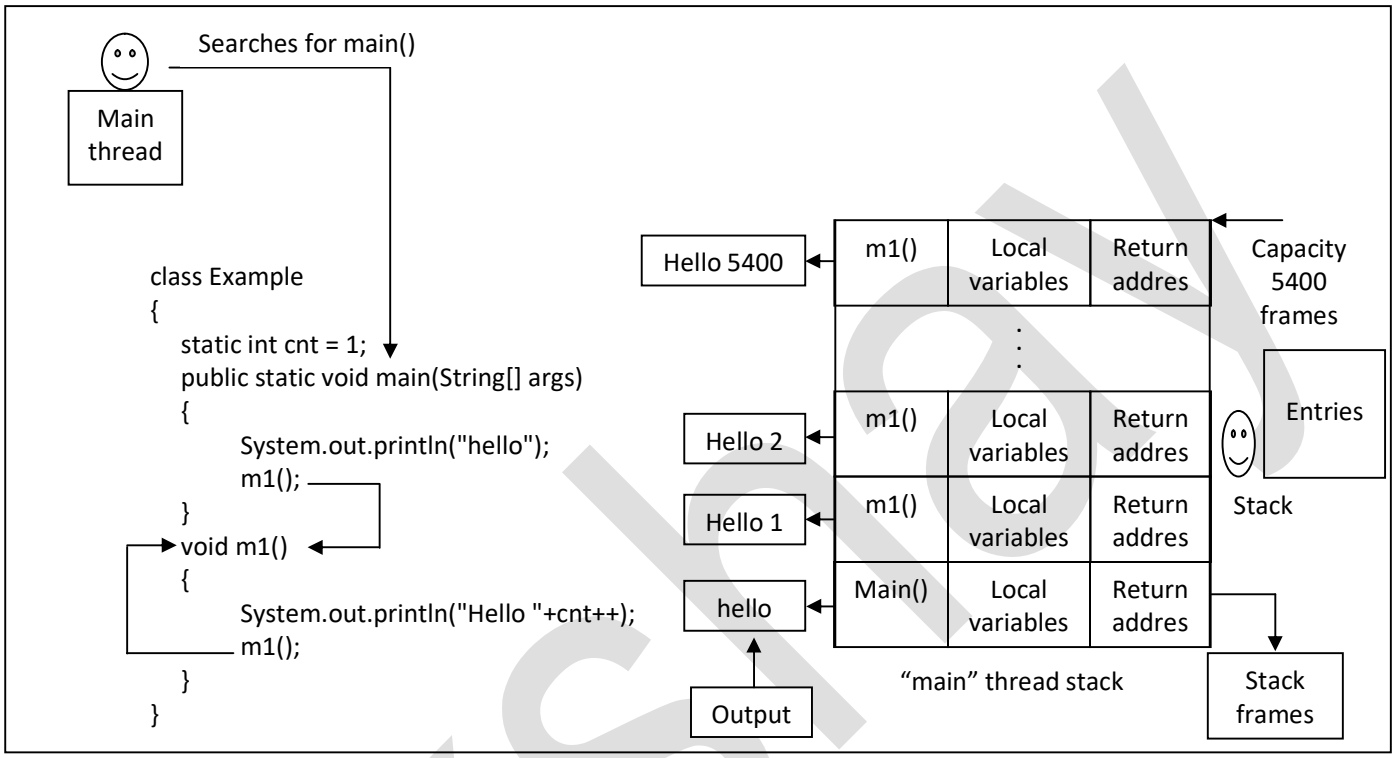
**e.g. for indirect calling**

```
class Example
{
  public static void main(String[] args)
  {
        System.out.println("Hello form main(String[] args)");
        m1();
  }
  public static void m1()//calling method
  {
        System.out.println("Hello from m1()");
        m2();   //called method
  }
  public static void m2()//calling method
  {
        System.out.println("Hello from m2()");
        m1();   //called method
  }
}
```

- At the runtime JVM creates a thread for the execution of a program.
- The thread created by JVM is known as a "main" thread.
- Main thread is responsible for the execution and a main thread at execution always searches for "main()".
- Runtime machine creates a runtime stack for a thread.
- If we have n number of threads it will create n number of stacks inside the stack area.
- One runtime stack be allocated one pc register (hardware) and one native method stack.
- Every stack which is been created will have a default initial capacity based on device.



- main thread invokes main() implicitly.
- Every method call will create a stack frame inside a runtime stack of that particular thread.
- Stack frame consists of local variables, return address and method signature with its formal arguments.
- Once the whole execution of a method is completed stack frame is destroyed of that particular method.
- And once the whole execution of the methods are completed the created runtime stack will be destroyed by the JVM.




- When we perform method recursion in this we continuously invokes the method and every method call will create a stack frame.
- Stack has a capacity if we try to create more number of stack frames than the capacity, it will create a StackOverFlowError.

## ★ StackOverflowError :-

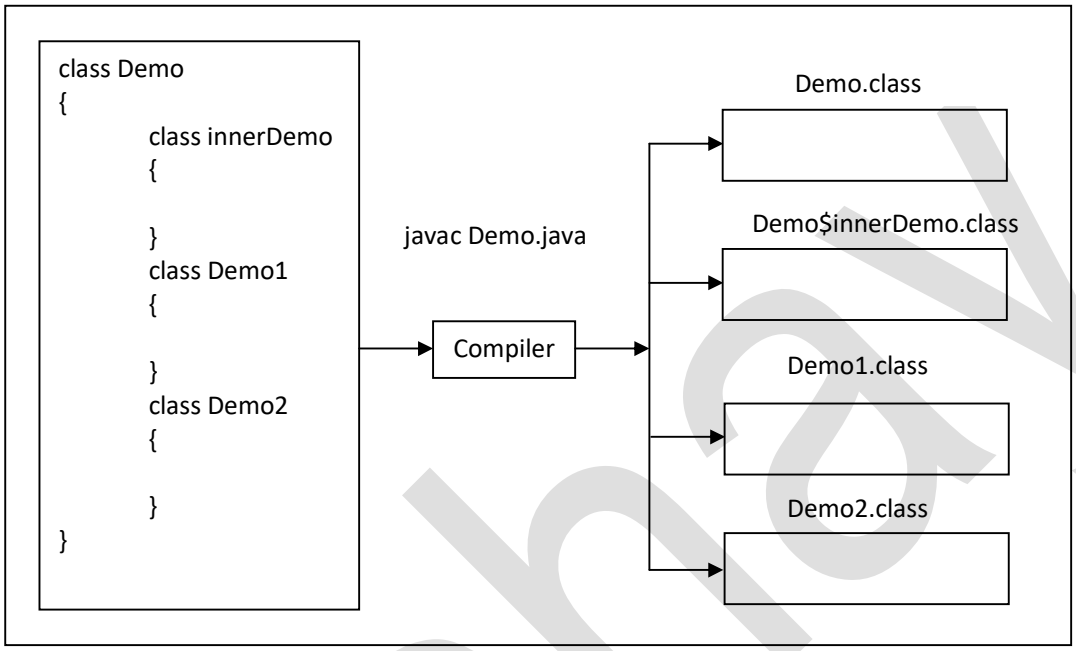
- It occurs when we perform recursion and the number of frames are more than the capacity.
- To overcome this problem we can either increase the capacity of a stack or we can specify a base condition to stop the recursive call.
- To increase the capacity of stack of we need to execute a command at the runtime,
  **java –Xss <size> classname**
  e.g.
  1. java –Xss 5m MethodRec //m for mb
  2. java –Xss 500k MethodRec //k for kb
  3. java –Xss 1g MethodRec //g for gb
- Base condition to avoid method recursion
- e.g.

```
class Example
{
        static int num =1;
        public static void main(String[] args)
        {
                printNumbers();
        }
        public static void printNumbers()
        {
                System.out.print(num++ +" ");
                if(num==11)    //base condition
                {
                        return;
                }
                printNumbers();
        }
}
```

- In the above program printNumbers() has a conditional statement which gets executed in every method call which keeps a track of the function calls (iterations).
- So whenever this condition satisfies the block is executed and recursive call gets terminated because we are using a return statement.
- Return statement transfers the control back to the calling method.

# ★ Multiple classes :-

- It is design process where a single source code contains multiple classes in it.
- Java compiler generates n number of byte code for n number of classes.
- At the time of execution we should always use byte code which contains main method in it.
- Class which has been created inside another class is called as inner class.




★ **public class :-**

- The class which is declared with public modifiers is known as public class.
- e.g.

```
public    class    Example
{

}
```

- We cannot have more than one public class inside a single source code.
- File name and public class name must be same.
- If we try to create a public class and the file name different compiler throws a compile time error.
- It is always recommended to define main() method inside public class, because that class is going to drive the execution process.
- **Source code:- Example.java**

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

    }
}
class Example2{

}
class Example3{

}
```

**Example 2 :- Example.java**

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

    }
}
public class Example2{

}
class Example3{

}
```

Output:-
public class Example must be declared in file name Example.java

## ★ Static:-

- static is a non-access modifier and a keyword in java.
- Whenever we create a static member in java they are stored inside a class area.
- static keyword in java used to share same variables and methods from one class to another class.
- static modifier can be prefixed with various java members such as,
  1. static blocks
  2. static variables
  3. static method
  4. static innerclass
- The members which is prefixed with static modifier are known as static members.
- We can access a static member any where inside the same class directly.
- Also we can access them outside class using class name as a reference.
- To invoke a static member we don't need to instantiate (object creation) a class.
- So static members in java are used for memory utilization as one copy of the member can be shared everywhere.
- Accessing a static member can be faster than non-static members.

### 1. static method:-

- A method which contains static modifier in its declaration is known as static method.
- A static method can be accessed directly anywhere within the same class.
- e.g.

```
public class Example
{
    public static void main(String[] args)
    {
            m1();
            m2();
    }
    public static void m1()
    {
            System.out.println("hello from m1()");
    }
    public static void m2()
    {
            System.out.println("hello from m2()");
    }
}
```

- **We can also access methods of a same class using class name as a reference.**
- e.g.

```
public class Example
{
        public static void main(String[] args)
        {
                Example.m1();
                Example.m2();
        }
        public static void m1(){
                System.out.println("hello from m1()");
         }
         public static void m2(){
                System.out.println("hello from m2()");
         }

}
```

- **We cannot access any non static members inside a static context (block) directly without creating an object.**
- e.g.

```
public class Example
{
        String a = "abc";
        public static void main(String[] args)
        {
                m1();                   //CTE non static method
                System.out.println(a); //CTE non static variable
        }
        public void m1(){
                System.out.println("hello from m1()");
         }
}
```

- **We can access static member directly within the same class inside a non static context.**
- e.g.

```
public class Example
{
          static String a = "abc";
        public static void main(String[] args)
        {

        }
        public static void m1()
         {
                System.out.println("hello from m1()");
         }
         public void m2()
         {
                System.out.println(a);
                m1();
         }

}
```

- **If we want to access static method of from one class to another class we have to use class name as a reference (mandatory).**
- e.g.

```
public class Example1
{
	static String a = “abc”;
	public static void main(String[] args)
	{
		System.out.println(“main() starts”);
		m1();
		System.out.println(“main() ends”);
	}
	public static void m1()
	{
		System.out.println(“m1() starts”);
		Example2.m2();
		System.out.println(“m1() ends”);
	}
}
public class Example2
{
	public static void m2()
	 {
		System.out.println(“Example2 m2() starts”);
		m3();
		System.out.println(“Example2 m2() ends”);
	 }
	 public static void m3()
	 {
		System.out.println(“Example2 m3() starts”);
		System.out.println(“Example2 m3() ends”);
	 }

}
```

**Output:-**

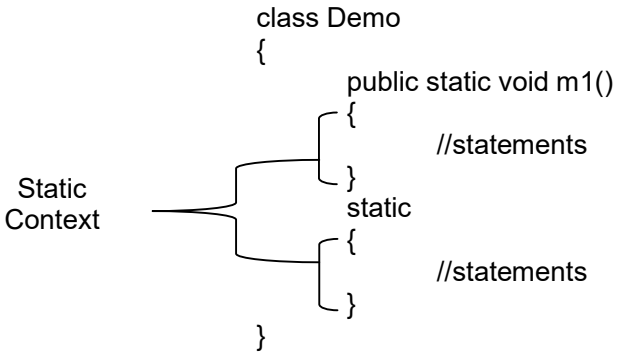
```
main() starts
m1() starts
Example2 m2() starts
Example2 m3() starts
Example2 m3() ends
Example2 m2() ends
m1() ends
main() ends
```

- **If you want to access a static member from one class to another inside a non static context we should use class name as a reference.**
- e.g.

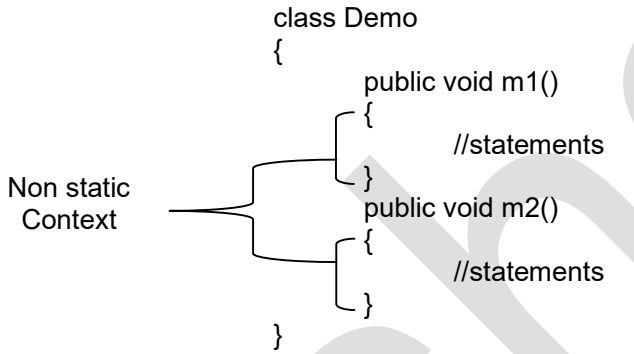
```
class Example1
{
        public static void main(String[] args)
        {

        }
        public static void m1()
        {
                System.out.println("Hello form m1() static);
        }
}
class Example2
{
        public void m2();
        {
                System.out.println("Hello from m2() non static");
                Example1.m1();
        }
}
```

★ **static context:-**

- A block in a class which is associated with static modifier is known as static context.
- e.g.




- A block in a class which is not associated to static modifier is known as non static context.




# ★ static initializers :-

- The static initializers in java are executed when a class is loaded by the JVM.
- They are primarily use to initialize static variables or to perform one time set of task that needs to happen before the actual execution.

**Characteristics of static initializers:-**

i. They are executed when a class loading process happens.
ii. Used to initialize static variables or execute some static logic.
iii. We can have multiple static blocks in a class.
iv. No need to create any instance.

- static initializers are of two types,
  i. static declaration and initialization
  ii. static block

★ **static variables:-**

- A variable created inside a class block which is prefixed with static modifier is known as static variable.
- Syatax:- static data type variable_name;
- e.g.
  static int a;
  static String b;
  static Scanner sc;

- **A static variable can be accessed anywhere inside same class directly**
- e.g.

```
class StaticVar
{
        static int a;
        static
        {
                System.out.print(a);
        }
        public static void main(String[] args)
        {
                System.out.println(a);
        }
        void m2()
        {
                System.out.println(a);
        }
        public static void m1()
        {
                System.out.println(a);
        }
}
```

- **We can also access static variable outside a class inside any member using class name as a reference**
- e.g.

```
class Staticvar
{
        static String str = “Static var class”;
        public static void main(String[] args)
        {
                System.out.println(str);
                System.out.println(StaticVar.str);
                Demo.m1();
        }
}
class Demo
{
        static String str = StaticVar.str+”Demo 2”;
        public static void m1()
        {
                System.out.println(“m1() Demo2 ”+str);
                m2();
        }
        static void m2()
        {
                System.out.println(“m2() Demo2”+StaticVar.str);
        }
}
```

- static variables are created inside a class block so they are also called as class members.
- static variables gets loaded at the time of class loading process and at the same time memory is allocated to them.
- They are assigned with the default values.
- Integers will be assigned with zero (0), floating point numbers (decimals) will assigned with 0.0 and all the non primitive variables will be assigned with null.
- As they are initialized with default values we can use them without initialization.
- e.g

```
class Example
{
        static byte a;
        static short b;
        static int c;
        static long d;
        static float e;
        static double f;
        static char g;
        static boolean h;
        static Scanner i;
        static Example j;

          public static void main(String[] args)
          {
                  System.out.println(a);
                  System.out.println(b);
                 System.out.println(c);
                 System.out.println(d);
                 System.out.println(e);
                 System.out.println(f);
                 System.out.println(g);
                 System.out.println(h);
                 System.out.println(i);
                 System.out.println(j);
          }
}
```

- We can create global and local variable with a same name.
- If we try to invoke the variable it will always access the local variable first if we want to access the global variable we need to use class name as a reference.
- e.g.

```
class Example
{
        static String str = “GLOBAL”;
        public static void main(String[] args)
        {
                String str = “LOCAL”;
                System.out.println(str);
                System.out.println(Example.str);
                m1();
        }
        public static void m1(){
                System.out.println(str);
        }
 }
```

**Note:-**

i. static variable can be initialized directly same line where it has been declared.

```
e.g.
class Example
{
    static int a = 10;
    static String str;
    str = “GLOBAL”;
}
```

ii. If we want to initialize a static variable by performing some complex operation we may use of static block.
But static block should be declared after the variable declaration.

```
e.g.
class Example1
{
        static int a;
        private static int b = 20;
        protected int c = 30;
        public static void main(String[] args)
        {
                m1();
                a = 10;
                Example2.m2();
        }
        public static void m1()
        {
                System.out.println(a);
                a = 100;
        }
}
class Example2
{
        public static void m2()
        {
                System.out.println(Example9.a);
                System.out.println(Example9.b);
                System.out.println(Example9.c);
        }
}
```

## ★ static block:-

- A block which is prefixed with static modifier is known as a static block.
- It must be created inside a class block.
- static block is a type of static initializer which gets executed at the time of class loading process.
- Syntax:-

```
static
{
    //statements to initialize static variables or some static logic
}
```

**Characteristics of static block:-**

i. static block does not contains any name, so we can't call it explicitly for the execution.
ii. It doesn't contain any formal arguments, so we can't pass any data.
iii. It cannot have any access modifiers.
iv. It cannot have any return value.
v. static blocks gets executed before the main() method.
vi. We can create multiple static blocks in a class they gets executed in a top to bottom order.
vii. static block gets executed once at a time of class loading process.

- e.g.

static block gets executed in top to bottom order.

```
class Example
{
    static
    {
            System.out.println("Hello from static block 1");
    }
    public static void main(String[] args)
    {
            System.out.println("Hello from main()");
    }
    static
    {
            System.out.println("Hello from static block 2");
    }
}
```

- e.g.

```
class Example1
{
        static
        {
                System.out.println("static block Example1");
        }
        public static void main(String[] args){
                System.out.println("main()");
        }
}
class Example2
{
        static
        {
                System.out.println("static block Example1");
        }
        public static void m1(){
                System.out.println("Example2 m1()");
        }
}
```

**If we have a multiple class in a source code**

- Only the class which contain main method can be executed.
- That class will be loaded first, if we are accessing any members from the other classes then only that class will come for the class loading process or else not.

e.g.

```
class Example1
{
        static
        {
                System.out.println("static block Example1");
        }
        public static void main(String[] args)
        {
                System.out.println("main()");
                Example2.m1();
        }
}
class Example2
{
        static
        {
                System.out.println("static block Example2");
        }
        public static void m1()
        {
                System.out.println("Example2 m1()");
        }
}
```

e.g.

```
class Example1
{
        static
        {
                System.out.println("static block Example1");
        }
        public static void main(String[] args)
        {
                System.out.println("main()");
                System.out.println(Example2.a);
                System.out.println("main() 2");
                System.out.println(Example2.a);
        }
}
class Example2
{
        static int a = 10;
        static
        {
                System.out.println("static block Example2");
        }
}
```

e.g.

```
class Example1
{
        static
        {
                System.out.println("static block Example1");
        }
        public static void main(String[] args)
        {
                System.out.println("main()");
                Example2 obj1 = new Example2();
                Example2 obj2 = new Example2();
        }
}
class Example2
{
        static
        {
                System.out.println("static block Example2");
        }
}
```

- When we create an instance of a class JVM loads that class so if they are having any static initializers in that class they will get executed at the time of class loading process and then object creation process will start.
- e.g.

```
class Example1
{
        static
        {
                System.out.println("static block");
        }
        static
        {
                System.out.println(a); //CTE
        }
        static int a;
        public static void main(String[] args)
        {
                System.out.println("main()");
                System.out.println(a);
        }
}
```

**Output:-**
Compile Time Error

1. static initializers (static variable and static block) gets executed in top to bottom order, i.e. If we declare a static variable and static block first variable gets executed then static block.
2. If we declare a static block and static variable first static block gets executed and then static variable.

## Can we execute a java program without an main()?

- Yes, we can execute a java program without a main method, but there are some pre conditions,
  i. We should have JDK version between 11, 17, 21 because this versions supports the direct execution of source code without generating a byte code.
  ii. For executing a source code directly we use the command, **java file_name.java**
  iii. When we use this command compilation, interpretation and execution happens simultaneously and in this process no byte code is generated.
  iv. At the execution time JVM always searches "main() method".
  If we are executing a byte code and there is no main() method define JVM throws a "run time error" please define the method as public static void main(String[] args)
  v. But, we can execute a source code directly JVM first execute the static initializers first and then searches for the main method.
  vi. So we can execute a static initializers and then at the end of the static block we should define System.exit(0);
  vii. System.exit(0); is used to terminate the execution.

**Note:-**

**Conclusion:-** Before JVM searches for main method we are terminating the execution by executing the static block and System.exit(0);

e.g.

```
import java.util.Scanner;
class Example
{
        static Scanner sc = new Scanner(System.in);
        static int op = addition(sc.nextInt(), sc.nextInt());
        static
        {
                System.out.println("Addition is : "+op);
                System.exit(0);
        }
        public static void addition(int num1,int num2)
        {
                int add = num1 + num2;
                return add;
        }
 }
```

## Class loading process for static members :-

1. The class loading process is handle by class loader which is responsible for locating and loading the members of class into the memory area.
2. After loading the class is linked by the linking in which it involves several steps such as,
   i. Verification :- The byte code of the class is verified to ensure that it contains the security rules and doesn't contain any invalid instructions.
   ii. Preparation :- The JVM allocates the memory for the static fields (variables) and initialize them with a default values.
3. Class initialization :- Once the class is loaded by the class loader and linked by the linking then the initialization process starts where it executes the initial values to the static variables.
4. All of the static members are stored inside class area (method area).
5. The method area where the class level data is stored which includes,
   i. static variables
   ii. static methods
6. This area holds all the information about a class which will be shared among all the instances (object) of that class.

**Note:-**

- Before JDK 1.8 method area was part of Heap area but after JDK 1.8 method area does not belongs to Heap area, method area belongs to Meta space.
- Static members in JAVA can be invoked using object (obj) reference, which is not at all recommended because it creates confusion.

# ★ Object:-

- An object is a real world entity.
- An object can be imagine as it is how it is represented in a real world.
- Object contains two parts that are,
  i. States
  ii. Behaviours
- States are used to define an object and behaviours are used to expose it.
  State / Attributes / Properties / Fields → variable
  Behaviour / Action → method
- An object is block of memory which is created at the runtime and stored inside heap area.
- e.g.
  1. Marker
     States :- Colour, brand, size
     Behaviour :- writing, weapon, tool
  2. Chair
     States :- Colour, height, size, type
     Behaviour :- seat, stand, sleep

e.g. Car

```
class Car
{
  String brand;
  String model;
  String type;
  String colour;
  int capacity;
  double price;
  public void carDisplay()
  {
        System.out.println("Brand : "+this.brand);
        System.out.println("Model : "+this.model);
        System.out.println("Type : "+this.type);
        System.out.println("Colour : "+this.colour);
        System.out.println("Capacity : "+this.capacity);
        System.out.println("Price : "+this.price);
  }
}
class CarDriver
{
        public static void main(String[] args)
        {
                Car obj = new Car();
                obj.brand = "Mahindra";
                obj.model = "Scorpio";
                obj.type = "SUV";
                obj.colour = "White";
                obj.capacity = 7;
                obj.price = 2200000;
                obj.carDisplay();
                System.out.println("_____");
                Car obj2 = new Car();
                obj2.brand = "TATA";
                obj2.model = "Harrier";
                obj2.type = "SUV";
                obj2.colour = "White";
                obj2.capacity = 5;
                obj2.price = 2250000;
                obj2.carDisplay();
        }
}
```

**NoteBook**

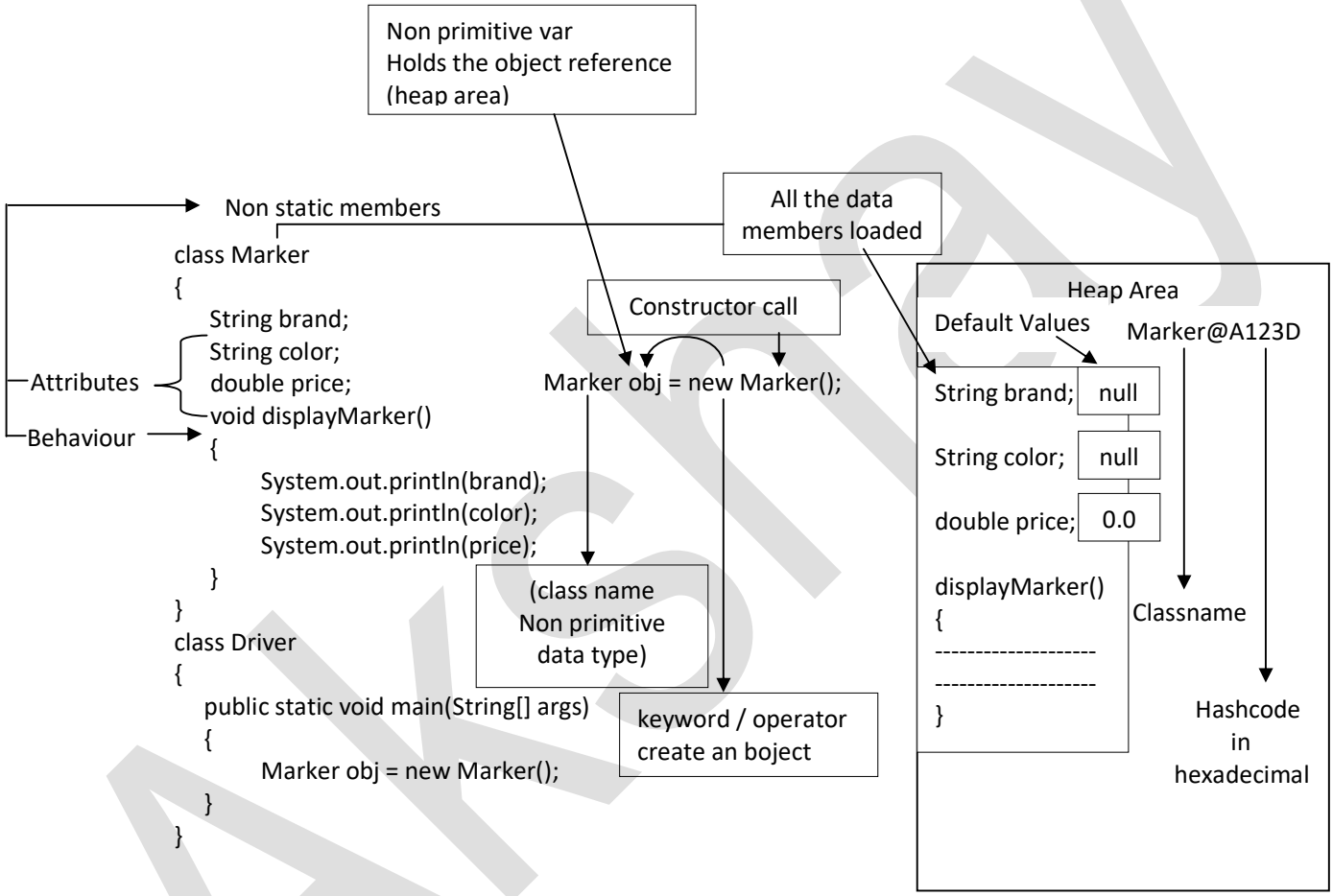
```
class NoteBook
{
        String brand;
        String size;
        String type;
        int pages;
        double price;
        public void notebookDisplay()
        {
                System.out.println("Brand : "+this.brand);
                System.out.println("Size : "+this.model);
                System.out.println("Type : "+this.type);
                System.out.println("Pages : "+this.pages);
                System.out.println("Price : "+this.price);
        }
}
class NoteBookDriver
{
        public static void main(String[] args)
        {
                NoteBook obj = new NoteBook();
                obj.brand = "Classmate";
                obj.size = "A4";
                obj.type = "Single Line";
                obj.pages = 200;
                obj.price = 75;
                obj.notebookDisplay();
                System.out.println("_____");
                NoteBook obj2 = new NoteBook();
                obj2.brand = "Shriram";
                obj2.size = "A4";
                obj2.type = "Single Line";
                obj2.pages = 200;
                obj2.price = 50;
                obj2.notebookDisplay();
        }
}
```

**Note:-**

- In real world before creating an object a blueprint is created.
- The blueprint provides all its specifications of that particular object.
- In java also before creating an object a blueprint must be created with the help of a class.
- e.g.
  Before manufacturing a car, blueprint of a car is created.
  Before constructing a house a prototype of house is created.

# ★ Class:-

- A class is a non primitive data type and a keyword in java.
- It is used to create a blueprint / prototype / template for creating an object.
- Class contains several members in it such as methods, fields, blocks, constructor, etc.
- There are ways to create objects in java,
  1. new keyword
  2. clone() (belongs to object class)
  3. deserialization
  4. newInstance() of Class class
  5. newInstance() of Constructor class




## ★ Object creation and loading process (new keyword):-

1. At the runtime JVM loads the marker class into the memory area if it hasn't been loaded already.
2. The new keyword will create an instance of the marker class inside the heap area and allocates a memory for it.
3. Then all the non static members are loaded inside that object with their default values.
4. The constructor (Marker()) is called to initialize the new object it performs several tasks or accept parameters to set new values.
5. Once the whole object creation process is completed the reference of newly created Marker object reference is assigned to obj variable.
6. By declaring obj with the type Marker we have specifying that obj variable holds the reference of Marker.
7. Marker is a class name i.e. it's a non primitive data type.

★ **Object reference format:-**

class_name@hascode in hexadecimal
e.g.
Marker@abc3635
Car@A64AHGjj

## ★ Non static members:-

- A member which is not prefiexed with static modifier is known as non static member.
- There are several non static members in java such as,
  1. non static variable
  2. non static block
  3. non static method
  4. non static innerclass
- All the non static members are stored inside an object.
- If we want to invoke any of a non static members we need to create an instance of that class.

★ **Non primitive data type:-**

- A non primitive data type is a multi value data.
- Its variable holds the object reference.
- A non primitive data type is also called as reference type because they refer to an object.
- A class in java is non primitive data type.
  e.g.
  class Bike → non primitive data type

  class Student → non primitive data type
- In the above example class is a keyword which is used to create a non primitive data type such as Bike and Student.
- With the help of the non primitive data type we can create an object which will specify the states and behaviours and the non primitive variable holds the reference.

```
class    ClassName
{

        //states        (non static variable)

        //behaviours    (non static method)

}
```

★ **new keyword :-**

- new keyword is used to create an object.
- After new keyword there must be a constructor call.
- new is also an operator (unary operator).

## ★ Non static variable:-

- A variable which is declared inside a class block which is not prefixed with static modifier is known as non static variable.
- e.g.

```
class Student
{
        String name;
        int sid;                    Non static variables
        String branch;
}
```

- Whenever we create an object inside that object a non static variable gets a memory and assigned with default values if no initialization happens.
- Every time when we create an instance a non static variable is allocated with the memory.
- Memory is allocated multiple times for multiple objects.
- Non static variables are also called as instance variable.
- A non static variable is a type of global variable.
- e.g.

```
class Employee
{
        String ename;
        int eid;
        String edesignation;
        double esalary;
        public void displayEmployee()
        {
                System.out.println("Employee name : "+ename);
                System.out.println("Employee ID : "+eid);
                System.out.println("Employee designation : "edesignation);
                System.out.println("Employee salary : "+esalary);
        }
}
class EmployeeDriver
{
        public static void main(String[] main)
        {
                Employee obj = new Employee();
                obj.ename = "Akshay";
                obj.eid = 10;
                obj.edesignation = "Trainee";
                obj.esalary = 25000;
                obj.displayEmployee();
        }
}
```

- **A non static variable can be accessed directly inside a non static context within the same class.**
- **e.g.**

```
class Example{
        String str = "Non static variable";
        {
                //non static block
                System.out.println(str);
        }
        public static void main(String[] args){

        }
        public void m1(){
                System.out.println(str);
        }
}
```

- **If we want to access a non static variable from one class to another class inside any context (static context or non static context) we need to create an object of that class.**
- **e.g.**

```
class Example1
{
        String str = "Non static variable";       //non static variable
        public static void main(String[] args)  //static context
        {
                Example2 obj = new Example2();      //object creation
                obj.m1();
        }
}
class Example2
{
        Example1 obj = new Example1();      //non static variable
        public void m1()         //non static context
        {
                System.out.println("Non static m1() Example2");
                System.out.println(obj.str);     //invoking a non static variable directly
                m2();
        }
        public void m2()         //non static context
        {
                System.out.println("Non static m2() Example2");
                System.out.println(obj.str);
        }
}
```

- **We cannot access a non static variable inside a static context directly without creating an object.**
- **e.g.**

```
class Example1
{
        String str = "Non static variable";       //non static variable
        static
        {
                System.out.println("Static Block");
                Example1 obj = new Example1();
                System.out.println(obj.str);
        }
        public static void main(String[] args)
        {
                System.out.println("main()");
                Example1 obj = new Example1();
                System.out.println(obj.str);
                m1();
        }
        public static void m1()//static context
        {
                System.out.println("m1()");
                Example1 obj = new Example1();
                System.out.println(obj.str);
        }
}
```

**e.g.**

```
class Example1
{
        String str = "Non static variable";          //non static variable
        static Example1 obj = new Example1();        //static variable
        static
        {
                System.out.println("Static Block");
                System.out.println(obj.str);
        }
        public static void main(String[] args)
        {
                System.out.println("main()");
                System.out.println(obj.str);
                m1();
        }
        public static void m1()                      //static context
        {
                System.out.println("m1()");
                System.out.println(obj.str);
        }
}
```

**e.g.**

```
class Example1
{
        String str = "Non static variable";          //non static variable
        static Example1 obj = new Example1();
        public static void main(String[] args)       //static context
        {
                System.out.println("main()");
                Example2 obj = new Example2();       //object creation
                obj.m1();
                obj.m2();
        }
}
class Example2
{
        public void m1()                             //non static context
        {
                System.out.println("Non static m1() Example2");
                System.out.println(Example1.obj.str);
        }
        public void m2()                             //non static context
        {
                System.out.println("Non static m2() Example2");
                System.out.println(Example1.obj.str);
        }
}
```

# ★ Non static method:-

- A method which does not contain static modifier in its declaration is known as non static method.
- Similar like a non static variable, a non static method must be invoked using object reference.
- Without creating any instance of a class we cannot access a non static method inside a static context, it might be within the same class or different class.
- A non static method at the time of class loading process is stored inside a method area.
- e.g.

```
class Example
{
        public static void main(String[] args)
        {
                Example obj = new Example();
                obj.m1();
        }
        public void m1()
        {
                System.out.println("Non static method m1()");
        }
}
```

- **We can invoke a non static method inside a non static context directly without creating any instance within the same class.**

```
class Example
{
        public static void main(String[] args)
        {
                System.out.println("Main()");
                Example obj = new Example();
                obj.m1();
        }
        public void m1()
        {
                System.out.println("Example m1()");
                m2();
        }
        public void m2()
        {
                System.out.println("Example m2()");
        }
}
```

- **If we want to invoke a non static method outside its class we need to create an instance. Using that object reference we can access that member anywhere i.e. either inside static or a non static context.**

```
class Example1
{
        public static void main(String[] args)
        {
                System.out.println("main()");
                Example2.m1();
        }
        public void m3()
        {
                System.out.println("Example1 m3()");
        }
}
class Example2
{
        static Example1 obj = new Example1();
        public static void m1()
        {
                System.out.println("Example m1()");
                obj.m3();
                Example2 obj1 = new Example2();
                obj1.m2();
        }
        public void m2()
        {
                obj.m3();
        }
}
```

- **Execution order of static and non static members:-**

```
class Example
{
  static
  {
        System.out.println("Static block");        //1
  }
  {
        System.out.println("non static block");   //3
  }
  Example()
  {
        System.out.println("Example1 Constructor");     //4
  }
  public static void main(String[] args)
  {
        System.out.println("Main()");    //2
        Example obj = new Example();
        obj.m1();
  }
  public void m1()
  {
        System.out.println("Example m1() non static method");            //5
  }
}
```

# ★ Non static initializers:-

- Non static initializers gets executed at the time of object loading process.
- They get executed every time when we create an object.
- Non static initializers are of two types,
  i. Non static declaration and initialization
  ii. Non static block

## i. Non static block:-

- A block which is declared inside a class block which is not prefixed with a static modifier is known as non static block.
- Syntax:-

```
class class_name
{
    {
        //statements
    }
}
```

**Characteristics of non static block:-**

- Non static block is a type of non static initializer in java.
- This block gets executed at the time of object loading process everytime.
- We can declare multiple non static blocks in a class, they gets executed in top to bottom order.
- The main purpose of a non static block is to perform a set of tasks.
- It doesn't contain any name, so a programmer cannot call it explicitly.
- It doesn't accept any argument.
- It doesn't have any return type.

**e.g. 1**

```
class Example
{
    {
        System.out.println("Non static block 1");
    }
    {
        System.out.println("Non static block 2");
    }
    public static void main(String[] args)
    {
        System.out.println("Main()");
        Example obj = new Example();
    }
    {
        System.out.println("Non static block 2");
    }
}
```

**Output:-**

```
Main()
Non static block 1
Non static block 2
Non static block 3
```

**e.g. 2**

```
class Example
{
    static Example obj = new Example();
    {
            System.out.println("Non static block 1");
    }
    public static void main(String[] args)
    {
            System.out.println("Main()");
    }
    {
            System.out.println("Non static block 2");
    }
}
```

**Output:-**

```
Non static block 1
Non static block 2
Main()
```

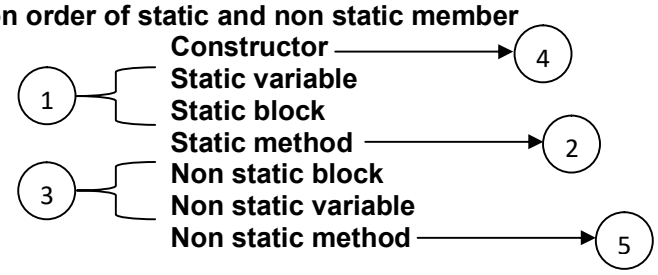
**e.g.3**

```
class Example1
{
    {
            System.out.println("Non static block Example 1");
    }
    public static void main(String[] args)
    {
            System.out.println("Main()");
            Example1 obj1 = new Example1();
            Example2 obj2 = new Example2();
    }
}
class Example
{
    static
    {
            System.out.println("static block Example 2");
    }
    {
            System.out.println("Non static block Example 2");
    }
}
```

**Output:-**

```
Main()
Non static block Example 1
static block Example 2
Non static block Example 2
```

- **Execution order of static and non static member**



# ★ Constructor:-

- A constructor is a block of code similar to a method.
- It is used in object creation.
- Whenever we create an object a constructor must be invoked (mandatory).
- Typically it performs operations required to initialize the members (non static variable) of a class before the method and fields are invoked.

### Characteristics of Constructor:-

1. Its main purpose is to initialize instance variable of an object.
2. Reuse code with the help of constructor chaining.
3. It supports inheritance by initializing the members of super class using 'super()' statement.
4. It provides flexibility in object creation by performing constructor overloading.

### Syntax for creating constructor:-

```
class class_name
{
    class_name()
    {
        //statements to initialize an object and perform constructor chaining
    }
}
```

Constructor

### Rules for creating a constructor:-

1. A constructor name must be same as class name.
2. A constructor doesn't contains any return type.
3. We can use any access modifiers such as public, protected, default, private (depends upon the type of constructor).
4. We cannot use any non access modifiers such as abstract, final, synchronized, etc.
5. It can accept any input data as it contains formal arguments.

### There are four types of constructors in java:-

1. Default constructor
2. No argument constructor
3. Parameterized constructor
4. Copy constructor

### Note:-

- Every class in java contains a constructor.
  (It might be provided by compiler or programmer).
- Every constructor body must contain first statement as super() statement or this() statement.
- **super()** :-
  - It used to invoke the parent class constructor.
- **this()** :-
  - It is used to invoke the current class constructor.
- If a programmer fails to add super() statement or this() statement then compiler will add a super() statement.
- If a class contains only one constructor in it we cannot use this() statement.

## 1. Default constructor:-

- The constructor which is added by the compiler is called as default constructor.
- If a programmer fails to add any constructor then only compiler will add default constructor.
- Default constructor first statement is always super() statement.
- It inherits the access modifier from the top level class (outer class).
- We cannot declare a top level class as a private and protected so a default constructor cannot be private and protected.
- super() statement invokes the parent class no argument constructor.
- e.g.

```
class Demo
{
    Demo()
    {
        super();
    }
}
```

Added by the compiler

Invokes the parent class constructor

e.g.

```
class Suresh
{
    Suresh()
    {
        super();
    }
}
class Ramesh extends Suresh
{
    Ramesh()
    {
        super();
    }
}
class ConstructorExample
{
    public static void main(String[] args)
    {
        Ramesh obj = new Ramesh();
    }
}
```

Invokes Object class constructor

Invokes Suresh class constructor

Invokes Ramesh class constructor

## 2. No argument constructor:-

- A constructor which does not have any formal arguments is called as an no argument constructor.
- It is programmer created constructor which does not accepts any parameter.
- e.g.

```
class Demo
{
    Demo()
    {
            super();
            System.out.println(“No argument constructor”);
    }
    public static void main(String[] args)
    {
            Demo obj = new Demo();
    }
}
```

**Output:-**
No argument constructor

- **A no argument constructor body can have either a super() statement or this() statement.**
- **e.g.**

```
class Demo{
    Demo(){
            this(10);
            System.out.println(“No argument constructor”);
    }
    Demo(int a){
            super();
            System.out.println(“int argument constructor”);
    }
    public static void main(String[] args){
            Demo obj = new Demo();
    }
}
```

- **No argument constructor can have any access modifier such as public, protected, private, default.**
- **e.g.**

```
class Demo{
    public Demo(){
            super();
            System.out.println("public access modifier");
    }
    protected Demo(){
            super();
            System.out.println("protected access modifier");
    }
    Demo(){
            super();
            System.out.println("default access modifier");
    }
    private Demo(){
            super();
            System.out.println("private access modifier");
    }
}
```

- **e.g.**

```
class Parent
{
    Parent()
    {
        System.out.println("No argument constructor");
    }
    Parent(int a)
    {
        System.out.println("int argument constructor");
    }
}
class Child extends Parent
{
    Child()
    {
        super(10);      //invokes parent class int argument constructor
        super();        //invokes parent class no argument constructor
        System.out.println("no argument constructor child");
    }
}
class ConstructorExample
{
    public static void main(String[] args)
    {
        Child obj = new Child();
    }
}
```

# Constructor Chaining

```
class Demo
{
    Demo()
    {
        this(10);
        System.out.println
        ("Demo()");
    }
    Demo(int a)
    {
        super();
        System.out.println("Demo (int a)");
    }
}
```

```
class Demo1 extends Demo
{
    Demo1()
    {
        this(10,"Hello");
        System.out.println
        ("Demo1");
    }
    Demo1(int a,String b)
    {
        super();
        System.out.println
        ("Demo1(int a,String b)");
    }
}
```

```
class Driver
{
    public static void main(String[] args)
    {
        Demo1 obj = new Demo1();
    }
}
```

Object Class Constructor

### 3. Parameterized Constructor:-

- A constructor which contains formal arguments is called as parameterized constructor.
- This constructor can accept any input parameters.
- e.g.

```
class Demo
{
    Demo(int a,int b)
    {
            System.out.println("Hello from argument constructor");
    }
}
class Driver
{
    public static void main(String[] args)
    {
            Demo obj1 = new Demo(10,20);
            Demo obj2 = new Demo(10);          //compile time error
            Demo obj3 = new Demo(10,20,30);    //compile time error
            Demo obj4 = new Demo();            //compile time error
    }
}
```

**e.g.**

```
class Student
{
    int sid;
    String sname;
    String graduation;
    Student(int sid,String sname,String graduation)
    {
            super();
            this.sid = sid;
            this.sname = sname;
            this.graduation = graduation;
    }
    void displayStudent()
    {
            System.out.println("Student Details");
            System.out.println("Student ID : "+sid);
            System.out.println("Student Name : "+sname);
            System.out.println("Graduation : "+graduation);
    }
}
class DriverStudent
{
    public static void main(String[] args)
    {
            Student obj1 = new Student(101,"Ramesh","CS");
            obj1.displayStudent();
            Student obj2 = new Student(102,"Suresh","IT");
            obj2.displayStudent();
    }
}
```

## ★ this keyword:-

- this keyword is a non static reference variable which holds the reference of current object which is under use.
- this keyword refers to the current object inside a method or a constructor.
- The most common use of this keyword is to eliminate the confusion between the class and local variable.
- It is used inside a constructor for initializing the non static members.
- e.g.

```
class Demo
{
  void m1()
  {
        System.out.println(this);
  }
}
class DemoDriver
{
  public static void main(String[] args)
  {
        Demo obj1 = new Demo();
        System.out.println("Obj1 = "+obj1);
        obj1.m1();
        Demo obj2 = new Demo();
        System.out.println("Obj2 = "+obj2);
        obj2.m1();
  }
}
```

- We can print this keyword, it will print the reference of current object under use.
- this keyword changes its reference based on the current object.

- **We can use this keyword only inside a non static context i.e. non static block, non static method and a constructor.**

```
class Demo
{
  String a = "Global Varibale";
  Demo()
  {
        this.a = "GLOBAL VARIABLE";
  }
  {
        System.out.println(this.a);
  }
  void m1()
  {
        System.out.println(this.a);
  }
  public static void main(String[] args)
  {
        System.out.println("hello");
        Demo obj1 = new Demo();
        obj1.m1();
        System.out.println(obj1.a);
  }
}
```

- this keyword is used to invoke the current class fields and methods.
- this() statement is used to call a current class constructor and it is used to achieve constructor chaining.
- this() statement and super() statement must be first statement inside a constructor body and we cannot use it anywhere else other than a constructor.
- e.g.

```
class Demo
{
        Demo(){
                System.out.println("Demo()");
        }
        void m1(){
                this();   //compile time error
        }
        public static void main(String[] args){
                Demo obj = new Demo();
                obj.m1();
        }
}
```

★ **Difference between method and constructor.**

| Method | Constructor |
|---|---|
| i. Method is used to expose the states of an object. | i. Constructor is used to initialize the non static member. |
| ii. Method name may or may not be same as class name. | ii. Constructor name must be same as class same. |
| iii. Methods has contains return type. | iii. Constructor doesn't contain any return type. |
| iv. Methods can have any non access modifier. | iv. Constructor does not have any non access modifier. |
| v. this() and super() can't be used from a method body. | v. this() and super() can be used in constructor body. |
| vi. In case if you are not defining any method the compiler will not provide any default method. | vi. If a programmer fails to add constructor compiler will provide a default constructor. |
| vii. We can achieve method recursion. | vii. We cannot achieve method recursion. |
| viii. Methods are inherited from one class to another class. | viii. Constructors cannot be inherited. |
| ix. We can override a method. | ix. We cant override a constructor. |

★ **Difference between default constructor and no argument constructor**

| Default constructor | No argument Constructor |
|---|---|
| i. Default compiler is provided by compiler. | i. No argument constructor is provided by programmer. |
| ii. Default constructor contains only super() statement. | ii. No argument constructor contains either super() and this() statement. |
| iii. Default constructor can have default and public modifier. | iii. No argument constructor can be public, private, protected, default. |
| iv. Default constructor inherit access modifier from class (top level class). | iv. No argument constructor can't inherit access modifier from top level class. |

# ★ Copy constructor:-

- A copy constructor is a constructor that initialize an object using another object of a same class.
- Its main purpose is to create a new instance i.e. a copy of an existing instance.
- If we want to create a copy of an object we should have an existing copy of a same class.
- We can use a copy constructor for creating a shallow copy or a deep copy it depends on the implementation.

★ **Creating a shallow copy using a copy constructor**

- A shallow copy refers to creating a new object that is a copy of an existing object.
- In this all the references of the existing objects attribute remains same for the new attributes of the copied object.
- In simple words existing object attribute (non primitive fields) and new object attribute (non primitive fields) shares the same reference.
- So if we try to make any modifications in any of the attribute it will be reflected in all the copies along with the existing (original) object.
- e.g.

```
class Address{
    String street;
    String landmark;
    int pincode;
    Address(String street,String landmark,int pincode){
            super();
            this.street = street;
            this.landmark = landmark;
            this.pincode = pincode;
    }
    void displayAddress()
    {
            System.out.println();
            System.out.println("Address Details");
            System.out.println("Street name : "+street);
            System.out.println("Landmark : "+landmark);
            System.out.println("Pincode : "+this.pincode);
    }
}
class User
{
    String username;
    String password;
    Address address;
    User(String username,String password,String street,String landmark,int pincode)
    {
            this.username = username;
            this.password = password;
            this.address = new Address(street,landmark,pincode);
    }
    User(User existing)
    {
            this.username = existing.username;
            this.password = existing.password;
            this.address = existing.address;
    }
    void diplayUser()
    {
            System.out.println("User Details");
            System.out.println("Username : "+username);
            System.out.println("Password : "+password);
    }
}
```

```
class Demo
{
    public static void main(String[] args)
    {
            User existing = new User("Ramesh","ramesh123","JM Road",
        "Opposite to HP Petro",411004);
            existing.diplayUser();
            existing.address.displayAddress();
            //copy of an obj.....invoking copy constructor
            User copy = new User(existing);
            copy.diplayUser();
            copy.address.displayAddress();
            copy.address.landmark = "Something";
            copy.address.displayAddress();
            System.out.println();
            System.out.println("Exisitng object below");
            existing.address.displayAddress();
            copy.username = "suresh";
            copy.diplayUser();
            existing.diplayUser();
    }
}
```

★ **Deep copy:-**

- Deep copy in java can be created using copy constructor.
- It creates a new copy of an existing object where all the attributes (non static variables) of the original object and a new objects attribute will have the different references.
- Because of which changes in the existing object will not be reflected in any other copy.
- Changes in copy (object) will not affect the (modify) existing object.
- e.g.

```
class Address
{
    String street;
    String landmark;
    int pincode;
    Address(String street,String landmark,int pincode)
    {
            super();
            this.street = street;
            this.landmark = landmark;
            this.pincode = pincode;
    }

    Address (Address existing)
    {
            this.street = existing.street;
            this.landmark = existing.landmark;
            this.pincode = existing.pincode;
    }

    void displayAddress()
    {
            System.out.println();
            System.out.println("Address Details");
            System.out.println("Street name : "+street);
            System.out.println("Landmark : "+landmark);
            System.out.println("Pincode : "+this.pincode);
    }
}
```

```
class User
{
    String username;
    String password;
    Address address;
    User(String username,String password,String street,String landmark,int pincode)
    {
            this.username = username;
            this.password = password;
            this.address = new Address(street,landmark,pincode);
    }
    User(User existing)
    {
            this.username = existing.username;
            this.password = existing.password;
            this.address = new Address (existing.address);
    }
    void diplayUser()
    {
            System.out.println("User Details");
            System.out.println("Username : "+username);
            System.out.println("Password : "+password);
    }
}
class Demo
{
    public static void main(String[] args)
     {
            User existing = new User("Ramesh","ramesh123","JM Road",
                                                    "Opposite to HP Petro",411004);
            existing.diplayUser();
            existing.address.displayAddress();
            System.out.println();
            //copy of an obj.....invoking copy constructor
            User copy = new User(existing);
            copy.diplayUser();
            copy.address.displayAddress();
            System.out.println();
            System.out.println("Changing copy object members data");
            copy.address.landmark = "Something";
            copy.address.displayAddress();
            copy.diplayUser();
            System.out.println();
            System.out.println("Exisitng object below");
            existing.address.displayAddress();
            System.out.println();
            copy.username = "suresh";
            copy.diplayUser();
            existing.address.pincode = 0;
            existing.address.displayAddress();
            existing.diplayUser();
    }
}
```

## Assignments

**e.g.1**

```
class Hospital
{
   int pid;
   String pname;
   String disease;
   String wardno;
   String bedno;

   Hospital(int pid, String pname, String disease, String wardno, String bedno)
   {
      this.pid = pid;
      this.pname = pname;
      this.disease = disease;
      this.wardno = wardno;
      this.bedno = bedno;
   }

   void displayPatient()
   {
      System.out.println("Patient Details ");
      System.out.println(" ID : " + pid);
      System.out.println("Patient Name : " + pname);
      System.out.println("Ward no. : " + wardno);
      System.out.println("Bed no. : " + bedno);
      System.out.println("Disease : " + disease);
   }
}

class HospitalDriver
{
   public static void main(String[] args)
   {
      Hospital obj1 = new Hospital(101, "Akshay", "Maleria", "4", "B-10");
      obj1.displayPatient();

      Hospital obj2 = new Hospital(102, "Tanmay", "Viral Infection", "5", "C-2");
      obj2.displayPatient();

      Hospital obj3 = new Hospital(103, "Ramesh", "HIV", "7", "A-1");
      obj3.displayPatient();

      Hospital obj4 = new Hospital(104, "Suresh", "Cancer", "2", "B-15");
      obj4.displayPatient();

      Hospital obj5 = new Hospital(105, "Vishal", "Kidney Stone", "5", "A-5");
      obj5.displayPatient();
   }
}
```

**e.g. 2**

```
class Car
{
	int cid;
	String ccompany;
	String cmodel;
	String ctype;
	double cprice;
	Car(int cid,String ccompany,String cmodel,String ctype,double cprice)
	{
		this.cid = cid;
		this.ccompany = ccompany;
		this.cmodel = cmodel;
		this.ctype = ctype;
		this.cprice = cprice;
	}
	void displayCarDetails()
	{
		System.out.println("Car Information");
		System.out.println("Car id : "+cid);
		System.out.println("Car Company : "+ccompany);
		System.out.println("Car model : "+cmodel);
		System.out.println("Car type : "+ctype);
		System.out.println("Car price : "+cprice);
	}
}
class CarDriver
{
	public static void main(String[] args)
	{
		Car obj1 = new Car(101,"Mahindra","Scorpio","SUV",2000000);
		obj1.displayCarDetails();
		Car obj2 = new Car(102,"Mahindra","Bolero","SUV",1800000);
		obj2.displayCarDetails();
		Car obj3 = new Car(103,"Tata","Harrier","SUV",2600000);
		obj3.displayCarDetails();
		Car obj4 = new Car(104,"Tata","Safari","SUV",2500000);
		obj4.displayCarDetails();
		Car obj5 = new Car(105,"Mitsubishi","Pajerao-Sports","SUV",2700000);
		obj5.displayCarDetails();
	}
}
```

# ★ Factory design pattern:-

- The factory design pattern is an approach to create objects in an efficient manner.
- It is used to enhance an application.
- There are several types of factory design patterns such as,
  1. Simple factory
  2. Factory method
  3. Abstract method

# ★ Singleton class:-

- A singleton class in java is a class of which we can create any one instance.
- We use singleton class when we want to create one instance and we want to access it globally throughout all the classes.
- We use it for connecting with servers, streams, database, connections, etc.
- **Steps to create a singleton class:-**
  1. Create a static private data member of that class.
  2. Declare the constructor as private.
  3. Create a public static method
     This method will help to validate that we are creating the instance first time or not.
- e.g.

```
class Singleton
{
    static  private Singleton obj = null;
    Singleton()
    {
            super();
    }
    public static Signleton getInstance()
    {
            if(obj==null)
            {
                    obj = new Singleton();
            }
            return obj;
    }
}
class SignletonDriver
{
    public static void main(String[] args)
    {
            Singleton obj1 = Singleton.getInstance();
            System.out.println(obj1);
            Signleton obj2 = Singleton.getInstance();
            System.out.println(obj2);
    }
}
```

# ★ Object Oriented Programming :-

- Object oriented programming is the core of java programming language.
- We create objects using classes, manipulate them to get an expected result.
- OOP's can be characterized as data controlling for accessing the members.
- There are various terms / concepts in OOP's:-
    1. Class
    2. Object
    3. Encapsulation
    4. Association (relationship)
    5. Composition
    6. Aggregation
    7. Is – a – relationship (inheritance)
    8. Polymorphism
    9. Abstraction

## ★ Advantages of OOP's:-

1. **Reusability:-**
    - When we say reusability it means write once and use multiple times.
    - In this we use a same functionality multiple times rather than creating it again and again.

2. **Avoid data redundancy (repetition) :-**
    - It is one of the main advantage of OOP's.
    - This condition occurs when we create multiple modules where we need a same functionality at multiple places.
    - So we can create a common class definition (module) and that functionality will be shared among all.

3. **Security:-**
    - Data hiding and abstraction are used to filter out limited exposure, which means we are providing only the necessary data to view because we have to maintain the security.

4. **Easy troubleshooting :-**
    - Everything in OOP's is an object, so we can use this object to interchange, modify, reuse some functionalities to meet the users need.
    - So if we have to find a problem it becomes more easy as the classes are integrated to each other and tracking becomes easy.
    - So we can find the root cause of a problem and fix it.

## ★ There are four main principles of OOP's

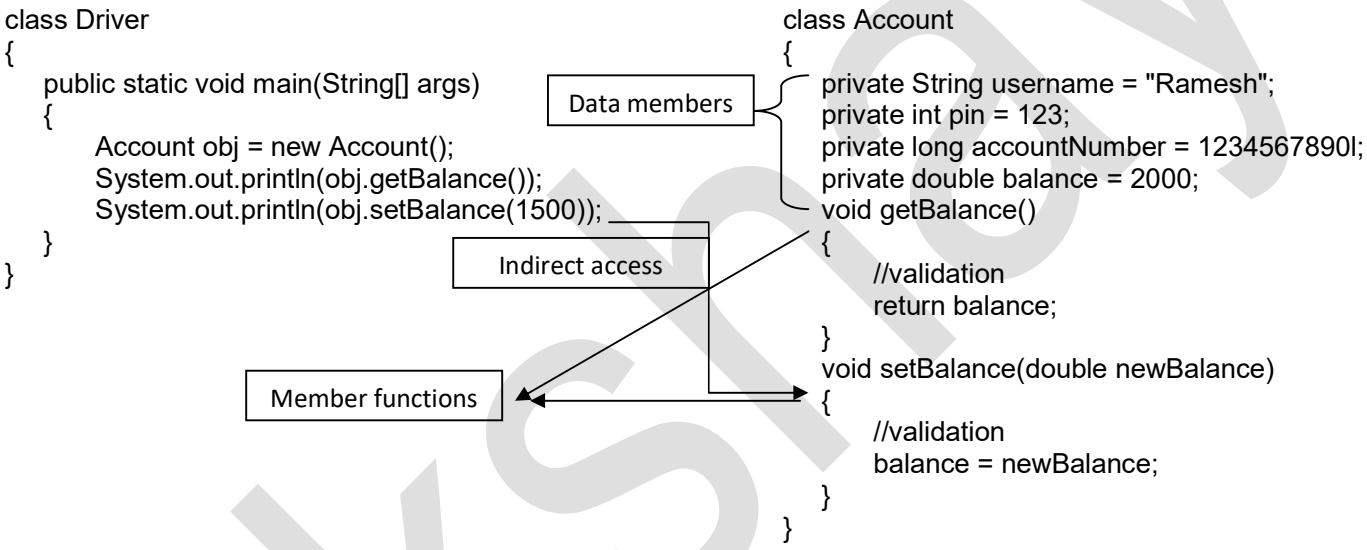
1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

# 1. Encapsulation:-

- The process of binding or wrapping the data members and member functions together as a single unit is known as encapsulation.
- The process of binding the attributes and behaviours of an object together as a single unit is called as encapsulation.
- E.g.:- AC, Car, Human body, Pen, etc.

**Data binding:-**

- The process of restricting the direct access to the data members and providing an indirect and secure access to the data members with help of getters() and setters() method is known as data binding.




**e.g.**

```
class UserAccount
{
        private String username;
        private int pin;
        private double bal;
        private long adhar;

        UserAccount(String username,int pin,double bal,long adhar)
        {
                this.username = username;
                this.pin = pin;
                this.bal = bal;
                this.adhar = adhar;
        }
        public String getUsername()
        {
                //validation
                return username;
        }
```

```
	public void setUsername(String newUsername)
	{
		username = newUsername;
	}
	public int getPin()
	{
		//validation
		return pin;
	}
	public void setPin(int newPin)
	{
		//validation
		pin = newPin;
	}
	public double getBalance()
	{
		//validation
		return bal;
	}
	public void setBalance(int newBalance)
	{
		//validaiton
		bal = newBalance;
	}
	public long getAdhar()
	{
		//validation
		return adhar;
	}
}
class EncapDriver
{
	public static void main(String[] args)
	{
		UserAccount obj = new UserAccount("Akshay",123,20000,123456789);
		System.out.println(obj.getUsername());
		System.out.println(obj.getPin());
		System.out.println(obj.getBalance());
		System.out.println(obj.getAdhar());
		System.out.println();
		obj.setUsername("Tanmay");
		obj.setPin(7809);
		obj.setBalance(1000);
		System.out.println(obj.getUsername());
		System.out.println(obj.getPin());
		System.out.println(obj.getBalance());
		System.out.println(obj.getAdhar());
	}
}
```

★ **Steps to achieve data hiding:-**

1. Declare the data member as private (so we cannot access them outside its class).
2. Provide public getters() and setters() method to view only and write only perform validation.

★ **Private modifier:-**

- private is a keyword and access modifier in java.
- It can be used with various java members such as innerclass, variable, method, constructor, etc.
- private member cannot be accessed outside its class.

**Note:-**

**We cannot make top level class (outer class) as private and protected.**

★ **Why do we need encapsulation?**

- As the data members are private because of security purpose we cannot access them outside its class.
- To access and modify this data members we create some helper methods.
- We have to bind the behaviours and states together.
- So it keeps the code cleaner and easy to read.

★ **What is binding?**

Merging all the various components as one unit.

## Program for Cab / Taxi / Uber

```
import java.util.*;
class CabDriver
{
  static int driverId = 123;
  private int id;
  private String name;   //get set
  private long contact;   //get set
  private String type;     //get
  private String car;      //get
  private long account;  //get set
  private String status = "Available";
  CabDriver(String name,long contact,String type,String car,long account)
  {
        super();
        this.name = name;
        this.contact = contact;
        this.type = type;
        this.car = car;
        this.account = account;
        this.id = driverId++;
  }
  public String getName()
  {
        return this.name;
  }
  public void setName(String newName)
  {
        this.name = newName;
  }
  public long getContact()
  {
        return this.contact;
  }
```

```
    public void setContact(long newContact)
    {
            this.contact = newContact;
    }
    public String getType()
    {
            return this.type;
    }
    public String getCar()
    {
            return this.car;
    }
    public long getAccount()
    {
            return this.account;
    }
    public void setAccount(long newAccount)
    {
            this.account = newAccount;
    }
    public String getStatus()
    {
            return this.status;
    }
    public void setStatus(String newStatus)
    {
            this.status = newStatus;
    }
    public int getId()
    {
            return this.id;
    }

    public void displayCabDriver()
    {
            System.out.println();
            System.out.println("*** CAB DETAILS ***");
            System.out.println("Booking ID : "+this.id);
            System.out.println("Driver name : "+this.getName());
            System.out.println("Contact : "+this.getContact());
            System.out.println("Type of car : "+this.getType());
            System.out.println("Car number : "+this.getCar());
            System.out.println("Account number : "+this.getAccount());
            System.out.println("Status : "+this.getStatus());
    }

    public void displayCabDriverAfterBooking(){
            System.out.println();
            System.out.println("*** CAB DETAILS ***");
            System.out.println("Booking ID : "+this.id);
            System.out.println("Driver name : "+this.getName());
            System.out.println("Contact : "+this.getContact());
            System.out.println("Type of car : "+this.getType());
            System.out.println("Car number : "+this.getCar());
            System.out.println("Account number : "+this.getAccount());
            System.out.println("Status : "+this.getStatus());
    }
}
```

```
class Passenger
{
  String name;   //get set
  String start;     //get set
  String end;               //get set
  long contact;   //get set
  int noPass;     //get set

  Passenger(String name,String start,String end,long contact,int noPass)
  {
          super();
          this.name = name;
          this.start = start;
          this.end = end;
          this.contact = contact;
          this.noPass = noPass;
  }
  public String getName()
  {
          return this.name;
  }
  public void setName(String newName)
  {
          this.name = newName;
  }
  public String getStart()
  {
          return this.start;
  }
  public void setStart(String newStart)
  {
          this.start = newStart;
  }
  public String getEnd()
  {
          return this.end;
  }
  public void setEnd(String newEnd)
  {
          this.end = newEnd;
  }
  public long getContact()
  {
          return this.contact;
  }
  public void setContact(long newContact)
  {
          this.contact = newContact;
  }
  public int getNoPass(){
          return this.noPass;
  }
  public void setNoPass(int newNoPass){
          this.noPass = newNoPass;
  }
}
```

```
class SavariDriver
{
  static ArrayList<CabDriver> listcab = new ArrayList<>();
  public static void main(String[] args)
  {
        Scanner sc = new Scanner(System.in);
        CabDriver obj1 = new CabDriver("Ramesh",9876543210l,"Sedan",
                                                        "MH12AA1234",123412341234l);
        CabDriver obj2 = new CabDriver("Suresh",8876543210l,"Hatchback",
                                                         "MH12AA5678",432112341234l);
        CabDriver obj3 = new CabDriver("Mahesh",7876543210l,"SUV",
                                                         "MH12AA3456",789012341234l);
        CabDriver obj4 = new CabDriver("Mukesh",6876543210l,"XUV",
                                                        "MH12AA8967",456712341234l);

        listcab.add(obj1);
        listcab.add(obj2);
        listcab.add(obj3);
        listcab.add(obj4);
        String start = null;
        String end = null;
        String name = null;
        long contact = 0;
        int noPass = 0;

        for (; ; )
        {
              System.out.println();
              System.out.println("Welcome to SAVARI");
              System.out.println();
              System.out.println("***Book a Ride***");
              System.out.print("Enter a name : ");
              sc.nextLine();
              name = sc.nextLine();
              System.out.println("Enter start dest : ");
              start = sc.nextLine();
              System.out.println("Enter end dest : ");
              end = sc.nextLine();
              System.out.println("Enter contact : ");
              contact = sc.nextLong();
              System.out.println("No of pass : ");
              noPass = sc.nextInt();
              sc.nextLine();

              Passenger pass = new Passenger(name,start,end,contact,noPass);
              System.out.println();
              for(CabDriver i : listcab)
              {
                    if(i.getStatus().equals("Available"))
                    {
                          i.displayCabDriver();
                    }
              }

              System.out.println();
              System.out.println("Enter the booking id : ");
              int bookid = sc.nextInt();

              for(CabDriver i : listcab)
              {
```

```
                        if(bookid==i.getId())
                        {
                                i.setStatus("Occupied");
                                i.displayCabDriverAfterBooking();
                                System.out.println();
                                System.out.println("Your Ride has been booked.");
                        }
                }
        }
  }
}
```

**Assignment**
**For above program include fair module**
**Use**
**fair = km*rate;**
Enter the distance to reach
You have to calculate duration also and kilometres

**Note:-**

- getters() and setters() method must be public.
- getter() method helps to fetch the data, so it contains a return type (specific data type).
- setter() method is used to set the value so it accepts an input parameter and doesn't returns anything so return type is void.

# ★ Relationship :-

- Relationship is an association between two or more objects (entity).
- In other words in relationship an object of one class has a reference of another class.
- Relationship is classified into two types,
  1. Has a relationship
  2. Is a relationship

## 1. Has a relationship :-

- Its an association between two or more objects such that they are dependent on each other is known as has a relationship.
- When an object of one class is created as a data member inside another object (class) is called as has a relationship.
- There are two types of has a relationship,
  i. Composition
  ii. Aggregation

### i. Composition:-

- Its an association between two objects where one cannot exists without another.
- e.g.
  human and heart,
  water and fish,
  human and oxygen,
  car and engine,
  software and hardware,
  human and brain,
  human and water,
  plant and water.

**Program :-**

```
class Car
{
        String brand;
        String model;
        String type;
        int passCap;
        String color;
        double price;
        // 1 Engine obj = new Engine("Diseal",302,4,7);
        // 2 Engine engine;
        Engine engine;

        Car(String model,String brand,String type,int passCap,String color,double price,
                                                                        Engine engine)
        {
                super();
                this.brand = brand;
                this.model = model;
                this.type = type;
                this.passCap = passCap;
                this.color = color;
                this.price = price;
                //2 this.engine = new Engine("Diseal",302,4,7);
                this.engine = engine;
        }
        public  void displayCar()
        {
```

```
                System.out.println("Car details");
                System.out.println("Brand : "+this.brand);
                System.out.println("Model : "+this.model);
                System.out.println("Type : "+this.type);
                System.out.println("Passenger capacity : "+this.passCap);
                System.out.println("Color : "+this.color);
                System.out.println("Price : "+this.price);
        }
}
class Engine
{
        String typeEngine;
        double bhp;
        int noPiston;
        double oilCapacity;

        Engine(String typeEngine,double bhp,int noPiston,double oilCapacity)
        {
                super();
                this.typeEngine = typeEngine;
                this.bhp = bhp;
                this.noPiston = noPiston;
                this.oilCapacity = oilCapacity;
        }

        public void displayEngine()
        {
                System.out.println("Engine details");
                System.out.println("Engine type : "+this.typeEngine);
                System.out.println("BHP : "+this.bhp);
                System.out.println("No of piston : "+this.noPiston);
                System.out.println("Oil capacity : "+this.oilCapacity);
        }
}
class HasARelationship
{
        public static void main(String[] args)
        {
                Car car = new Car("Scorpio","Mahindra","SUV",7,"White",1800000,
                                                                (new Engine("Diseal",302,4,7)));
                car.displayCar();
                car.engine.displayEngine();
        }
}
```

**Note :-**

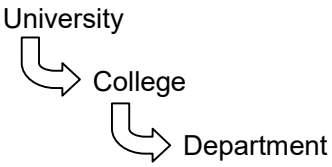
- If we try to invoke any variables or methods from a class of which object is not yet created, we get a runtime exception "**NullPointerException**".
- It will not create any compile time error.

**Program 2:-**

```
class Librarian
{
	String name;
	int employeeId;
	int yearsOfExperience;
	String qualification;
	String shift;
	Librarian(String name, int employeeId, int yearsOfExperience, String qualification, String shift)
	{
		this.name = name;
		this.employeeId = employeeId;
		this.yearsOfExperience = yearsOfExperience;
		this.qualification = qualification;
		this.shift = shift;
	}

	void displayLibrarian() {
		System.out.println("Librarian Name: " + name);
		System.out.println("Employee ID: " + employeeId);
		System.out.println("Years of Experience: " + yearsOfExperience);
		System.out.println("Qualification: " + qualification);
		System.out.println("Shift: " + shift);
	}
}

class Library
{
	String libraryName;
	String location;
	int numberOfBooks;
	Librarian librarian;

	Library(String libraryName, String location, int numberOfBooks, Librarian librarian)
	{
		this.libraryName = libraryName;
		this.location = location;
		this.numberOfBooks = numberOfBooks;
		this.librarian = librarian;
	}

	void displayLibrary() {
		System.out.println("Library Name: " + libraryName);
		System.out.println("Location: " + location);
		System.out.println("Number of Books: " + numberOfBooks);
	}
}

class HasARelationship1
{
	public static void main(String[] args) {
		Librarian librarian = new Librarian("Alice Johnson", 1012, 15,
                                                    "Master of Library Science", "Morning");
		Library library = new Library("City Central Library", "Downtown", 50000, librarian);
		library.displayLibrary();
		library.librarian.displayLibrarian();
   }
}
```

### ii. Aggregation :-

- It is an association between two or more object where objects are not dependent on each other and survive (exists) individually.
- It's an unidirectional association i.e. one way.
- e.g.




University can have different colleges but a college cannot have different universities.

e.g.
class and student,
book and bag,
human and chair,
mobile and mobilecover,
computer and pendrive,
human and shoes,
water and bottle

**Aggregation Example (WhiteBoard, Marker, Duster):-**

```
class WhiteBoard
{
	String brand;
	double price;
	String dim;
	Marker marker;
	Duster duster;

	WhiteBoard(String brand,double price,String dim)
	{
		super();
		this.brand = brand;
		this.price = price;
		this.dim = dim;
	}
	public void displayWhiteBoard()
	{
		System.out.println("WhiteBoard : [Brand : "+brand+", Price : "+price+
					", Dimension : "+dim+"]");
	}
	public void createMarker()
	{
		marker = new Marker("Camlin","Red","WhiteBorad Marker",30);
	}
	public void createDuster()
	{
		duster = new Duster("Polo","Black","Plastic","Rectangular",50);
	}
}
class Marker
{
	String brand;
	String color;
	String type;
	double price;
```

```
	Marker(String brand,String color,String type,double price)
	{
		super();
		this.brand = brand;
		this.color = color;
		this.type = type;
		this.price = price;
	}
	public void displayMarker()
	{
		System.out.println("Marker : [Brand = "+brand+", Color = "+color+", Type : "+type+
					", Price = "+price+" ]");
	}
}
class Duster
{
	String brand;
	String color;
	String material;
	String shape;
	double price;

	Duster(String brand,String color,String material,String shape,double price)
	{
		super();
		this.brand  = brand;
		this.color = color;
		this.material = material;
		this.shape = shape;
		this.price = price;
	}
	public void displayDuster()
	{
		System.out.println("Duster : [Brand = "+brand+", Color = "+color+", Material = "+material+
							", Shape = "+shape+", Price = "+price+"]");
	}
}
class WhiteBoardDriver
{
	public static void main(String[] args)
	{
		WhiteBoard board =  new WhiteBoard("Cello",12000,"3m X  10m");
		board.displayWhiteBoard();
		board.createMarker();
		board.marker.displayMarker();
		board.createDuster();
		board.duster.displayDuster();
	}
}
```

**Composition and aggregation example Fish Tank (FishTank, Water, Fish, FishFood, Plants, Stone, Filter, Light):-**

```
//1
class Fish{
	String fishBreed;
	String fishType;
	double fishPrice;
	String fishColor;
	Fish(String fishBreed,String fishType,double fishPrice,String fishColor){
		super();
		this.fishBreed = fishBreed;
		this.fishType = fishType;
		this.fishPrice = fishPrice;
		this.fishColor = fishColor;
	}
	void displayFish(){
		System.out.println("Fish : [breed = "+fishBreed+", Type = "+fishType
							+", Price ="+fishPrice+", Color ="+fishColor+"]");
	}
}
//2
class FishTank{
	String material;
	String shape;
	String dimension;
	double price;
	double capacity;

	Water water;
	Stone stone;
	FishFood food;
	Plant plant;
	Light light;
	Filter filter;
	Fish fish;

	FishTank(String material,String shape,String dimension,double price,double capacity){
		super();
		this.material = material;
		this.shape = shape;
		this.dimension = dimension;
		this.price = price;
		this.capacity = capacity;
	}
	 void displayFishTank(){
		System.out.println("FishTank : [Material = "+material+", Shape = "+shape
				+", Dimension ="+dimension+", Price ="+price+", Capacity = "+capacity+"]");
	 }
	 public void addFish(){
		if(water !=null)
			fish = new Fish("Gold Fish","Fresh Water Fish",150,"Gold");
		else
			System.out.println("Add water first.");
	 }
	 public void addWater(){
		water = new Water("Distilled water",10,7);
	 }
	 public void addFood(){
		food = new FishFood("Chitale",100,120);
	 }
```

```
	public void addPlant(){
		plant = new Plant("Java moss",100,"Green",10);
	}
	public void addLight(){
		light = new Light("Bajaj","Yellow",50,5);
	}
	public void addFilter(){
		filter = new Filter("Bajaj",1500,15,"3 l");
	}
	public void addStone(){
		stone = new Stone("Circlular",100,5,5);
	}
}
//3
class Water {
	String type;
	double liter;
	int ph;
	Water(String type,double liter,int ph){
		super();
		this.type = type;
		this.liter = liter;
		this.ph = ph;
	}
	void displayWater(){
		System.out.println("Water : [Type = "+type+", Liter = "+liter+", ph = "+ph+"]");
	}
}
//4
class FishFood
{
	String brand;
	double quantity;
	double price;
	FishFood(String brand,double quantity,double price){
		super();
		this.brand = brand;
		this.quantity = quantity;
		this.price = price;
	}
	 void displaYFishFood(){
		System.out.println("FishFood : [Brand = "+brand+", Quantity = "+quantity+",
                                                                price = "+price+"]");
	 }
}
//5
class Plant{
	String type;
	double price;
	String color;
	int quantity;
	Plant(String type,double price,String color,int quantity){
		super();
		this.type = type;
		this.price = price;
		this.color = color;
		this.quantity = quantity;
	}
```

```
	void displayPlant(){
		System.out.println("Plant : [Type = "+type+", Price = "+price+", Color = "+color+
							", Quantity : "+quantity+"]");
	}
}
//6
class Stone
{
	String shape;
	int quantity;
	double price;
	double weight;
	Stone(String shape,int quantity,double price,double weight){
		super();
		this.shape = shape;
		this.quantity = quantity;
		this.price = price;
		this.weight = weight;
	}
	void displayStone(){
		System.out.println("Stone : [Shape = "+shape+", Quantity = "+quantity+", Price = "+price+
							"rs, Weight : "+weight+" gm]");
	}
}
//7
class Light{
	String brand;
	String color;
	double price;
	int watt;
	Light(String brand,String color,double price,int watt){
		super();
		this.brand = brand;
		this.color = color;
		this.price = price;
		this.watt = watt;
	}
	void displayLight(){
		System.out.println("Light : [Brand = "+brand+", Color = "+color+
							", Price = "+price+", Watt = "+watt+"]");
	}
}
//8
class Filter{
	String brand;
	double price;
	double watt;
	String cap;
	Filter(String brand,double price,double watt,String cap){
		this.brand = brand;
		this.price = price;
		this.watt = watt;
		this.cap = cap;
	}
	void displayFilter(){
		System.out.println("Filter : [Brand = "+brand+", Price = "+price+
							", Watt = "+watt+", Capacity = "+cap+"]");
	}
}
```

```
class FishRelationship{
        public static void main(String[] args){
                FishTank tank = new FishTank("Glass","Rectangular","30h X 15w X50l",1000,12);
                tank.displayFishTank();
                tank.addWater();
                tank.water.displayWater();
                tank.addFish();
                tank.fish.displayFish();
                tank.addFood();
                tank.food.displaYFishFood();
                tank.addPlant();
                tank.plant.displayPlant();
                tank.addLight();
                tank.light.displayLight();
                tank.addFilter();
                tank.filter.displayFilter();
                tank.addStone();
                tank.stone.displayStone();
        }
}
```

**Note:-**

**For composition :-**

- Composition is also known as tight binding / early instantiation / early binding.
- In this we create an object of one class as a data member inside another class implicitly.
- E.g. Car and engine.
- Whenever a user creates an object of a Car automatically (implicitly) an object of engine is created, therefore if we are not creating an object of Car, engine will not be created.

**For aggregation :-**

- Aggregation is a loose coupling where one object can exists without another its also known as loose coupling / lazy instantiation / late binding.
- In this the instance of the dependent object is not created implicitly.
- Instead of that we create a method and the method that helps to create an object of dependent object.
- This method is known as a helper method.

# 2. Inheritance :-

- Inheritance in java is a mechanism in which one object can acquire all the properties and behaviours from a parent object.
- The idea behind inheritance in java is that we can create new classes that are developed upon existing classes.
- When we inherit a class we can reuse the methods and fields of the parent class, more ever we can also add new members for that current class.

**Advantages of Inheritance:-**

1. Reusability of functionalities
2. Less time consumption
3. Avoids data redundancy
4. Improves performance

**Terminologies used in inheritance:-**

1. **Class:-**
   - A class is a blueprint which objects are created.
2. **Child class:-**
   - A class which inherits an another class is known as child class / sub class / derived class / extended class.
3. **Parent class:-**
   - A class from where a sub class inherits the functionalities is known as parent class / super class / base class.
4. **Reusability:-**
   - In simple words reusability means Write Once and Use Manytimes.

**Types of inheritance in java:-**

1. Single level inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Hybrid inheritance
5. Multiple inheritance
6. Cyclic inheritance

There are two keywords which we can use to achieve inheritance
1. extends
2. implements

1. **extends:-**
   - extends is a keyword in java which is used to achieve inheritance between classes and interfaces.
   - It is used to achieve is a relationship between classes and interfaces (i.e. parent child relationship).
   - e.g.

```
class Ramesh
{
        String str = "Ramesh class";
        public void m1(){
                System.out.println("m1() from Ramesh class");
        }
}
class Suresh extends Ramesh
{
```

```
        String str1 = "Suresh class";
        public void m2()
        {
            System.out.println("m2() from Suresh class");
        }
}
class InheritanceExample
{
        public static void main(String[] args)
        {
            Suresh obj = new Suresh();
            System.out.println(obj.str1);
            obj.m2();
            System.out.println(obj.str);
            obj.m1();
        }
}
```

Output:-
Suresh class
m2() from Suresh class
Ramesh class
m1() from Ramesh class

- When we use extends keyword a class which uses extends keyword becomes a subclass and the class after extends keyword is a superclass.
- When we create an object of the subclass all the data members and member functions of its super class are loaded inside the object along with its data member and member functions.
- So using one object reference we can access them.
- e.g.

```
class Parent1{

}
class Parent2{

}
class Child extends Parent1,Parent2{

}
class Driver{
    public static void main(String[] args) {

    }
}
```

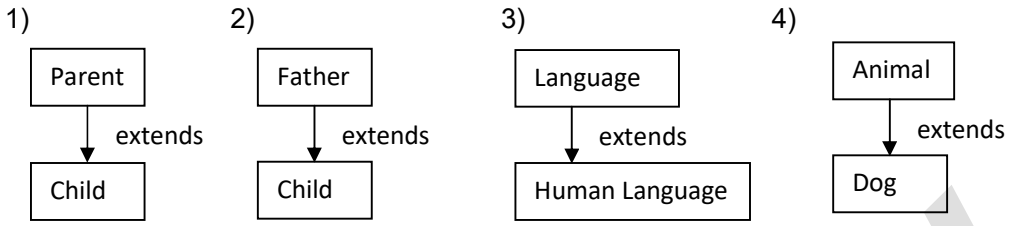
**We can extend interface with multiple interfaces (i.e. nothing but multiple inheritance)**
**E.g.**

```
interface Parent1{

}
interface Parent2{

}
interface Child extends Parent1,Parent2{

}
```

**Note:-**
**We cannot extend an interface using a class.**

## 1. Single level inheritance:-

- Inheritance achieved on one level where one super class has one sub class is known as single level inheritance.
- e.g.




**Example 1 :- country and states**

```
class India
{
	String president;
	String primeMinister;
	String capital;
	int states;
	int union;
	String lang;

	India(String president,String primeMinister,String capital,int states,int union,String lang)
	{
		super();//object class constructor call
		this.president = president;
		this.primeMinister = primeMinister;
		this.capital = capital;
		this.states = states;
		this.union = union;
		this.lang = lang;
	}
	public void displayIndia()
	{
		System.out.println("India details.");
		System.out.println("President : "+this.president);
		System.out.println("Prime minister : "+this.primeMinister);
		System.out.println("Capital : "+this.capital);
		System.out.println("States = "+this.states);
		System.out.println("Union = "+this.union);
		System.out.println("Language = "+this.lang);
	}
}
class Maharashtra extends India
{
	String cm;
	String capital;
	int districts;
	String lan;
	long pop;
	double area;

	Maharashtra(String cm,String capital, String lan,int districts, long pop, double area,
			String president,String primeMinister,String capital1,int states,int union,String lang)
	{
		super(president,primeMinister,capital1,states,union,lang);
		this.cm = cm;
		this.capital = capital;
```

```
		this.districts = districts;
		this.lan = lan;
		this.pop = pop;
		this.area = area;
	}
	public void displayMaharashtra()
	{
		System.out.println("Maharashtra Details.");
		System.out.println("Chief minister : "+this.cm);
		System.out.println("Capital : "+this.capital);
		System.out.println("Language : "+this.lan);
		System.out.println("Number of districts : "+this.districts);
		System.out.println("Population : "+this.pop);
		System.out.println("Area : "+this.area+" km^2");
	}
}
class SignleLevelInheritanceExample1
{
	public static void main(String[] args)
	{
		Maharashtra obj = new Maharashtra("Eknath Shinde","Mumbai","Marathi",36,112374333,
				307713,"Droupadi Murmu","Narendra Modi","Delhi",8,29,"Hindi");
		obj.displayIndia();
		System.out.println();
		obj.displayMaharashtra();
	}
}
```

**Example 2 :- vehicle and car**

```
class Vehicle
{
	String model;
	int year;
	String color;
	double mileage;
	String type;
	Vehicle(String model, int year, String color, double mileage,String type)
	{
		super();
		this.model = model;
		this.year = year;
		this.color = color;
		this.mileage = mileage;
		this.type = type;
	}
	public void displayVehicle(){
		System.out.println("Model = "+this.model);
		System.out.println("Year = "+this.year);
		System.out.println("Color = "+this.color);
		System.out.println("Mileage = "+this.mileage);
		System.out.println("Type = "+this.type);
	}

}
class Car extends Vehicle
{
	int noDoor;
	String fuelType;
	boolean isElectric;
	String transmission;
```

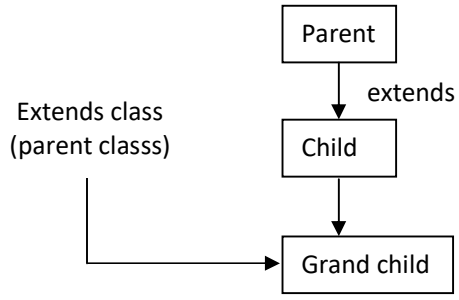
```
	int passCapacity;
	Car(int noDoor, String fuelType, boolean isElectric, String transmission,int passCapacity,
			String model, int year, String color, double mileage,String type)
	{
		super(model,year,color,mileage,type);
		this.noDoor = noDoor;
		this.fuelType = fuelType;
		this.isElectric = isElectric;
		this.transmission = transmission;
		this.passCapacity = passCapacity;
	}
	public void displayCar()
	{
		System.out.println();
		System.out.println("No of door = "+this.noDoor);
		System.out.println("Fuel type = "+this.fuelType);
		System.out.println("Is electronic = "+this.isElectric);
		System.out.println("Transmission = "+this.transmission);
		System.out.println("Passenger capacity = "+this.passCapacity);
	}
}
class SingleLevelInheritance
{
	public static void main(String[] args)
	{
		Car obj1 = new Car(5,"Diseal",false,"Manual",9,"Scorpio",2023,"White",14,"SUV");
		obj1.displayVehicle();
		obj1.displayCar();
		Car obj2 = new Car(5,"Petrol",false,"Manual",5,"XUV 3XO",2023,"White",18,"Sub-compact");
		System.out.println();
		System.out.println();
		obj2.displayVehicle();
		obj2.displayCar();
	}
}
```

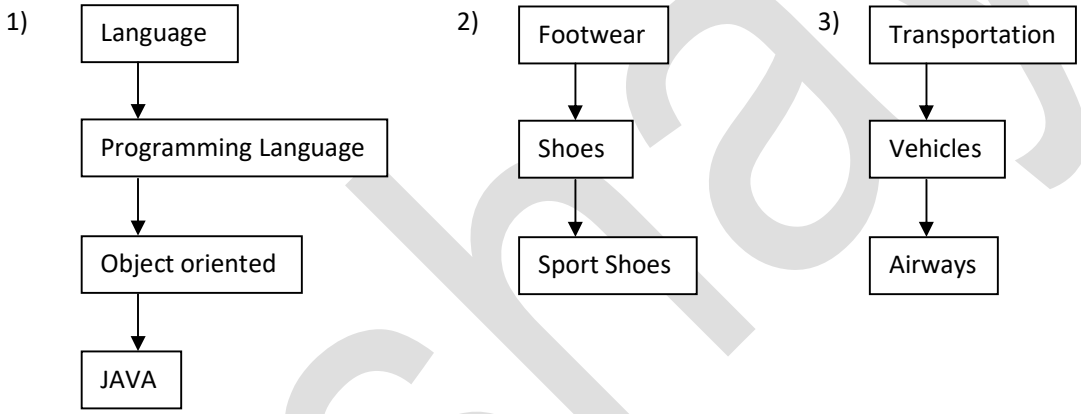
## 2. Multilevel Inheritance:-

- One class extends another class and the extended class become parent for another class, is known as multilevel inheritance.

**OR**

- Inheritance achieved on more than one level where one parent class has one child class, is known as multilevel inheritance.
- e.g.




**Example :- Device → Phone → Smartphone**

```
// Base class
class Device {
    String brand;
    String model;
    double weight;
    String batteryType;
    double price;

    // Constructor for Device
    Device(String brand, String model, double weight, String batteryType, double price) {
        this.brand = brand;
        this.model = model;
        this.weight = weight;
        this.batteryType = batteryType;
        this.price = price;
    }
    // Display device details
    void displayDeviceDetails() {
        System.out.println("Device Details:");
        System.out.println("Brand: " + this.brand);
        System.out.println("Model: " + this.model);
        System.out.println("Weight: " + this.weight + " grams");
        System.out.println("Battery Type: " + this.batteryType);
        System.out.println("Price: $" + this.price);
    }
```

```
}

// Intermediate class
class Phone extends Device {
    String networkType;
    int simSlots;
    boolean hasCamera;
    boolean hasBluetooth;
    boolean hasWifi;

    // Constructor for Phone
    Phone(String brand, String model, double weight, String batteryType, double price,
          String networkType, int simSlots, boolean hasCamera, boolean hasBluetooth, boolean hasWifi) {
        super(brand, model, weight, batteryType, price);
        this.networkType = networkType;
        this.simSlots = simSlots;
        this.hasCamera = hasCamera;
        this.hasBluetooth = hasBluetooth;
        this.hasWifi = hasWifi;
    }

    // Display phone details
    void displayPhoneDetails() {
        displayDeviceDetails();
        System.out.println("Network Type: " + this.networkType);
        System.out.println("SIM Slots: " + this.simSlots);
        System.out.println("Has Camera: " + this.hasCamera);
        System.out.println("Has Bluetooth: " + this.hasBluetooth);
        System.out.println("Has Wi-Fi: " + this.hasWifi);
    }
}

// Derived class
class SmartPhone extends Phone {
    String operatingSystem;
    int storageCapacity;
    int ram;
    boolean hasFingerprintSensor;
    boolean hasFaceUnlock;

    // Constructor for SmartPhone
    SmartPhone(String brand, String model, double weight, String batteryType, double price,
                        String networkType, int simSlots, boolean hasCamera, boolean hasBluetooth, boolean
hasWifi,
                        String operatingSystem, int storageCapacity, int ram, boolean hasFingerprintSensor,
                         boolean hasFaceUnlock)
{
        super(brand, model, weight, batteryType, price, networkType, simSlots, hasCamera, hasBluetooth,
hasWifi);
        this.operatingSystem = operatingSystem;
        this.storageCapacity = storageCapacity;
        this.ram = ram;
        this.hasFingerprintSensor = hasFingerprintSensor;
        this.hasFaceUnlock = hasFaceUnlock;
    }
```

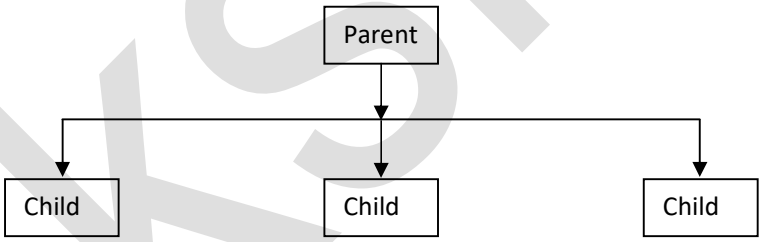
```
    // Display smartphone details
    void displaySmartPhoneDetails() {
        displayPhoneDetails();
        System.out.println("Operating System: " + this.operatingSystem);
        System.out.println("Storage Capacity: " + this.storageCapacity + " GB");
        System.out.println("RAM: " + this.ram + " GB");
        System.out.println("Has Fingerprint Sensor: " + this.hasFingerprintSensor);
        System.out.println("Has Face Unlock: " + this.hasFaceUnlock);
    }
}

// Main class to test multilevel inheritance
public class MultilevelInheritanceExample {
    public static void main(String[] args) {
        // Creating a SmartPhone object
        SmartPhone smartphone = new SmartPhone("Apple", "iPhone SE 2", 150, "Lithium-Ion", 499,
                                               "4G", 2, true, true, true,
                                                   "iOS", 128, 3, true, false);

        // Displaying SmartPhone details
        smartphone.displaySmartPhoneDetails();
    }
}
```

### 3. Hierarchical Inheritance:-

- Multiple subclasses inheriting from a same superclass is known as hierarchical inheritance.

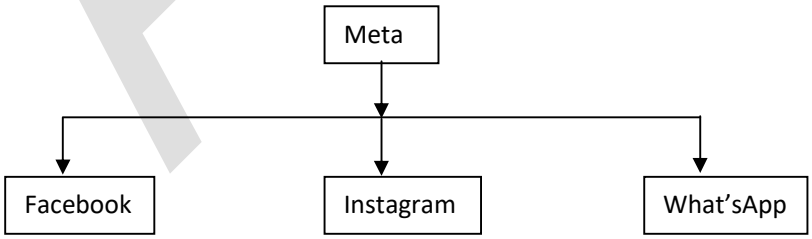
**OR**

- A super class has more than one subclass on a same level is known as hierarchical inheritance.

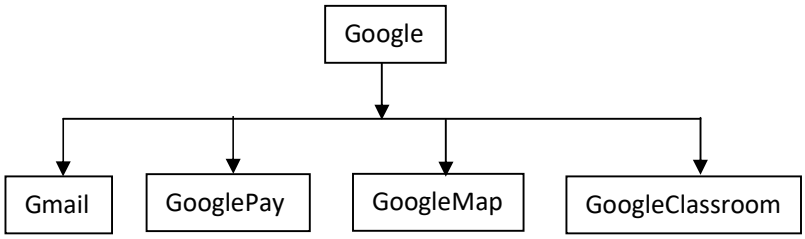



- **e.g.**

1)



2)




**Example on Google – Gmail,GooglePay,GoogleMap,GoogleClassroom**

```
//base class
import java.util.Arrays;
class Google
{
String username;
String pass;
String dob;
String emailId;
long contact;
String gender;
Google(String username,String pass,String dob,String emailId, long contact,String gender)
{
        super();
        this.username = username;
        this.pass = pass;
        this.dob = dob;
        this.emailId = emailId;
        this.contact = contact;
        this.gender = gender;
}
public void displayGoogle()
{
        System.out.println();
        System.out.println("Google Details");
        System.out.println("Username : "+this.username);
        System.out.println("Password : "+this.pass);
        System.out.println("Date of birth : "+this.dob);
        System.out.println("Email id : "+this.emailId);
        System.out.println("Gender : "+this.gender);
}
}
//child class
class GooglePay extends Google
{
String name;
long accountNumber;
String ifscCode;
int upiPin;
long debitCard;
GooglePay(String name,long accountNumber,String ifscCode,int upiPin,long debitCard,
                        String username,String pass,String dob,String emailId, long contact,String
gender)
{
        super(username,pass,dob,emailId,contact,gender);
        this.name = name;
        this.accountNumber = accountNumber;
        this.ifscCode = ifscCode;
        this.upiPin = upiPin;
```

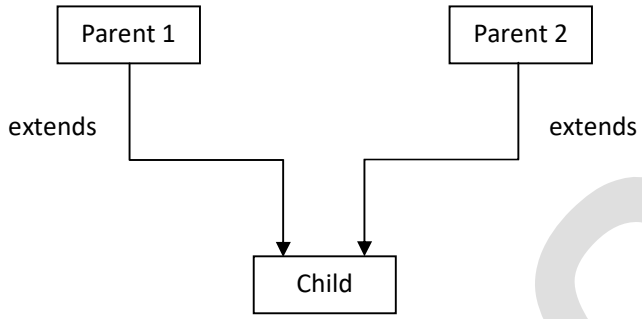
```
        this.debitCard = debitCard;
    }
    public void dipslayGooglePay(){
        displayGoogle();
        System.out.println();
        System.out.println("Google Pay Details");
        System.out.println("Name : "+this.name);
        System.out.println("Account Number : "+this.accountNumber);
        System.out.println("IFSC code : "+this.ifscCode);
        System.out.println("UPI pin : "+this.upiPin);
        System.out.println("Debit card : "+this.debitCard);
    }
    }
    //child class
    class GoogleMap extends Google
    {
    String currentLoc;
    String [] bookmarks;
    String modeTravel;
    GoogleMap(String currentLoc,String [] bookmarks,String modeTravel,
                    String username,String pass,String dob,String emailId, long contact,String
    gender)
    {
        super(username,pass,dob,emailId,contact,gender);
        this.currentLoc = currentLoc;
        this.bookmarks = bookmarks;
        this.modeTravel = modeTravel;
    }
    public void displayGoogleMap()
    {
        displayGoogle();
        System.out.println();
        System.out.println("Google Map Details");
        System.out.println("Current location : "+this.currentLoc);
        System.out.println("bookmarks : "+Arrays.toString(this.bookmarks));
        System.out.println("Mode of travel : "+this.modeTravel);
    }
    }
    //child class
    class GoogleMeet extends Google
    {
    String hostName;
    String joiningLink;
    String schedule;
    String meetId;
    GoogleMeet(String hostName,String joiningLink,String schedule,String meetId,
                    String username,String pass,String dob,String emailId, long contact,String
    gender)
    {
        super(username,pass,dob,emailId,contact,gender);
        this.hostName = hostName;
        this.joiningLink = joiningLink;
        this.schedule = schedule;
        this.meetId = meetId;
    }
    public void displayGoogleMeet()
    {
        displayGoogle();
        System.out.println();
        System.out.println("Google Meet Deatils");
```

```
        System.out.println("Host name : "+this.hostName);
        System.out.println("Joining link : "+this.joiningLink);
        System.out.println("Schedule : "+this.schedule);
        System.out.println("Meet id : "+this.meetId);
}
}
//child class
class GoogleClassroom extends Google
{
String className;
String subject;
String teacherName;
String classId;
int countStudent;
GoogleClassroom(String className,String subject,String teacherName,String classId,int
countStudent,
                String username,String pass,String dob,String emailId, long contact,String
gender)
{
        super(username,pass,dob,emailId,contact,gender);
        this.className = className;
        this.subject = subject;
        this.teacherName = teacherName;
        this.classId = classId;
        this.countStudent = countStudent;
}
public void displayGoogleClassroom()
{
        displayGoogle();
        System.out.println();
        System.out.println("Google Classroom Details");
        System.out.println("Class Name : "+this.className);
        System.out.println("Subject : "+this.subject);
        System.out.println("Teacher name : "+this.teacherName );
        System.out.println("Class id : "+this.classId);
        System.out.println("Count of student : "+this.countStudent);
}
}
//driver class
class HierarchicalInheritanceExample
{
public static void main(String[] args)
{
        GooglePay obj1 = new
GooglePay("Akshay",123456789l,"MODNIM007",7809,878967551234L,
        "Akshay","Akshay@123","06/08/1999","akshaymali1137@gamil.com",8007929317l,"Male");
        obj1.dipslayGooglePay();
        System.out.println();

        GoogleMap obj2 = new GoogleMap("JM road Deccan Pune",
                                        (new String[] {"FC Road","JM Road"}),"Bike",
        "Akshay","Akshay@123","06/08/1999","akshaymali1137@gamil.com",8007929317l,"Male");
        obj2.displayGoogleMap();
        System.out.println();

        GoogleMeet obj3 = new GoogleMeet("HR Qspiders","meeet.google.com/abghfk123..//kjk",
                                        "09.00 AM","ABC777",
        "Akshay","Akshay@123","06/08/1999","akshaymali1137@gamil.com",8007929317l,"Male");
        obj3.displayGoogleMeet();
        System.out.println();
```

```
        GoogleClassroom obj4 = new GoogleClassroom("M-17","JAVA","Shrikant
Kokate","jet89",150,
        "Akshay","Akshay@123","06/08/1999","akshaymali1137@gamil.com",8007929317l,"Male");
        obj4.displayGoogleClassroom();
}
}
```

### 4. Multiple Inheritance:-

- When a subclass inherits from more than one super class is known as multiple inheritance.
- e.g.




- Unlike any other programming languages like C++, java doesn't support multiple inheritance in classes.
- If we try to achieve multiple inheritance it leads to diamond problem, so instead of resolving this problem java has restricted multiple inheritance in classes and because of which we can extend only one class with another.

```
class Parent1
{
    Parent1()
    {
        super();
    }
    void m1()
    {
    System.out.println("Parent1 m1()");
    }
}
```

```
class Parent2
{
    Parent2()
    {
        super();
    }
    void m1()
    {
    System.out.println("Parent2 m1()");
    }
}
```

Diamond Problem

```
class Child extends Parent1,Parent2
{
    Child()
    {
        super();
    }
    public static void main(String[] args)
    {
        Child obj = new Child();
        obj.m1();
    }
}
```

- A diamond problem arises when a class inherits from more than one super class.
- Its an ambiguity which occurs at the runtime.
- So consider we have two super class Parent1 and Parent2 and a subclass Child.
- When we create an object of the Child class, constructor of Child class contains super() statement which will create an ambiguity which parent constructor must be invoked i.e. Parent1 or Parent2.
- If both super classes have same method signature again it will create an amniguity for the child class which method must be called for the execution.

**Note:-**

- Instead of solving this problem java has restricted multiple inheritance in classes but we can achieve it using interfaces.

**e.g.**

(classes)

```
class Parent1
{

}
class Parent2
{

}
class Child extends Parent1,Parent2
{

}

Output:-
Compile time error
```

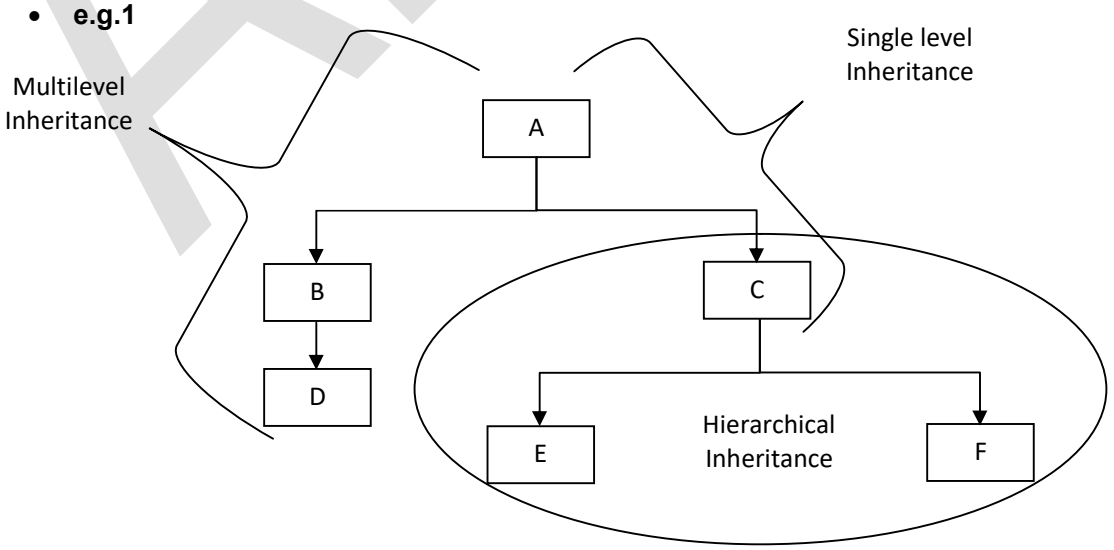
(interfaces)

```
interface Parent1
{

}
interface Parent2
{

}
interface Child extends Parent1,Parent2
{

}

Output:-
No Compile time error
```

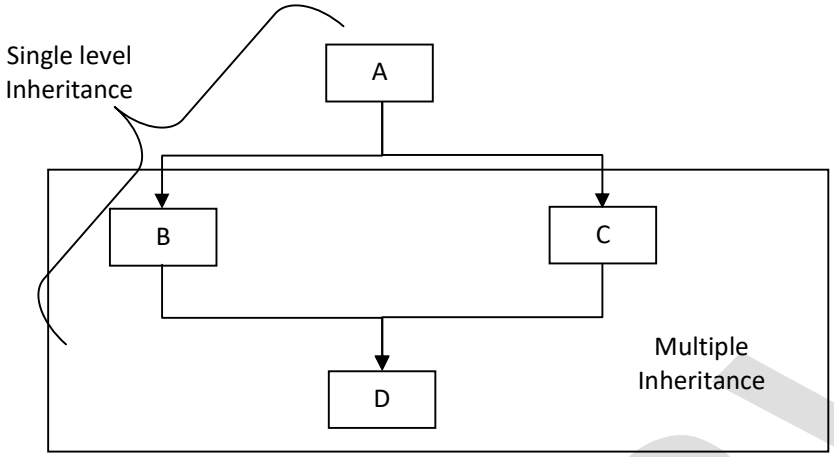
## 5. Hybrid Inheritance:-

- It's a combination of more than one type of inheritance.
- If we combine single level, multi level and hierarchical inheritance then we can achieve hybrid inheritance.
- But if we try to include multiple inheritance or cyclic inheritance with any other type of inheritance then we cannot achieve hybrid inheritance in classes.

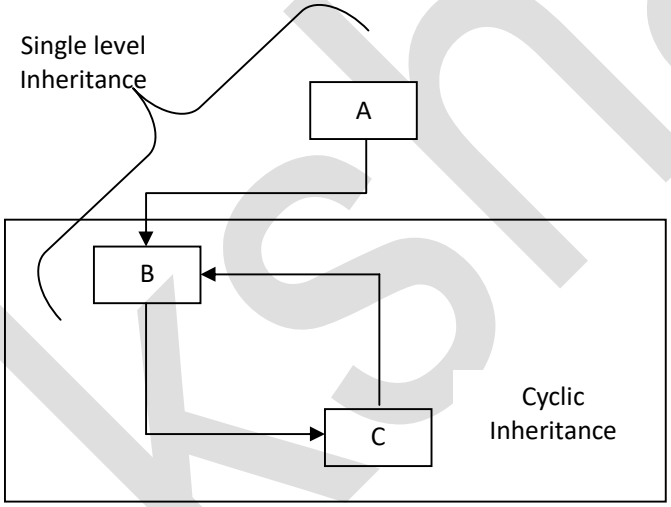
- **e.g.1**



- **e.g.2**




- **e.g.3**



## 6. Cyclic Inheritance:-

- A cyclic inheritance occurs when a class is directly or indirectly depends on itself.
- e.g. class A extend class B and class B extend class A
  This is indirect dependency where a subclass becomes a parent for its superclass.
- When a class extends itself means it is directly dependent on itself.

**Note:-**

Cyclic inheritance is conceptual thing and java doesn't support it.

**e.g. 1. Direct Dependency**

```
class Example extends Example
{

}
```

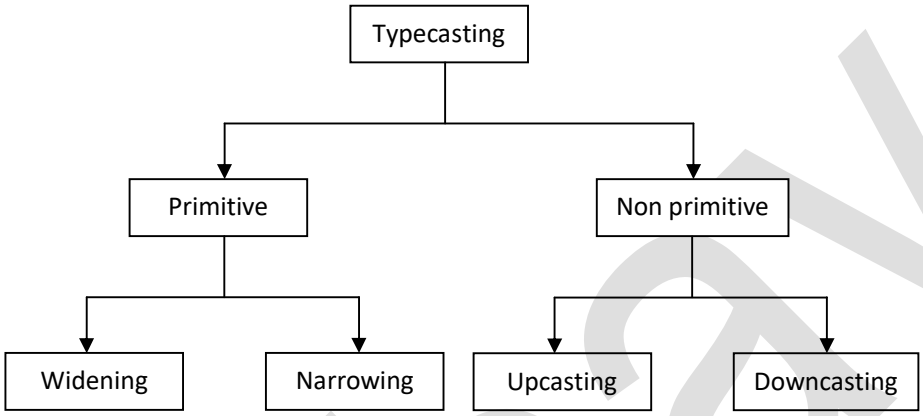
**2. Direct Dependency**

```
class Ramesh extends Suresh
{

}
class Suresh extends Ramesh
{

}
```

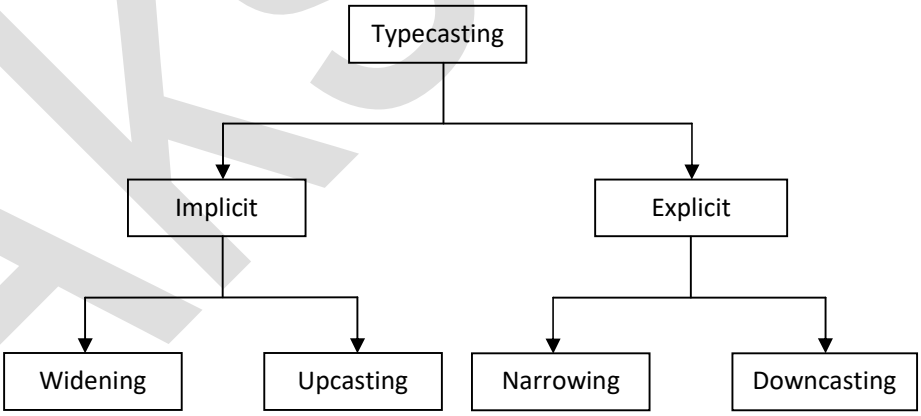
# ★ Typecasting Non-primitive:-

- It is a process of coverting one data type into another data type.
- Typecasting classified as based on data types, based on operation

## Based on data types




## Based on opoeration



★ **Non primitive Typecasting:-**

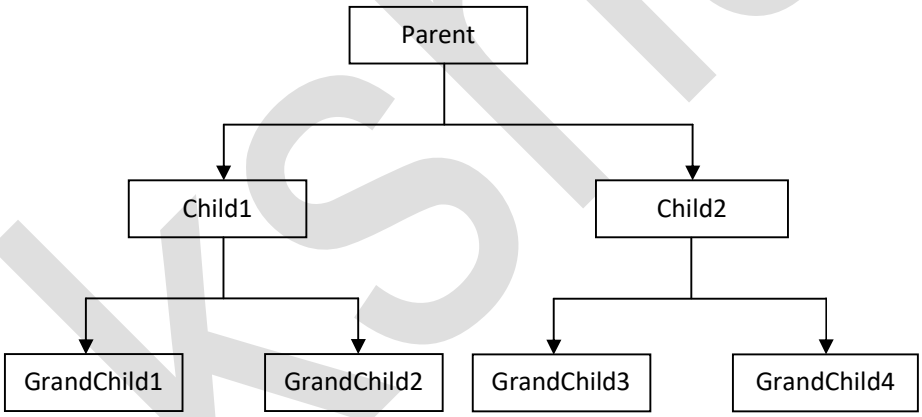
- The process of converting one type of reference into another type is known as non primitive type casting (derived type casting).
- Non primitive type casting is classified into two types,
  1. Upcasting:-
  2. Downcasting:-

**Note:-**

To perform non primitive type casting is–a–relationship is mandatory (It can be parent-child or child-parent).

**1. Upcasting:-**

- When the reference variable of super class refers to the object of subclass is known as upcasting.
- **Syntax:-** Superclass obj = new Subclass();
- e.g:- Object obj1 = new String();
  Object obj2 = new StringBuffer();
  List obj3 = new LinkedList();
  List obj4 = new ArrayList();
- Upcasting is a process of converting a subclass type to a superclass type.
- It is done implicitly by the compiler.
- Upcasting is done to achieve generalization.



**Program:-**

```
class Parent{}
class Child1 extends Parent{}
class Child2 extends Parent{}
class GrandChild1 extends Child1{}
class GrandChild2 extends Child1{}
class GrandChild3 extends Child2{}
class GrandChild4 extends Child2{}
class Upcasting
{
	public static void main(String[] args)
	{
		Child1 obj = new GrandChild1();
		Child1 obj1 = new GrandChild3();	//CTE
		Child2 obj2 = new GrandChild3();
		Child2 obj3 = new GrandChild1();	//CTE
		Parent obj4 = new Child1();
		Parent obj5 = new Child2();
		Parent obj6 = new GrandChild1();
		Parent obj7 = new GrandChild2();
		Parent obj8 = new GrandChild3();
		Parent obj9 = new GrandChild4();
	}
}
```

**Note:-**

We cannot store a superclass object reference into subclass type.
(Parent object cannot be converted into child type)

**Child1 obj = new Parent(); //CTE**

**Program:-**

```
class Parent
{
	String a = "Hello from parent";
	public void m1()
	{
		System.out.println("Parent class m1()");
	}
	public void greeting()
	{
		System.out.println("Good Morning");
	}
}
class Child extends Parent
{
	String b = "Hello from child";
	public void m2()
	{
		System.out.println("Child class m2()");
	}
	public void greeting()
	{
		System.out.println("Good afternoon");
	}
}
class Demo
{
```

```
    public static void main(String[] args)
    {
        Child obj = new Child();        //specialization
        System.out.println(obj.a);
        obj.m1();
        System.out.println(obj.b);
        obj.m2();
        Parent obj1 = new Child();      //generalization
        System.out.println(obj.a);
        obj1.m1();
        System.out.println(obj1.b);     //X
        obj1.m2();                      //X
        obj1.greeting();                        //Good afternoon
    }
}
```

★ **Specialization:-**

"Creating an object of a class and storing it inside a same reference variable of that class is called as specialization."

- When two classes has is-a-relationship with each other, and when we achieve specialization i.e. creating a object of subclass we can access / invoke all the members of sub class as well as super class.
- When we achieve generalization child type object behaves like a parent type object and we can only access the members of its parent class except the overridden method.

**Note:-**

To access the members of child class we need to perform downcasting.

★ **Benefits of Upcasting:-**

1. Polymorpshism:-
   It enables to use one method with different implementation.

2. Code reusability:-
   Methods defined for superclass type can be reused for any subclass.

3. Dynamic method dispatch:-
   It ensures that correct method is called at the runtime based on the type of object.

## 2. Downcasting:-

- When a subclass reference refers to superclass object type is known as downcasting.

**OR**

- When a super class object type is converted to subclass is known as downcasitng.
- It is also known as specializaiton.

**Program:-**

```
class Ramesh{}
class Suresh extends Ramesh{}
class Demo{
    public static void main(String[] args) {
            Ramesh obj = new Suresh();
            Suresh obj1 = (Suresh)obj;
    }
}
```

- Downcasting is done by the programmer explicitly using a typecast operator.

★ **Purpose of downcasting:-**

- When we perform upcasting (generalization) we cannot invoke the members of subclass.
- Therefore to access the member we perform downcasting.
- We cannot perform downcasting without performing upcasting.

★ **Rules for downcasting:-**

```
Object   obj   =   new   String();

String   obj1   =   (String)obj;
    |      |          |     |
    ↓      ↓          ↓     ↓
    A      b    =     (C)   d;
```

**Rules at compile time:-**

- The type of 'C' and 'd' must have some relationship either parent-child or child-parent.
- 'C' must be the same type of 'A' or 'C' must be the derived type of 'A'.

**Rules at runtime:-**

- The underline original object of 'd' must be the same type or derived type of 'C'.

**Example:-**

```
class Parent{}
class Child1 extends Parent{}
class Child2 extends Parent{}
class GrandChild1 extends Child1{}
class GrandChild2 extends Child1{}
class GrandChild3 extends Child2{}
class GrandChild4 extends Child2{}
class Downcasting
{
    public static void main(String[] args)
    {
            Parent obj = new GrandChild2();              //no runtime error
            Child1 obj1 = (Child1)obj;              //no runtime error
            GrandChild1 obj2 = (GrandChild1)obj1;       //runtime error
            Parent obj = new GrandChild3();
            Child2 obj1 = (GrandChild3)obj;
            GrandChild3 obj2 = (GrandChild3)obj1;       //no CTE or RTE

            GrandChild3 obj3 = (GrandChild3)obj;

            Child1 obj = new GrandChild1();
```

```
            GrandChild2 obj1 = (GrandChild2)obj;        //Runtime error
            GrandChild1 obj1 = (GrandChild1)obj;
        }
    }
```

★ **ClassCastException:-**

- When we try to downcast a reference variable to a subclass type and the object reference does not have an instance of the subclass type, then we will get a ClassCastException.
- In simple words it means when we try to perform downcasting without performing upcasting we get a ClassCastException.

**Program:-**

```
import java.util.Scanner;
class Payment
{
    //username
    //account number
    //phone number
    //ifsc code
    public void modeOfPayment(Payment obj)   //upcasting
    {
            if(obj instanceof GooglePay)
            {
                    GooglePay obj1 = (GooglePay)obj;   //downcasting
                    System.out.println(obj1.str);
            }
            else if(obj instanceof PhonePay)
            {
                    PhonePay obj1 = (PhonePay)obj;     //downcasting
                    System.out.println(obj1.str);
            }
            else
            {
                    Paytm obj1 = (Paytm)obj;      //downcasting
                    System.out.println(obj1.str);
            }
    }
}
class GooglePay extends Payment
{
    String str = "Hello from GooglePay";
}
class PhonePay extends Payment
{
    String str = "Hello from PhonePay";
}
class Paytm extends Payment
{
    String str = "Hello from Paytm";
}

class EcommercePayment
{
    public static void main(String[] args)
    {
            System.out.println("Payment Option");
            System.out.println("1.GooglePay");
            System.out.println("2.PhonePay");
```

```
                System.out.println("3.Paytm");
                System.out.println("4.COD");
                System.out.println("Enter your option : ");
                int opt = new Scanner(System.in).nextInt();
                switch(opt)
                {
                        case 1 :
                        {
                                GooglePay obj = new GooglePay();
                                obj.modeOfPayment(obj);
                                break;
                        }
                        case 2 :
                        {
                                PhonePay obj = new PhonePay();
                                obj.modeOfPayment(obj);
                                break;
                        }
                        case 3 :
                        {
                                Paytm obj = new Paytm();
                                obj.modeOfPayment(obj);
                                break;
                        }
                        case 4 :
                        {
                                System.out.println("Cash on Delivery");
                                break;
                        }
                        default:
                        System.out.println("Wrong option");
                }
        }
}
```

★ **instanceof:-**

- instanceof is a keyword and operator in java which is used to check whether an object is an instance of a class.
- It is also known as type comparison operator because it compares the instance with its type.
- The return type of instanceof opearator is boolean.
  (i.e. true or false)
- It is an type of binary operator (mislleaneous operator).

# 3. Polymorphism:-

- Polymorphism is a greek word where poly means many and morph means forms.
- Polymorphism stands for one name and many forms.
- The dictionary definition of polymorphism comes from biology where a species has multiple forms.
- E.g. water → liquid, gas, solid
- Polymorphism is classified into two types,
  1. Compile time polymorphism
  2. Runtime polymorphism

## 1. Compile time polymorphism:-

- The binding achieved at compile time is known as compile time polymorphism / static polymorphism / static binding / early binding.

OR

- The binding achieved at compile time and some behaviour is executed, this scenario is done by compiler is known as compile time polymorphism.
- Binding means an association of method calling statement and its definition.
- E.g.
  Method call statement will be binded with method definition.
  Constrtuctor call will be binded with constructor definition.

**Types of compile time polymorphism:-**

1. Method Overloading
2. Constructor Overloading
3. Operator Overloading
4. Method Shadowing
5. Variable Shadowing

## 1. Method Overloading:-

- It is a design process of creating more than one method with same name and different formal arguments in a same class is known as method overloading.
- E.g. print(), println(), wait() (object class method), valueof() (String class method),
- split() (String class method)

**Overloading of static method:-**

```
class Demo{
    public static int addition(int num1,int num2){
            return num1+num2;
    }
    public static int addition(int num1,int num2,int num3){
            return num1+num2+num3;
    }
    public static int addition(int num1,int num2,int num3,int num4){
            return num1+num2+num3+num4;
    }
}
class Example{
    public static void main(String[] args) {
            System.out.println(Demo.addition(10,10));
            System.out.println(Demo.addition(10,10,10));
            System.out.println(Demo.addition(10,10,10,10));
    }
}
```

**Overloading a non static method:-**

```
class Demo
{
    public int multiplication(int num1,int num2)
    {
            return num1*num2;
    }
    public int multiplication(int num1,int num2,int num3)
    {
            return num1*num2*num3;
    }
    public int multiplication(int num1,int num2,int num3,int num4)
    {
            return num1*num2*num3*num4;
    }
}
class Example
{
    public static void main(String[] args)
    {
            Demo obj = new Demo();
            System.out.println(obj.multiplication(5,5));
            System.out.println(obj.multiplication(5,5,5));
            System.out.println(obj.multiplication(5,5,5,5));
    }
}
```

---

```
class Parent
{
    public void m1()
    {
            System.out.println("No argument method");
    }
}
class Child extends Parent
{
    public void m1(int a)
    {
            System.out.println("Argument method");
    }
}
class Demo
{
    public static void main(String[] args)
    {
            Child obj = new Child();
            obj.m1(10);
    }
}
```

The above program is an example of method overloading because it inherits a no argument m1() from parent class and has its own parameterized m1() method.

```
class Parent
{
    public static void m1()
    {
            System.out.println("No argument method");
    }
}
class Child extends Parent
{
    public static void m1(int a)
    {
            System.out.println("Argument method");
    }
}
```

Static members are not inherited from one class to another class so the above example is not method overloading as the child class has only one m1() method.

---

```
class Demo
{
    public static void m1()
    {
            System.out.println("m1() static");
    }
    public void m1(int a)
    {
            System.out.println("m1() non static");
    }
}
```

It doesn't matter method might be static or non static compiler checks for method name and different formal arguments.

---

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

Output:-
Compiler Time Error
m1() is already defined

**Can we overload main()?**
Yes we can overload main() by changing its argument and defining it more than once.

```
class Demo
{
  public static void main(String[] args)
  {
        System.out.println("Hello from main");
        Demo.main();
  }
  public static void main()
  {
        System.out.println("hello from no argument main()");
  }
}
```

## 2. Constructor Overloading:-

- Creating more than one constructor in a class is known as constructor overloading.
- e.g.

```
class Demo
{
    Demo()
    {
        super();
        System.out.println("No argument const");
    }
    Demo(int a)
    {
        super();
        System.out.println("int argument const");
    }
    Demo(byte a)
    {
        super();
        System.out.println("byte argument const");
    }
}
class Driver
{
    public static void main(String[] args)
    {
        Demo obj1 = new Demo();
        Demo obj2 = new Demo(10);
        Demo obj3 = new Demo((byte)10);
    }
}
```

**Output:-**

```
No argument const
int argument const
byte argument const
```

★ **Constructor Chaining:-**

- It is process of calling one constructor from another constructor is known as constructor chaining.
- Constructor chaining can be achieved using this() statement and super() statement.

**Example:-**

1. **Constrctor chaining using this() statement:-**

```
class Demo
{
    Demo()
    {
            super();                    //invokes object class constructor
            System.out.println("No argument const");
    }
    Demo(int a)
    {
            this();             //invokes current class constructor
            System.out.println("int argument const");
    }
    Demo(String str)
    {
            this(10);           //invokes int argument constructor
            System.out.println("String argument const");
    }
}
class Driver
{
    public static void main(String[] args)
    {
            Demo obj = new Demo("Hello");       //invokes string argument constrctor
    }
}
```

**Outout:-**

```
No argument const
int argument const
String argument const
```

2. **Constructor chaining using super() statement:-**

```
class Parent
{
    Parent()
    {
            super();                    //invokes object class constructor
            System.out.println("Parent class No argument const");
    }
    Parent(int a)
    {
            this();             //invokes current class constructor
            System.out.println("Parent class int argument const");
    }
}
class Child extends Parent
{
    Child()
    {
            super(10);      //invokes parent class int argument constructor
            System.out.println("Child class no argument const");
    }
    Child(int a)
    {
            this();   //invokes current class constructor
            System.out.println("Child class int argument const");
    }
}
class Driver
{
    public static void main(String[] args)
    {
            Child obj = new Child(10);      //invokes child class int argument constructor
    }
}
```

Output:-

```
Parent class No argument const
Parent class int argument const
Child class no argument const
Child class int argument const
```

★ **Can we call a constructor explicitly?**

- Yes we can call a constructor explicitly.
- this() statement and super() statement are used to call a constructor explicitly.

★ **Method Chaining:-**

- It is a process of calling one method from another method is known as method chaining.
- Example:-

```
class Demo
{
    public static void main(String[] args)
    {
            String str = "hello";
            char ch = str.toUpperCase().toLowerCase().charAt(0);
            System.out.println(ch);
    }
}
```

Output:-

```
h
```

**3. Operator Overloading:-**

- Operator overloading is used to specify user defined implementation for operation on one or more operands.
- E.g. '+' operator, if both the operands are numbers it will perform addition operation,
  If any one of the operand is String it will perform concatenation operation.

**'+' is the only operator overloading example in java.**

**Note:-**

Operator overloading is not supported in java to avoid the complexity and confusion with the operators.

**4. Method Shadowing:-**

- If a sub class and a super class has static method with same signature (method name and formal argument) is known as method shadowing.
- Where a subclass method shadows (hides) the super class method.

★ **Rules for method shadowing:-**

1. Is a relationship is mandatory.
2. Methods must be static.
3. Method signature must be same.
4. Return type:-
   i. Return type must be same for primitive datatype.
   ii. For non primitive datatype it can be covariant.
5. We cannot reduce the scope of access modifier from parent to child.

**Note:-**

Method shadowing is method overriding for static methods.

**Program:-**

```
class Parent{
  public static void m1(){
        System.out.println("Hello from parent m1()");
  }
}
class Child extends Parent{
  public static void m1(){
        System.out.println("Hello from child m1()");
  }
}
class Demo {
  public static void main(String[] args) {
        Child obj1 = new Child();       //specialization
        obj1.m1();
        Parent obj2 = new Child();      //upcasting
        obj2.m1();
        Child obj3 = (Child)obj1;       //downcasting
        obj3.m1();
  }
}
```

Output :-

```
Hello from child m1()
Hello from parent m1()
Hello from child m1()
```

- In specialization child class method shadows the parent class method and we cannot access parent class shadowed method.
- So if you need to access the parent class shadow method there are two ways,
  1. We can access it by using classname as a reference
  2. By performing upcasting

**Progam:-**

```
class Parent{
  public static void m1(){
        System.out.println("Hello from parent m1()");
  }
}
class Child extends Parent{
  public static int m1(){
        System.out.println("Hello from child m1()");
        return 0;
  }
}
class Demo {
  public static void main(String[] args) {
        Child obj1 = new Child();       //specialization
        obj1.m1();
        Parent obj2 = new Child();      //upcasting
        obj2.m1();
        Child obj3 = (Child)obj1;       //downcasting
        obj3.m1();
  }
}
```

Output:-

Compile time error

m1() in Child cannot hide m1() in Parent

return type int is not compatible with void

**Program:-**

```
class Parent{
  public static int m1(){
        System.out.println("Hello from parent m1()");
        return 0;
  }
}
class Child extends Parent{
  public static int m1(){
        System.out.println("Hello from child m1()");
        return 0;
  }
}
```

Output:-

No compile time error

**Program:-**

```
class Parent{
  public static String m1(){
        System.out.println("Hello from parent m1()");
        return null;
  }
}
class Child extends Parent{
  public static String m1(){
        System.out.println("Hello from child m1()");
        return null;
  }
}
```

Output:-
No compile time error

**Program:-**

```
class Parent{
  public static Object m1(){
        System.out.println("Hello from parent m1()");
        return null;
  }
}
class Child extends Parent{
  public static String m1(){
        System.out.println("Hello from child m1()");
        return null;
  }
}
```

Output:-
No compile time error

**Program:-**

```
class Parent{
  public static String m1(){
        System.out.println("Hello from parent m1()");
        return null;
  }
}
class Child extends Parent{
  public static Object m1(){
        System.out.println("Hello from child m1()");
        return null;
  }
}
```

Output:-
Compile time error
m1() in Child cannot hide m1() in Parent
return type Object is not compatible with String

**Program:-**

```
class Parent{
  public static String m1(){
        System.out.println("Hello from parent m1()");
        return null;
  }
}
class Child extends Parent{
  public static StringBuffer m1(){
        System.out.println("Hello from child m1()");
        return null;
  }
}
```

Output:-
Compile time error
m1() in Child cannot hide m1() in Parent
return type StringBuffer is not compatible with String

**Program:-**

```
class Parent{
  public static int m1(){
        System.out.println("Hello from parent m1()");
        return 0;
  }
}
class Child extends Parent{
  private static int m1(){
        System.out.println("Hello from child m1()");
        return 0;
  }
}
```

Output:-
Compile time error
m1() in Child cannot override m1() in Parent
attempting to assign weaker access privileges; was public

**Program:-**

```
class Parent{
  static int m1(){
        System.out.println("Hello from parent m1()");
        return 0;
  }
}
class Child extends Parent{
  public static int m1(){
        System.out.println("Hello from child m1()");
        return 0;
  }
}
```

Output:-
No compile time error
Because we are increasing the scope

**Note:-**

- **Method shadowing is also known as method hiding.**
- **Method shadowing is method overriding for static methods.**

## 5. Variable Shadowing:-

- When a subclass and superclass contains a variable with same name, subclass variable shadows the variable of parent class is known as variable shadowing.
- Variable shadowing can happen with static or non static variable.

★ **Variable shadowing (static variable):-**

- When variable shadowing happens, you cannot access the variable of parent class using child reference so we need to perform (Upcasting) generalization, and also we can access it using classname as a reference.

**Program:-**

```
class Parent{
    static String str = "Parent var";
}
class Child extends Parent{
        static String str = "Child var";
        public static void m1(){
                String str = "Local var";
                System.out.println(str);                //local variable
                System.out.println(Child.str);          //Child var
                System.out.println(Parent.str);         //Parent var
        }
}
class Demo
{
        public static void main(String[] args){
                System.out.println("main()");
                Child.m1();
                Parent obj = new Child();
                System.out.println(obj.str);
                Child obj1 = (Child)obj;
                System.out.println(obj1.str);
        }
}
```

★ **Variable shadowing(non static variable):-**

- Variable shadowing in non static variable the subclass variable shadows parent class variable and to access the parent class variable we need to use super keyword.
- If variable shadowing happenes between global and local variable of same class to invoke global variable we use this keyword.

**Program:-**

```
class Parent{
    String str = "Parent var";
}
class Child extends Parent{
    String str ="Child var";
    public void m1(){
            String str = "Local var";
            System.out.println(str);
            System.out.println(this.str);
            System.out.println(super.str);
    }
}
class Demo{
    public static void main(String[] args){
            Child obj = new Child();
            obj.m1();
    }
}
```

## 2. Runtime Polymorphism:-

- The binding achieved at runtime is known as runtime polymorphism.
- Runtime polymorohism is also known as dynamic polymorphism / dynamic binding / late binding.
- Runtime polymorphism is achieved by method overriding.

### ★ Method overrinding:-

- If a subclass and a superclass have non static method with same signature is known as method overriding.
- Example for method overriding,

```
class Parent{
    public void m1(){
            System.out.println("parent m1()");      //overriden method
    }
}
class Child extends Parent{
    public void m1(){
            System.out.println("child m1()");       //overriding method
    }
}
class Demo{
    public static void main(String[] args){
            Child obj = new Child();
            obj.m1();
    }
}
```

- The method present inside the parent class is called overriden method.
- The method present inside the child class is called overriding method.

### ★ Rules for method overriding:-

1. Is—a—relationship is mandatory.
2. Methods must be non static.
3. Method must have be same signature.
4. Return type:-
   i. If return type is primitive both should have same primitive return type.
   ii. If return type is non primitive it can be covariant. (but should be in relationship Parent-Child).
5. We cannot reduce the scope of access modifier from parent to child.

### ★ Advantages of method overriding:-

1. The main advantage of method overriding is that it gives the child class the ability to change the behaviour of parent class.
2. It provides multiple implementation for same method and one can call parent or child method as per the need.
3. It prevents duplication of code across multiple classes since subclass can override only the required methods from superclass.
4. Override is crucial when implementing interfaces.
5. By overriding methods a subclass can hide the internal working of several methods and we can achieve abstraction.

**Example:-**

```
Method shadowing and method overriding
class Parent
{
    public static void m1()
    {
            System.out.println("Parent m1() static");
    }
    public void m2()
    {
            System.out.println("Panret m2() non-static");
    }
}
class Child extends Parent
{
    public static void m1()
    {
            System.out.println("Child m1() static");
    }
    public void m2()
    {
            System.out.println("Child m2() non-static");
    }
}
class Demo
{
    public static void main(String[] args)
    {
            Child obj = new Child();        //specialization
            obj.m1();
            obj.m2();
            Parent obj1 = new Child();     //generalization
            obj1.m1();
            obj1.m2();
    }
}
```

★ **How does JVM decides which method must be call at the runtime?**

- When a method is overridden in a subclass the JVM decides which version of method to call based on the actual object type at runtime not the reference type this term is refered as dynamic method displatch or late binding.
- The actual type of an object is the object which is created using new keyword.

1. **Can we override a main() ?**
   No, we cannot override a main() because it is static method and static methods are associated with the class itself and staic method cannot be inherited.

2. **Can we overload a main()?**
   Yes, we can overload a main(), by decalring the main() more than once with different formal arguments.

3. **Why main() is static?**
   In java the main() method is declared as static for several reasons, primarily related to how the java runtime environment starts a program,
   No need to create an object. The main() method serves as the entry point of the program and it is invoked by Java Virtual Machine (JVM) to start the execution of the program.

4. **Why main() contains String[] args as an argument?**
   Because, String[] args allows the user to pass arguments from the command line when running the program.

5. **Can we overload a constructor?**
   Yes, we can overload a constructor by declaring it more than once in a class with different formal arguments.

6. **Can we override a constructor ?**
   No, we cannot override a constructor because constructor having same name as class name and constructor is not inherited from parent class to child class but we can call the parent constructor by using the super() statement from child class.

7. **Is constructor recursion possible?**
   No, constructor recursion is not possible in java, because Java doesn't support constructor recursion the compiler throws an error that Recursive Constructor Invocation. Because every constructor must have the super() statement to call the parent class constructor (every class has an subclass of Object class in Java). By the constructor recursion the call never goes to Object class.

8. **Can we inherit the constructor from one class to another class?**
   No, we cannot inherit the constructor from one class to another class because the constructor have same name as class name and it is not inherited from parent class to child class. To call the constructor of parent class we have super() statement.

9. **Why cant we declare a class as private and protected?**
   We cant declare a class as private because private access modifier is only accessible inside the class i.e. declaring a class as private doesn't make any sense.
   We cant declare a class as protected because it will violate the inheritance property.

10. **Can we declare a class as final?**
    Yes we can declare a class as final, final class cannot be inherited, we declare a class as final when we don't want any other class to change its implementation. All the immutable classes in java are declared as final such as Stirng, Wrapper class, etc.

11. **Can we declare a method as final?**
    Yes, we can declare a method as final. A final method cannot be overrided. A method is declared as final because its implementation is complete and the superclass does not wants the subclass to change implementation so the method is declared as final.

**e.g.**

```
class RBI
{
  public final void guidelines()
  {
        System.out.println("Dont share your personal details with anyone.");
        System.out.println("RBI dont call for OTP");
  }
}
class SBI extends RBI
{
  public void rateOfInterest()
  {
        System.out.println("ROI : "+8.9+" % ");
  }
  public void guidelines()        //compiler time error
  {
        System.out.println("Dont share your personal details with anyone.");
        System.out.println("RBI dont call for OTP");
  }
}
class BOI extends RBI
{
  public void rateOfInterest()
  {
        System.out.println("ROI : "+8.9+" % ");
  }
}
class Demo
{
  public static void main(String[] args) {
        SBI obj = new SBI();
        obj.guidelines();
        obj.rateOfInterest();
  }
}
```

**Note:-**

- We cannot use final keyword in interface and abstract class declaration.
- We cannot declare an abstract method as final.

# 4. Abstraction:-

- It is a process of hiding implementation detail and displaying only the essential features is called as abstraction.

OR

- It is the design process which helps the service specifier (customer) to provide the most essential features of a project without providing implementaition details is known as abstraction.
- E.g. applying the break of car, human for digesting energy

**Abstraction can be achieved in two ways,**

1. abstract class
2. interface

## ★ abstract method:-

- A method which contains abstract modifier in its declaration statement is known as an abstract method.
- An abstract method does not contains any implementation.
- Syntax:-

  access modifier abstract return type method name (formal argument) ;

- E.g.

```
abstract class Demo
{
    public abstract void m1();
}
```

**Note:-**

If we want a class to have a particular method but we want the implementation of that method to be determined by its class child, then we declare the parent method as abstract.

e.g.

```
abstract class Parent{
        public abstract int addition(int num1,int num2);
}
class Child extends Parent{
        @Override
        public int addition(int num1,int num2){
                int op = num1+num2;
                return op;
        }
}
class Demo{
        public static void main(String[] args) {
                Child obj = new Child();
                System.out.println("Addition : "+obj.addition(10,20));;
        }
}
```

**Note:-**

- An abstract method cannot have implementation (method body).
- Semicolon is mandatory at the end of method declaration statement and method should contain an abstract modifier.
- If a class contains a single abstract method in it, then it is mandatory to declare the class as abstract.
- If we have any derived class of the abstract class then it is mandatory for the child class to provide implementation for the parent abstract method or it has to be declare itself as abstract.

**Can we declare a static method as abstract?**

- No, we cannot declare static method as abstract it will create a compile time error illegal combination of modifiers.
- Because an abstract method must be implemented by child class by overriding.
- Static methods in java cannot be override as they are not inherited from one class to another.
- e.g.

```
abstract class Parent
{
  static abstract void m1();
}
```

★ **Concrete method:-**

- A concrete method in java is a method that contains an implementation.
- It is opposed to abstract method which does not have any implementation.
- Concrete method can be overrided by its child class only if it not declared as final.
- e.g

```
class Demo
{
    public  void m1()
    {
            System.out.println("implementation of m1()");
    }
}
```

★ **Concrete class:-**

- A class which constains implementation for all its method is called as concrete class.
- e.g.

```
class Demo{
    void m1(){
            System.out.println("Hello form m1()");
    }
    void m2(){
            System.out.println("Hello from m2()");
    }
}
```

**Can we declare an abstract method final?**

- No, we cannot declare an abstract method as final because it will create a compile time error, illegal combination of modifiers.
- Because an abstract method are intended to be overrided by subclass while a final method are intended to prevent overriding.
- In simple words abstract means no implementation and final means complete implementation and it creates an illegal combination.

## 1. abstract class:-

- A class which contains an abstract keyword in its declaration is known as abtract class.
- Syntax:-

```
abstract class Class_Name
{

}
```

- An abstract class may or may not contains any abstract method in it.
- But if a class has atleast one abstract method in it then it is mandatory to declare the class as abstract.
- An abstract class cannot be instanciated (Cannot create an object).

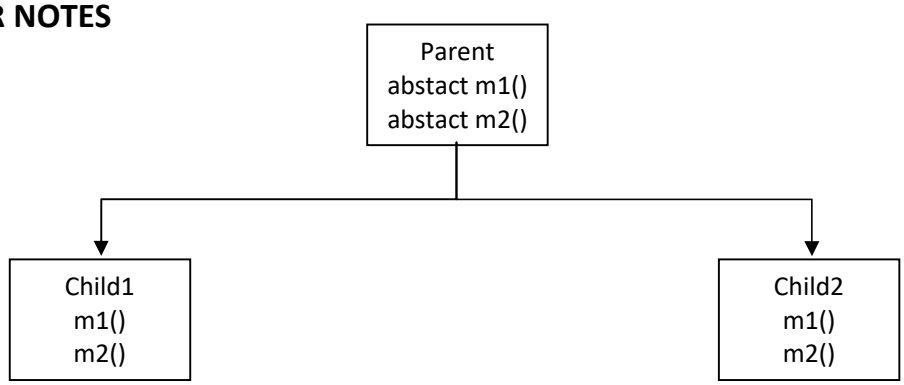
**Note:-**

To utilize any members of an abstract class we have to inherit it using in child class.

e.g.

```
abstract class Parent{
    abstract void m1();
    abstract void m2();
    void m3(){
            System.out.println("m3() from Parent class");
    }
}
abstract class Child extends Parent{
    void m1(){
            System.out.println("m1() implementaion from Child class");
    }
    void m3(){
            System.out.println("m3() from child class");
    }
}
class GrandChild extends Child{
    void m2(){
            System.out.println("m2() implementaion from GrandChild class");
    }
    void m4(){
            System.out.println("m4() from child class");
    }
}
class Demo{
    public static void main(String[] args) {
            GrandChild obj = new GrandChild();
            obj.m1();
            obj.m2();
            obj.m3();
            obj.m4();
    }
}
```

**e.g.**




```
abstract class Parent
{
    abstract void m1();
    abstract void m2();
}
class Child1 extends Parent
{
    void m1()
    {
            System.out.println("m1() implementation from Child1");
    }
    void m2()
    {
            System.out.println("m2() implementation from Child1");
    }
}
class Child2 extends Parent
{
    void m1()
    {
            System.out.println("m1() implementation from Child2");
    }
    void m2()
    {
            System.out.println("m2() implementation from Child2");
    }
}
class Demo
{
    public static void main(String[] args)
    {
            Child1 obj1 = new Child1();
            obj1.m1();
            obj1.m2();
            Child2 obj2 = new Child2();
            obj2.m1();
            obj2.m2();
    }
}
```

**Example of abstract class:-**

```
abstract class Animal
{
    public abstract void sound();
    public void eat()
    {
            System.out.println("Animal eats.");
    }
    public void walk()
    {
            System.out.println("Animal walk");
    }
}
class Cat extends Animal
{
    public void sound()
    {
            System.out.println("Cat meaws.");
    }
}

class Dog extends Animal
{
    public void sound()
    {
            System.out.println("Dog barks");
    }
}
class Tiger extends Animal
{
    public void sound()
    {
            System.out.println("Tiger roars");
    }
}
class Demo
{
    public static void main(String[] args)
    {
            Cat obj1 = new Cat();
            obj1.sound();
            obj1.eat();
            obj1.walk();
            Dog obj2 = new Dog();
            obj2.sound();
            obj2.eat();
            obj2.walk();
            Tiger obj3 = new Tiger();
            obj3.sound();
            obj3.eat();
            obj3.walk();
    }
}
```

## 2. interface :-

- An interface in java is an abstract type i.e. used to describe the behaviours (methods) that a class must implement.
- An interface is a reference type just similar to a class.
- An interface is a collection of abstract methods which is used to achieve 100% abstraction before JDK 1.8.
- In JDK 1.8 java introduced default and static methods in it as well as nested interface.
- From version 1.9 we can declare private methods inside an interface.

**Declaration of interface (syntax):-**

```
interface Interface_Name
{

}
```

**e.g.**

```
interface RandomAccess
{

}
```

- An interface keyword is used to declare an interface.
- Interface declaration is similar to a class.

## Characteristics of an interface :-

1. An interface is implicitly abstract so we don't need to use abstract keyword in its declaration.
2. An interface does not contain any constructor.
3. We cannot instanciate an interface.
4. All non static methods inside an interface are by default **public and abstract**.
5. Only **static, default and private** method can have implementation.
6. Variable created inside an interface is by default **public static final** and we cannot use it without initialization.
7. We cannot declare any initializer (static and non static block) inside an interface.
8. We can achieve multiple inheritance in an interface (i.e. one interface can extends multiple interfaces).
9. An interface cannot be extended by a class but it is implemented using it.

### ★ An interface is similar to the class in following ways,

1. An interface can contains any number of methods in it.
2. Interface file saved with .java extension, if the name of an interface matching with the filename.
3. After compilation of an interface a bytecode is generated with the .class extension.
4. Convention for an interface is similar to a class, we need to follow the rules of identifier by declaring an interface.

★ **Difference between class and interaface.**

| Class | Interface |
|---|---|
| i. We can instanciate a class. | i. We cannot instanciate an interface. |
| ii. A class can have a constructor. | ii. Interface doesn't contains any constructor. |
| iii. Non static methods in a class can be concrete or abstract. | iii. Non static methods inside an interface are abstract. |
| iv. A class can have static and non static initializers in it. | iv. An interface does not have static and non static initializers in it. |
| v. A class can be extended by another class. | v. An interface is extended by another interace. |
| vi. We cannot achieve multiple inheritance in class. | vi. We can achieve multiple inheritance in interface. |
| vii. We can create protected method inside a class. | vii. We cannot create an protected method inside an interace. |

★ **Implementing an interface:-**

- An interface must be implemented by a class.
- A class uses implements keyword.
- The implements keyword also establish a parent – child relationship between them.
- The implementation class must override all the abstract method of an interface.

**Example:-**

```
interface Arithmetic
{
  int addition(int num1,int num2);
  int multi(int num1,int num2);
}
class Implementation implements Arithmetic
{
  public int addition(int num1,int num2)
  {
        int op = num1+num2;
        return op;
  }
  public int multi(int num1,int num2)
  {
        int op = num1*num2;
        return op;
  }
}
class InterfaceArithmetic
{
  public static void main(String[] args)
  {
        Implementation obj = new Implementation();
        System.out.println(obj.addition(10,20));
        System.out.println(obj.multi(10,20));
  }
}
```

**Example 2:-**

```
interface Vehicle{
  void seatingCapacity();
  void price();
  void mileage();
  void brand();
}
interface Car extends Vehicle{
  void airbag();
  void autoPilot();
}
interface Bike extends Vehicle{
  void engineCapacity();
  void typeBike();
}
class Scorpio implements Car{
  public void seatingCapacity(){
        System.out.println("Seating capacity : 7");
  }
  public void price(){
        System.out.println("Price : 18,00,000 Rs.");
  }
  public void mileage(){
        System.out.println("Mileage : 15km/l");
  }
  public void brand(){
        System.out.println("Brand : Mahindra Scorpio");
  }
  public void airbag(){
        System.out.println("Aigbags : 6");
  }
  public void autoPilot(){
        System.out.println("AutoPilot : No");
  }
}
class Splender implements Bike{
  public void seatingCapacity(){
      System.out.println("Seating capacity : 2");
  }
  public void price(){
        System.out.println("Price : 96,000 Rs.");
  }
  public void mileage(){
        System.out.println("Mileage : 60km/l");
  }
  public void brand(){
        System.out.println("Brand : Hero Splender");
  }
  public void engineCapacity(){
        System.out.println("Engine capacity : 100cc");
  }
  public void typeBike(){
        System.out.println("Bike type : Commutor");
  }
}

class Demo{
  public static void main(String[] args) {
        Splender obj1 = new Splender();
        obj1.seatingCapacity();
```

```
            obj1.price();
            obj1.mileage();
            obj1.brand();
            obj1.engineCapacity();
            obj1.typeBike();
            System.out.println();
            Scorpio obj2 = new Scorpio();
            obj2.seatingCapacity();
            obj2.price();
            obj2.mileage();
            obj2.brand();
            obj2.airbag();
            obj2.autoPilot();
    }
}
```

★ **Extending an interface:-**

- An interface can extends another interface, similarly in a way that a class can extends another class.
- But the only difference is an interface can extends multiple interfaces.
- When we extends one interface to another interface, the extended interface becomes the sub interface (child) and the child interface can inherits all the fields and behaviours from the parent.
- A class cannot extends an interface.
- e.g.

```
interface Parent1{

}
interface Parent2{

}
interface Child extends Parent1,Parent2{

}
```

★ **Multiple inheritance in interface:-**

- We can achieve multiple inheritance in interface because,

1. Non static methods are abstract and doesn't contains any implementation so there is no ambiguity.
2. As interface does not contains any constructors in it that's why there is no ambiguity of super() statement.
3. From JDK 1.8 static methods has implementation ie. Body, static methods are not inherited from one class to another that's why no ambiguity.
4. Private methods can have implementation but they are not inherited as there scope belongs to class.

**Most important things to remind :-**

- Defaults methods inside an interface can have implementation and they are inherited from one interface to another.
- If both the parent interface have same method signature it will create an ambiguity for the child interface which parent method must be inherited (diamond problem).
- To resolve this issue child interface has to override parent default method.

**Note:-**

If we want to invoke the parent default methods we need to use a interface name as a reference.(dot)super keyword and then method name.

**Example:-**

```
interface Parent1
{
  default void m1()
  {
        System.out.println("Parent1");
  }
}
interface Parent2
{
  default void m1()
  {
        System.out.println("Parent2");
  }
}
interface Child extends Parent1,Parent2
{
  default void m1()
  {
        System.out.println("Child");
        Parent1.super.m1();
        Parent2.super.m1();
  }
}
class Demo implements Child
{

}
class InterfaceExample1
{
  public static void main(String[] args)
  {
        Demo obj = new Demo();
        obj.m1();
  }
}
```

**A class can implement more than one interface at a time.**

```
interface Demo1{
  void m1();
}
interface Demo2{
  void m2();
}
class Demo3 implements Demo1,Demo2{
  public void m1(){
        System.out.println("m1()");
  }
  public void m2(){
        System.out.println("m2()");
  }
}
class Demo{
  public static void main(String[] args) {
        System.out.println("main()");
        Demo3 obj = new Demo3();
        obj.m1();
        obj.m2();
  }
}
```

**If a class wants to inherit another class as well as needs to provide implementation for an interface first we should extend a class and then implement an interface.**

```
interface Demo1{
}
class Parent{
}
class Demo3 extends Parent implements Demo1{
}
```

★ **Functional interface:-**

- An interface which contains exactly single abstract method in it is known as functional interface.
- It can contains a n number of other methods.
- We can use @FunctionalInterface annotation for creating a functional interface.
- e.g.

```
@FunctionalInterface
interface Demo
{
    void m1();
    default void m2()
    {
        System.out.println("default m2()");
    }
    public static void m3()
    {
        System.out.println("static m3()");
    }
}
```

- e.g. 1. Function, 2. Predecate, 3.Consumer, 4. Supplier, 5. Comparable, 6. Comparator

**Note:-**

In JDK 1.8 java introduced lambda expression which works only on functional interfaces.

```
e.g.
@FunctionalInterface
interface Arithmetic
{
    int addition(int num1,int num2);
}
class Demo
{
    public static void main(String[] args)
    {
            Arithmetic obj = (num1,num2)->num1+num2;
            System.out.println(obj.addition(20,20));
    }
}
```

★ **Marker interface:-**

- A marker interface is an interface that does not contains any fields or behaviours in it.
- It is used to indicate that a class contains certain properties or behaviours.
- It is used to provide a meta data about class.
- E.g. ArrayList implements RandomAccess, Cloneable, Serializable
- ArrayList indicates that it can perform fast searching of elements by implementing RandomAccess.
- It can be used to create a clone which is indicated by Cloneable interface.
- Marker interface is also called tagged interface or empty interface.
- Implementation for this interface is been provided by virtual machine.

★ **Difference between abstract class and interface**

| | abstract class | | interface |
|---|---|---|---|
| i. | An abstract class must be declare with an abstract modifier. | i. | interface keyword is used to declare an interface. |
| ii. | An abstract class can have abstract and concrete methods in it. | ii. | An interface can have only abstract methods in it before JDK 1.8 |
| iii. | An abstract class contains constructor. | iii. | An interface does not contains any constructor. |
| iv. | It can have static or non static initializers in it. | iv. | It cannot have static or non static initializers in it. |
| v. | Abstract class can have static, non static, final or non final variable. | v. | An interface contains only static final variables. |
| vi. | An abstract class is implemented using aonther class by the using extends keyword. | vi. | An interface is implemented by an another class by using implements keyword. |
| vii. | We cannot achieve multiple inheritance in abstract class. | vii. | We can achieve multiple inheritance in interface. |
| viii. | An abstract class can have protected member. | viii. | An interface does not contains any protected member. |

# Important questions:-

1. **Can we declare an interface as final?**

➔ No, we cannot declare an interface as final, because an interface is meant to be implemented by other classes and making it as final would prevent that. The final keyword is used to indicate that a class cannot be extended.

2. **How we can achieve multiple inheritance in interface?**

➔ We can achieve multiple inheritance in interface because,

1. Non static methods are abstract and doesn't contains any implementation so there is no ambiguity.
2. As interface does not contain constructors in it that's why there is no ambiguity of super() statement.
3. From JDK 1.8 static methods can have implementation i.e.Body, static methods are not inherited from one class to another that's why no ambiguity.
4. Private methods can have implementation but they are not inherited as there scope belongs to class.

3. **Why we cannot create protected method in interface?**

➔ Interfaces are meant to be define public contract that any class can implement, so all methods must be accessible to any class that implements the interface.
Protected methods are meant to be used only within the same package or by subclasses, but interfaces aren't designed with that kind of restriction. They need to be fully accessible to all implemting class.

4. **Purpose of creating private method in interface?**

➔ A private method is just for internal use within the interface to avoid repeating code in default or static methods. It keeps the interface cleaner but is not accessible by the classes that implement the interface.
1. Code Reusability, 2. Cleaner code

5. **Does interface have any super interface?**

➔ Super interface is not an separate entity, it refers to the hierarchy. When an interface is extended or implemented it becomes a super interface of what extended or implemented it.

# ★ Object class:-

- Object class is derived in java.lang package.
- Object class sits at the top of class hierarchy.
- Every class which is created in java i.e. built in or user defined has a direct or indirect relationship between Object class.
- This class contains several important methods in it which every class required, so they inherit it.
- Object class is a super class for classes such as String, StringBuffer, StringBuilder, Exception hierarchy class, Wrapper class, etc.

**Constructor of Object class:-**

```
public class Object
{
        public Object()
        {
                //implementation
        }
}
```

**Note:-**

- Object class contains a no argument constructor.
- Every time when we create an object, the constructor chaining will come towards Object class.
- As it is the root class it does not contains any super() statement or this() statement to stop the constructor chaining.

## ➢ Methods of Object class:-

1. public final native java.lang.Class<?> getClass();
2. public native int hashCode();
3. public boolean equals(java.lang.Object);
4. protected native java.lang.Object clone() throws java.lang.CloneNotSupportedException;
5. public java.lang.String toString();
6. public final native void notify();
7. public final native void notifyAll();
8. public final void wait() throws java.lang.InterruptedException;
9. public final void wait(long) throws java.lang.InterruptedException;
10. public final void wait(long, int) throws java.lang.InterruptedException;
11. protected void finalize() throws java.lang.Throwable;

### 1. public String toString();

- toString() method in Object class is used to display a String representation of an object.
- This method returns a String that includes ClassName@object memory address in hexadecimal form.
- It is always recommended to override toString() to display the data of an object in a meaningful way.

- e.g.

```
class Student
{
        int id;
        String name;
        String edu;
        int yop;

        Student(int id,String name,String edu,int yop)
        {
                this.id = id;
                this.name = name;
                this.edu = edu;
                this.yop = yop;
        }
        @Override
        public String toString()
        {
                return "id : "+id+", Name : "+name+", Education : "+edu+", YOP : "+yop;
        }
}
class ExampleToString
{
        public static void main(String[] args)
        {
                Student obj1 = new Student(1,"Ramesh","BE",2024);
                Student obj2 = new Student(2,"Suresh","BSC",2023);
                Student obj3 = new Student(3,"Ganesh","MCA",2022);
                Student obj4 = new Student(4,"Mukesh","Btech",2021);
                System.out.println(obj1);
                System.out.println(obj2);
                System.out.println(obj3);
                System.out.println(obj4);
        }
}
```

**Note:-**

- The toString() method is overridden in several classes such as String, StringBuffer, Wrapper class.
- In this classes the toString() method prints the data instead of object reference.

## 2. public final native java.lang.Class<?> getClass();

- The getClass() is used to fetch the runtime name of a class.
- This method returns an instance of that particular class.
- e.g.

```
class Student{
    int id;
    String name;
    Student(int id,String name){
            this.id=id;
            this.name=name;
    }
}
class ExampleGetClass{
    public static void main(String[] args) {
            Student obj1 = new Student(1,"Ramesh");
            System.out.println(obj1.getClass());
            String str = new String("Hello");
            System.out.println(str.getClass());
            Integer a = new Integer(10);
            System.out.println(a.getClass());
    }
}
```

## 3. public boolean equals(java.lang.Object);

- equals() method is used to compare the reference of an object and returns true if the references are same or else returns false.
- Equals() method works similar to equality operator(==).
- The equality operator used for comparing both primitive and non primitive data.
- In case of primitive it compares the value.
- In case of non primitive it compares the reference.
- So if you need to compare the data inside an object instead of a reference we have to use equals() and need to override it.

**Note:-**

It is highly recommended to override equals() along with toString() and hashCode().

**Example:-**

```
class ExampleEquals{
    public static void main(String[] args) {
            String str1 = new String("hello");
            String str2 = new String("hello");
            System.out.println(str1==str2);
            System.out.println(str1.equals(str2));
            Demo obj1 = new Demo();
            Demo obj2 = new Demo();
            Demo obj3 = obj2;
            System.out.println(obj1==obj2);
            System.out.println(obj2==obj3);
            System.out.println(obj1.equals(obj2));
    }
}
class Demo{

}
```

**Example:-**

```
class Employee{
    int eid;
    String ename;
    Employee(int eid,String ename){
            this.eid = eid;
            this.ename = ename;
    }
    public boolean euqals(Object obj){
            Employee obj2 = (Employee)obj;
            if((this.eid==obj2.eid)&&(this.ename==obj2.ename))
                    return true;
            return false;
    }
    public String toString(){
            return "Eid : "+eid+", Ename : "+ename;
    }
}
class ExampleEquals2
{
    public static void main(String[] args) {
            Employee obj1 = new Employee(1,"Ramesh");
            Employee obj2 = new Employee(2,"Suresh");
            System.out.println(obj1==obj2);
            System.out.println(obj1.euqals(obj2));
            Employee obj3 = new Employee(1,"Ramesh");
            System.out.println(obj1.euqals(obj3));
    }
}
```

★ **Difference between equality operator and equals().**

| Equality operator | equals() |
|---|---|
| i. Equality operator is an operator. | i. equals() is a method. |
| ii. It is use to compare primitive as well as non primitive. | ii. It is use to compare non primitive data. |
| iii. We cannot overload equality operator (==) | iii. We can overload equals() and change its implementation. |

## 4. public native int hashCode();

- The hashCode() in java is very important when we are working with hashbased collections such as Hashtable, HashSet, HashMap.
- It returns an integer value that represents an hashCode of an object.
- We should override this method for meaningful use in collections.

★ **Why we should override hashCode() ?**

- If we are overriding equals() we should also override hashCode() because these two methods are linked together.
- If two objects are equals then they must have same hashCode value.
- If two objects are not equals they might have same hashCode value which leads to hashCollision.

- **e.g.**

```
class Employee{
    int eid;
    String ename;
    Employee(int eid,String ename){
            this.eid = eid;
            this.ename = ename;
    }
    public int hashCode(){
            return this.eid;
    }
    public boolean equals(Object obj){
            Employee obj2 = (Employee)obj;
            if((this.eid==obj2.eid)&&(this.ename==obj2.ename))
                    return true;
            return false;
    }
}
class ExampleHashCode{
    public static void main(String[] args) {
            Employee obj1 = new Employee(1,"Ramesh");
            Employee obj2 = new Employee(1,"Ramesh");
            Employee obj3 = new Employee(3,"Mahesh");
            System.out.println(obj1.equals(obj2));
            System.out.println(obj1.hashCode());
            System.out.println(obj2.hashCode());
            System.out.println(obj3.hashCode());

            String str1 = new String("hello");
            String str2 = new String("hello");
            System.out.println(str1.equals(str2));
            System.out.println(str1.hashCode());
            System.out.println(str2.hashCode());
    }
}
```

## 5. protected void finalize() throws java.lang.Throwable;

- The finalize() belongs to Object class and it is called by Garbage Collector before an object is destroyed.
- finalize() does all the cleanup activities such as closing resources that are streams, sockets, file, DB connections, etc.

★ **Why we should override finalize()?**

- It is used to make sure that all the resources are closed properly before an object is destroyed.
- It is unpredictable that the finalize() will be called by Garbage Collector any time and its not reliable way to manage resources.
- So from JDK 1.9 java deprecated the use of finalize() method.

**Example:-**

```
import java.util.Scanner;
class Student{
	String sname;
	Student(String sname){
		this.sname = sname;
	}
	@Override
	public String toString(){
		return sname;
	}
	@Override
	@SuppressWarnings("removal")
	public void finalize() throws Throwable{
		System.out.println("Object got destroyed");
	}
}
class FinalizeExample
{
	static Scanner sc = new Scanner(System.in);
	public static void main(String[] args) throws Throwable{
		System.out.println("Enter the name");
		String name = sc.next();
		Student obj = new Student(name);
		System.out.println(obj);
		obj.finalize();
		obj = null;
	}
}
```

**Approach 1:-**

- In the above example we are calling finalize() explicitly to destroy an unreferenced object.
- This approach is similar to a destructor like other programming languages.

**Approach 2:-**

```
//Approach 2 Example
class FinalizeExample2
{
	static Scanner sc = new Scanner(System.in);
	public static void main(String[] args) throws Throwable{
		System.out.print("Enter the name");
		String name = sc.next();
		Student obj = new Student(name);
		obj = new Student("Ramesh");
		System.gc();
	}
}
```

- Garbage collector is Daemon thread which works at background and helps a non-Daemon thread (main).
- Garbage collector invokes finalize() for cleanup activity before object is been destroyed.
- All of this happens implicitly.
- We can invoke a Garbage Collector explicitly with the help of System.gc()
- This gc() will invoke the garbage collector and garbage collector will invoke / calls finalize().

**e.g.**

```
import java.util.Scanner;
class Student{
    String sname;
    Student(String sname){
        this.sname = sname;
    }
    @Override
    public String toString(){
        return sname;
    }
    @Override
    @SuppressWarnings("removal")
    public void finalize() throws Throwable{
        System.out.println("Object got destroyed");
    }
}
class FinalizeExample2
{
    static Scanner sc = new Scanner(System.in);
    public static void main(String[] args) throws Throwable{
        System.out.print("Enter the name");
        String name = sc.next();
        Student obj = new Student(name);
        System.gc();
    }
}
```

## 6. protected native java.lang.Object clone() throws java.lang.CloneNotSupportedException;

- clone() is used to create a shallow copy of an object of the same class.
- This way of creating object is faster as compare to new keyword.

**Advantages:-**

- We don't need to write lengthy and repetatitve code.
- It is an easiest and efficient way to create an object (clone).
- Its most widely used to create a clone of an array.

**Steps to clone an object:-**

1. A class must implement Cloneable interface.
2. We need to override clone() and handle CloneNotSupportedException.
3. Before creating a clone we must have an existing object.

**E.g.**

```
class Student implements Cloneable
{
    String name;
    String nPlace;
    Education edu;
    Student(String name,String nPlace,Education edu)
    {
        this.name = name;
        this.nPlace = nPlace;
        this.edu = edu;
    }
    public String toString()
    {
        return "Name : "+name+", Native Place : "+nPlace+"Education : "+edu;
    }
```

```
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }

}
class Education
{
    String graduation;
    int yop;
    Education(String graduation,int yop)
    {
        this.graduation = graduation;
        this.yop = yop;
    }
    public String toString()
    {
        return "Education : [ Graduation : "+graduation+", Year of pass : "+yop+" ]";
    }
}
class CloneExample
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Student orgObj =  new Student("Ramesh","Pune",new Education("BE",2024));
        System.out.println(orgObj);
        Student copy = (Student)orgObj.clone();
        System.out.println(copy);
        orgObj.edu.graduation = "MCA";
        System.out.println(orgObj);
        System.out.println(copy);
    }
}
```

**Output:-**

```
Name : Ramesh, Native Place : PuneEducation : Education : [ Graduation : BE, Year of pass : 2024 ]
Name : Ramesh, Native Place : PuneEducation : Education : [ Graduation : BE, Year of pass : 2024 ]
Name : Ramesh, Native Place : PuneEducation : Education : [ Graduation : MCA, Year of pass : 2024 ]
Name : Ramesh, Native Place : PuneEducation : Education : [ Graduation : MCA, Year of pass : 2024 ]
```

7. **wait()**
   - The wait() method in Object class is an overloaded method.
   - This method is used for inter thread communication in multithreading.
   - This method causes the current thread to wait for a specific period of time based on the argument passed / until the current execution thread complete its execution.

**Note:-**

Once the executing thread complete its execution it will notify a specific thread or notify all the waiting threads.

8. **notify()**
   - notify() method wakes up the current single thread i.e. waiting for the execution on that object.

9. **notifyAll()**
   - notifyAll() method wakes up all the waiting thread for the execution on that object.

**Note:-**

The above 5 methods are used for multithreading and whenever we use them we have to handle InterruptedException.

# ★ Exception Handling :-

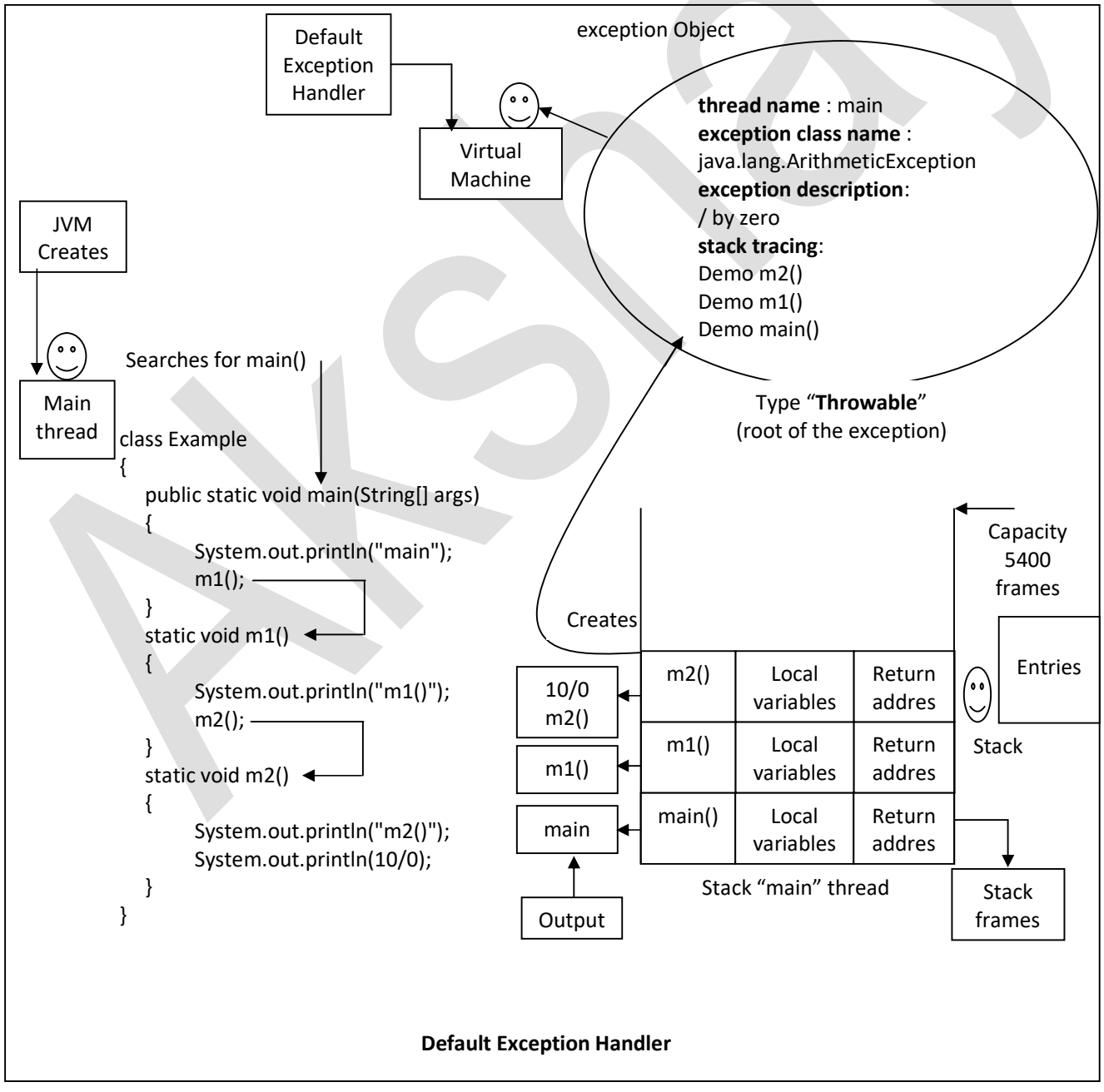
- Exception handling is one of the most important feature of java that allows us to handle the exceptions which are casued by some unwanted events which makes java a robust programming language.

### ★ Exception:-

- An exception is an unwanted event that occurs during the execution of a program that disturbs normal termination of a program.
- When an exception occurs the program execution is terminated abnormally.
- In this case we get a system generated message, which is not at all understandable by user what went wrong.
- But there is good thing about exception that they can be handled.

### ★ Why exception occurs?

- If we try to open a non existing file.
- If we try to provide some bad input data.
- Because of bad network connection.
- If we are declaring the statement which is not logically correct, etc.




**Default Exception Handler**

**Output of default exception handler:-**

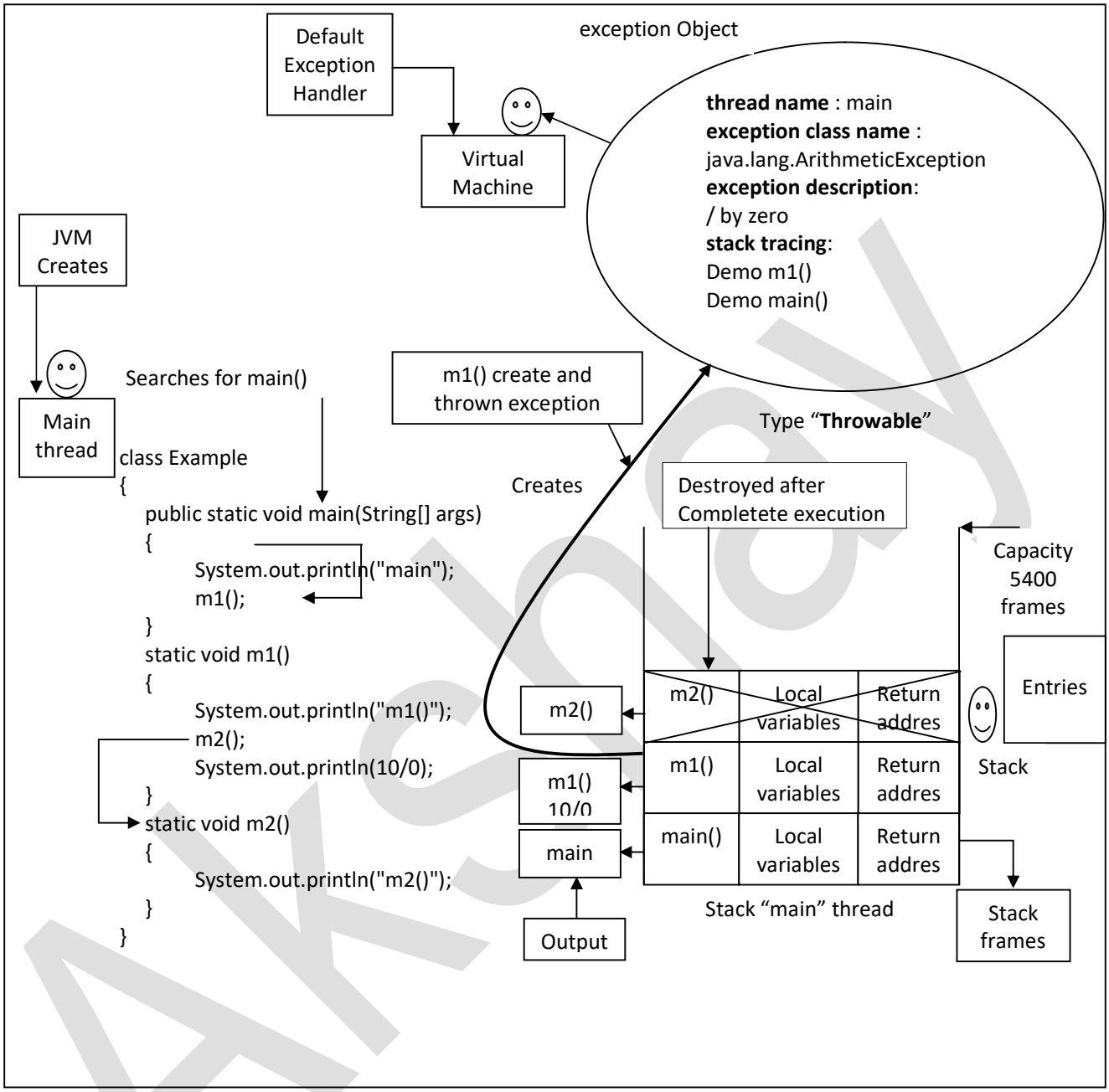
```
main
m1()
m2()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Example1.m2(Example1.java:16)
        at Example1.m1(Example1.java:11)
        at Example1.main(Example1.java:6)
```

Tracing and entries

- At the time of execution whenever an execution occurs inside a method, that method is responsible for creating an exception object.
- The exception object which is been created is of Throwable type.
- The exception object consists of,
  1. Thread name
  2. Exception class name
  3. Exception description
  4. Stack tracing:- Stack tracing is nothing but the methods where exception propagates (flow).
- Once an exception object is created it is been given to the virtual machine.
- Firstly the virtual machine checks with same method has an exception handling mechanism or not.
- Then it propagates the exception object to calling method and checks the same.
- If no method has an exception handling mechanism then eventually the exception is handled by default exception handler.

**Note:-**

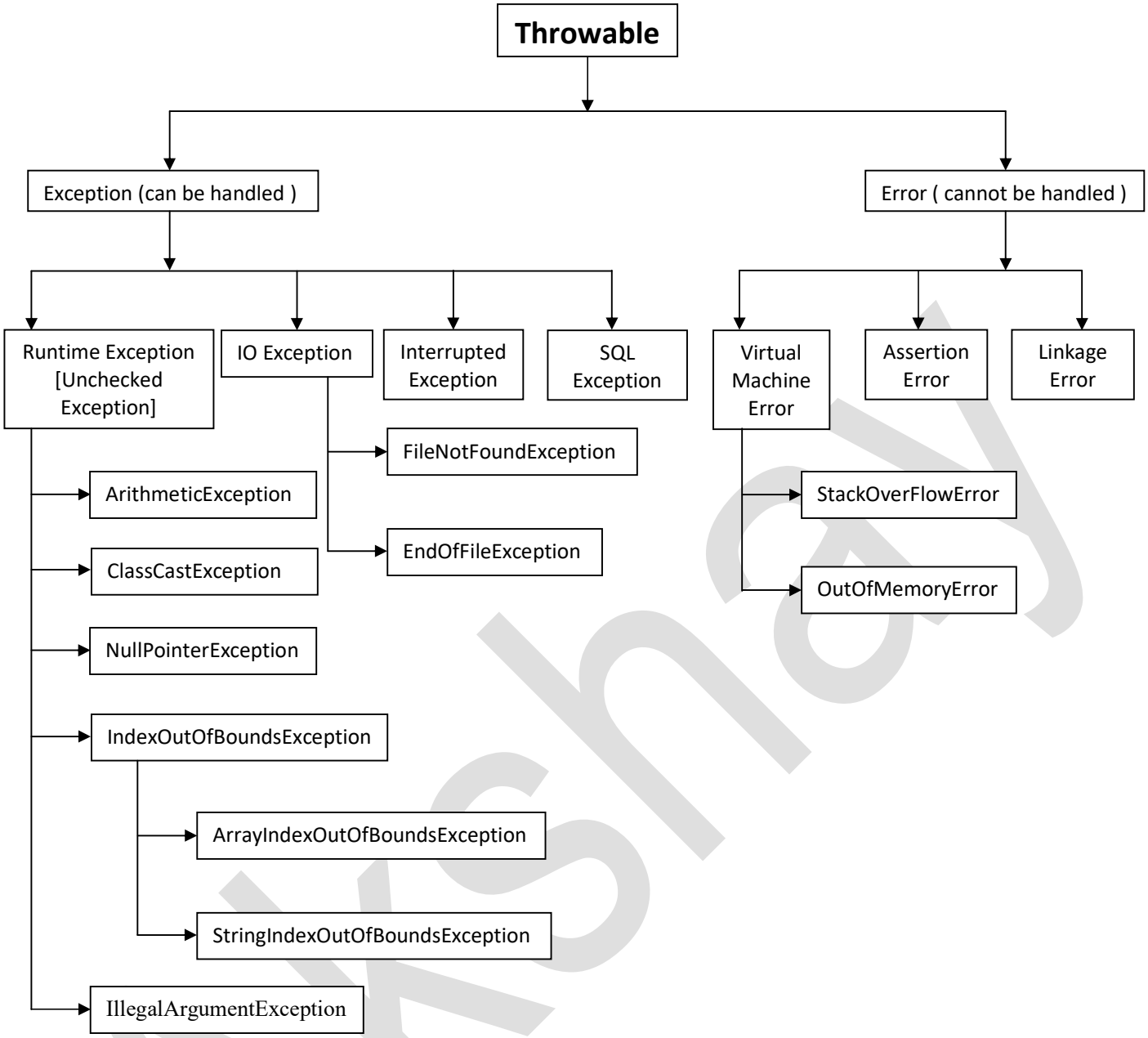
If an exception object is handled by default exception handler then 100% its an abnormal termination.




**Output of default exception handler:-**

```
main
m1()
m2()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Example1.m1(Example1.java:12)
        at Example1.main(Example1.java:6)
```

Tracing and entries



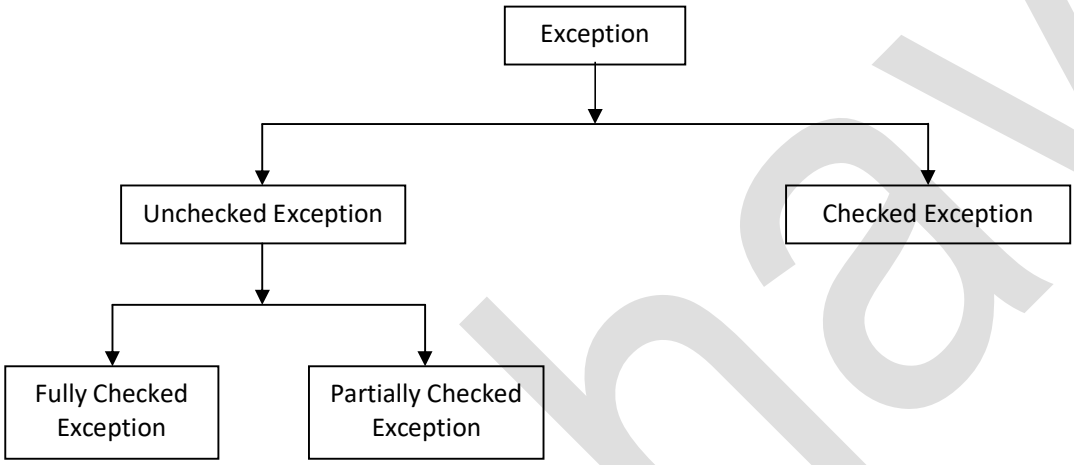

★ **Difference between exception and error:-**

| Exception | Error |
|---|---|
| i. An issue in a program that prevent the normal execution of program is known as “Exception”. | i. An indication of unexpected condition that occurs due to lack of system resource is known as “Error”. |
| ii. It occurs because of an unwanted events (Logical, connection, wrong input, etc.) | ii. It occurs because of lack of system resource (memory, bytecode, etc.) |
| iii. They are recoverable [can be handled]. | iii. They are not recoverable [can not be handled]. |
| iv. It can be classified as checked and unchecked exception. | iv. It is classified as unchecked exception. |
| v. We can handle them using try, catch and throws. | v. There is no way to handle error in programs. |
| vi. e.g. NullPointerException, ArithmeticException | vi. e.g. AssertionError, Linkage Error, StackOverFlowError |

## ★ Exception Handling:-

1. If an exception occurs which is not handled by a programmer then program execution gets terminated abnormally.
2. To avoid this abnormal termination we need to create an alternative way for a normal termination.
3. A message which is displayed when an exception occurs is not user friendly.
4. Our user unable to understand what went wrong, in order to let our user know the reason why this unwanted event occurs, we need to handle exception by user friendly warning message.
5. It is also used to perform some cleanup task before a program gets terminated.
   [cleanup task such as → closing objects, closing streams, connection, servers, DB, etc.]

### Classificaiton of Exception:-




### 1. Checked Exception:-

- Exceptions checks by the compiler at the compile time for the normal termination of a program at runtime is known as **"Checked Exception"**.

**Note:-**

i. Compiler is not aware about an exception but it checks whether the programmer has handled it or not.

ii. All exceptions other than RunTimeException are Checked Exception.

e.g. 1

```
class Example1
{
        public static void main(String[] args)
  {
                String str = "Hello";
        for(int i=0;i<str.length();i++)
        {
                System.out.println(str.charAt(i));
                Thread.sleep(1000);
        }
  }
}
```

**Output:-**

```
Example1.java:9: error: unreported exception InterruptedException; must be caught or declared to be thrown
                Thread.sleep(1000);
```

i. In the last program we are using Thread.sleep() which throws “InterruptedException” which need to be handled.
ii. “InterruptedException” is an example of the checked exception.

E.g. 2

```
import java.io.*;
class Example2
{
        public static void main(String[] args)
        {
                FileReader file = new FileReader("abc.txt");
        }
}
```

**Output:-**

```
Example2.java:6: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
          FileReader file = new FileReader("abc.txt");
```

- **Checked Exception is classified into two types:-**
  i. Fully Checked Exception
  ii. Partially Checked Exception

**i. Fully Checked Exception:-**

- In fully checked exception both parent and child class exception is checked by the compiler.
- e.g.

| Parent | | Child |
|---|---|---|
| Exception | & | IOException |
| Exception | & | InterruptedException |

**ii. Partially Checked Exception:-**

- In partially checked exception only parent class exception is checked by the compiler.
- e.g.
  1. Exception & RuntimeException
  2. Throwable & Error

**2. Unchecked Exception:-**

- The exception which are checked or caught at runtime is known as unchecked exception.
- These exceptions are not checked ar the compile time, so compiler is not aware about them, but it is responsibility of a programmer to handle them for a normal termination of a program.
- A “RuntimeException” is also known as the UncheckedException.
- e.g.

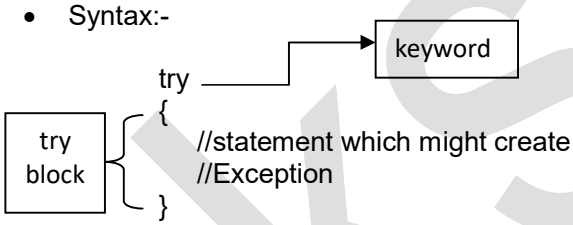
1. System.out.println(10/0); → ArithmeticException
2. 
```
class Demo
{
  public static void main(String[] args)
  {
        String str = "Akshay";
        for(int i=0;i<=str.length();i++)
        {
                System.out.println(str.charAt(i));
        }
  }
}
```

Output:-
A
k
s
h
a
y
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Index 6 out of bounds for length 6
at java.base/jdk.internal.util.Preconditions$1.apply(Preconditions.java:55)
at java.base/jdk.internal.util.Preconditions$1.apply(Preconditions.java:52)
at java.base/jdk.internal.util.Preconditions$4.apply(Preconditions.java:213)
at java.base/jdk.internal.util.Preconditions$4.apply(Preconditions.java:210)
at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:98)
at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:106)
at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:302)
at java.base/java.lang.String.checkIndex(String.java:4832)
at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:46)
at java.base/java.lang.String.charAt(String.java:1555)
at Demo.main(Demo.java:7)

## ★ Exception handling using try{} catch() block

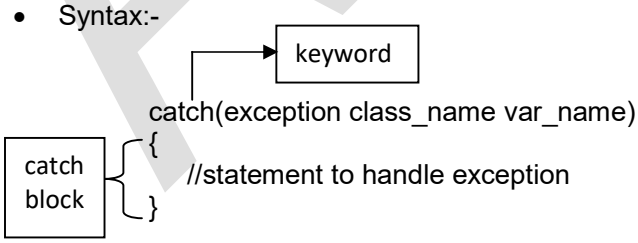
### ➢ try:-

- try is keyword in java.
- The try block contains the statement because of which an exception might occurs.
- Number of statement inside a try block must be less because once an exception occurs statement after it will be skipped inside try block.
- A try block is followed by catch block or finally block.
- Syntax:-




### ➢ catch:-

- catch is keyword.
- A catch block is where we handled exception.
- catch block must follow a try block or another catch block.
- It contains the statements which gets executed when an exception occurs inside in try block.
- A try block can have a multiple catch blocks associated with it.
- Syntax:-



- **try, catch syntax:-**

```
try
{
    //statement which might create exception
}
catch(exception class_name var_name)
{
    //statment to handle exception
}
```

**Normal termination of program:-**

```
class Demo{
    public static void main(String[] args) {
            try{
                    System.out.println(10/0);
            }
            catch(ArithmeticException ae){
                    System.out.println("Exception block");
            }
    }
}
```

**Output:-**

```
Exception block
```

**Abnormal termination of program:-**

```
class Demo{
    public static void main(String[] args) {
            System.out.println(10/0);
    }
}
```

**Output:-**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Demo.main(Demo.java:5)
```

**E.g. 1**

```
class Demo1{
    public static void main(String[] args) {
            System.out.println("Hello 1");
            System.out.println(10/0);
            try{
                    System.out.println(10/0);
            }
            catch(ArithmeticException ae){
                    System.out.println("ArithmeticException handled");
            }
            System.out.println("Hello 2");
    }
}
```

**Output:-**

```
Hello 1
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Demo1.main(Demo1.java:6)
```

**Note:-**

An exception occurred outside the try block cannot be handled it will terminate the execution abnormally.

## ⋆ Multiple catch blocks:-

- A single try block can have any number of catch block associated with it.
- If no exception occurs inside the try block all the corresponding catch blocks are skipped.
- The Generic (parent type) catch block can handle all type of exceptions.
- Corresponding catch block is executed for specific type of execution,
  i. catch(ArithmeticException ae)
     These catch block can handle only arithmetic exception.

  ii. catch(RuntimeException re)
     These catch block can handle any runtime exception such as:- ClassCastException (CCE),
     NullPointerException (NPE), IndexOutOfBoundsException (IOOBE)

  iii. catch(Exception e)
     These catch block can handle any type of exception such as :- RuntimeException, IOExceptipon, SQLException, FileNotFoundException.

**Note:-**

1. Whenever we creating multiple catch blocks, the other catch block must be from child to parent.
2. If we declare a parent catch block that is as generic catch block can alone handle all type child exceptions.

✓
```
catch(ArithmeticException ae)
catch(RuntimeException re)
catch(Exception e)
catch(Throwable t)
```

✓
```
catch(RuntimeException re)
catch(InterruptedException ie)
catch(Exception e)
```

✓
```
catch(InterruptedException ie)
catch(RuntimeException re)
catch(Exception e)
```

✓
```
catch(Exception e)
catch(Throwable t)
```

X
```
catch(Exception e)
catch(ArithmeticException ae)
catch(NullPointerException npe)
```

**E.g. 1**

```
class Demo2
{
	public static void main(String[] args)
	{
		System.out.println("Hiiiii");
		try
		{
			System.out.println("try");
			System.out.println(args[1]);
		}
		catch(RuntimeException re)
		{
			System.out.println("catch 1");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("catch 2");
		}
		System.out.println("Bye");
	}
}
```

**Output:-**

```
Demo2.java:14: error: exception ArrayIndexOutOfBoundsException has already been caught
		catch(ArrayIndexOutOfBoundsException e)
```

**E.g.2**

```
class Demo3
{
	public static void main(String[] args) {
		System.out.println("Hiiiii");
		try{
			System.out.println("try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae){
			System.out.println("catch 1");
			System.out.println(10/0);	//line 14
		}
		catch(Exception e){
			System.out.println("catch 2");
		}
		System.out.println("bye");
	}
}
```

**Output:-**

```
Hiiiii
try
catch 1
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Demo3.main(Demo3.java:14)
```

**E.g. 3**

```
import java.io.*;
class Demo4{
	public static void main(String[] args) {
		System.out.println("Hiiiiii");
		try{
			System.out.println("try");
			FileReader file = new FileReader("abc.txt");
		}
		catch(RuntimeException re){
			System.out.println("catch 1");
		}
		catch(IOException ie){
			System.out.println("catch 2");
		}
		catch(Exception e){
			System.out.println("catch 3");
		}
		System.out.println("Bye");
	}
}
```

**Output:-**

```
Hiiiiii
try
catch 2
Bye
```

**E.g.4**

```
class Demo5
{
	public static void main(String[] args) {
		System.out.println("Hiiii");
		try{
			System.out.println("Outer try");
			try{
				System.out.println("inner try");
				System.out.println(10/0);
			}
			catch(ArrayIndexOutOfBoundsException ae)
			{
				System.out.println("outer catch");
			}
		}
		System.out.println("bye");
	}
}
```

**Output:-**

```
Demo5.java:5: error: 'try' without 'catch', 'finally' or resource declarations
		try{
```

**E.g. 5**

```
class Demo6{
	public static void main(String[] args) {
		System.out.println("Hiiii");
		try{
			System.out.println("Outer try");
			System.out.println(10/0);
			try{
				System.out.println("Inner try");
				System.out.println(10/0);
			}
			catch(Exception e){
				System.out.println("Inner catch");
			}
		}
		catch(RuntimeException re){
			System.out.println("Outer catch");
		}
		System.out.println("bye");
	}
}
```

**Output:-**

```
Hiiii
Outer try
Outer catch
Bye
```

**E.g.6**

```
class Demo7{
	public static void main(String[] args) {
		System.out.println("Hiiiii");
		try{
			System.out.println("outer try");
			try{
				System.out.println("inner try");
				System.out.println(10/0);
			}
			catch(ArithmeticException ae){
				System.out.println("inner catch");
				String a = null;
				System.out.println(a.isEmpty());
			}
		}
		catch(NullPointerException npe){
			System.out.println("outer catch");
		}
		System.out.println("bye");
	}
}
```

**Output:-**

```
Hiiiii
outer try
inner try
inner catch
outer catch
bye
```

**E.g.7**

```
class Demo8
{
	public static void main(String[] args)
	{
		System.out.println("Hiiiii");
		try
		{
			System.out.println("Outer try");
			Thread.sleep(2000);
		}
		catch(InterruptedException ie)
		{
			System.out.println("Outer catch");
			try
			{
				System.out.println("Inner try");
				System.out.println(10/0);
			}
			catch(ArithmeticException ae)
			{
				System.out.println("Inner catch");
			}
		}
		System.out.println("Bye");
	}
}
```

**Output:-**

Hiiiii
Outer try → here it will pause for 2000 miliseconds and then execute next
Bye

**E.g.8**

```
class MyThread extends Thread
{
	public void run()
	{
		try
		{
			Thread.sleep(2000);
		}
		catch(InterruptedException ie)
		{
			System.out.println("InterruptedException handled");
		}
		for(int i=1;i<=10;i++)
			System.out.println("Ramesh"+i);
	}
}
class Demo9
{
	public static void main(String[] args)
	{
		MyThread t1 = new MyThread();
		t1.start();
		for(int i=1;i<=10;i++)
		{
			System.out.println("Main"+i);
			if(i==9)
```

```
                    t1.interrupt();
                }
        }
}
```

**Output:-**

```
Main1
Main2
Main3
Main4
Main5
Main6
Main7
Main8
Main9
Main10
InterruptedException handled
Ramesh1
Ramesh2
Ramesh3
Ramesh4
Ramesh5
Ramesh6
Ramesh7
Ramesh8
Ramesh9
Ramesh10
```

★ **finally:-**

- finally is a keyword.
- finally block contains all crucial statements that must be executed whether exceptions occurred or not.
- finally block follows catch block or try block.
- Syntax:-

```
finally
{
    //crucial statements that must be
    //executed
}
```

- **e.g.**

```
class FinallyExample
{
    public static void main(String[] args)
    {
            System.out.println("DB Connection");
            System.out.println("Hello");
            try
            {
                    System.out.println("try block");
                    System.out.println(10/0);
            }
            finally
            {
                    System.out.println("Crucical statements");
                    System.out.println("DB Connection close");
            }
    }
}
```

**Output:-**

```
DB Connection
Hello
try block
Crucical statements
DB Connection close
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at FinallyExample.main(FinallyExample.java:10)
```

**Note:-**

finally block contains the crucial statements such as, closing connection, objects, streams, database, servers, etc.

## ★ Characteristics of finally block:-

1. finally block is associated with a try block.
2. We cannot use finally without try.
3. finally block is optional because try and catch block is sufficient to handle the exception.
4. However, if we have specified a finally block it will get executed after the execution of try and catch block.
5. In a normal case if there is no exception in try block then finally block gets executed directly by skipping the catch block.
6. However when an exception occurs then catch block get executed first and then finally block gets executed.
7. If an exception occurs inside a finally block it behaves exactly like any other exception occurred in block.

**Statements inside an finally block gets executed even if try-catch block contains control transfer statements such as return, break or continue.**

**In following cases the finally block doesn't get executed,**

i. **Death of thread**
ii. **System.exit(0);**
iii. **When an exception occurs an outside of try block.**

**E.g.**

```
class MyThread extends Thread{
  public void run(){
        try{
              System.out.println("try");
              System.out.println(10/0);
        }
        catch(ArithmeticException ae){
              System.out.println("catch block");
              try{
                    Thread.sleep(5000);
              }
              catch(InterruptedException ie){
                    System.out.println("InterruptedException handled");
              }
        }
        finally{
              System.out.println("finally block");
        }
  }
}
class FinallyExample1{
  public static void main(String[] args) throws InterruptedException{
        MyThread t1 = new MyThread();
        t1.setDaemon(true);
        t1.start();
        for(int i=1;i<=5;i++){
              System.out.println("main"+i);
              Thread.sleep(500);
        }
  }
}
```

**Output:-**
try
catch block
main1
main2
main3
main4
main5

**Note:-**

- "main" is a Non Daemon thread.
- A Non Daemon thread is a high priority thread.
- A Daemon thread is a low priority thread, which is running at the background and support Non Daemon Thread in execution.
- E.g.
  gc() :- Daemon thread
  main :- Non Daemon thread
- If a Non Daemon thread complete its execution it goes to dead state and even if the Daemon thread has left out the execution still it goes to dead state.

**E.g.1**

```
class FinallyExample2
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try");
			System.out.println("10/0");
		}
		catch(ArithmeticException ae)
		{
			System.out.println("ArithmeticException handled");
		}
		finally
		{
			System.out.println("finally block");
		}
	}
}
```

**Output:-**
try
10/0
finally block

**E.g.2**

```
class FinallyExample3
{
	public static void main(String[] args)
	{
		System.out.println(10/0);
		try
		{
			System.out.println("try");
			System.out.println("10/0");
		}
		catch(ArithmeticException ae)
		{
			System.out.println("ArithmeticException handled");
		}
		finally
		{
			System.out.println("finally block");
		}
	}
}
```

**Output:-**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at FinallyExample3.main(FinallyExample3.java:5)
```

**E.g.3**

```
class FinallyExample4
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try");
			System.out.println("10/0");
		}
		catch(ArithmeticException ae)
		{
			System.out.println("ArithmeticException handled");
			System.exit(0);
		}
		finally
		{
			System.out.println("finally block");
		}
	}
}
```

**Output:-**

```
try
10/0
finally block
```

**E.g.4**

```
class FinallyExample5
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("ArithmeticException handled");
			System.exit(0);
		}
		finally
		{
			System.out.println("finally block");
		}
	}
}
```

**Output:-**

```
try
ArithmeticException handled
```

**E.g.5**

```
class FinallyExample6
{
	public static void main(String[] args)
	{
		System.out.println("Execution starts");
		m1();
		System.out.println("Execution ends");
	}
	public static void m1()
	{
		try
		{
			System.out.println("try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("catch block");
			return;
		}
		finally
		{
			System.out.println("finally block");
		}
	}
}
```

**Output:-**

```
Execution starts
try
catch block
finally block
Execution ends
```

**E.g.6**

```
class FinallyExample7
{
	public static void main(String[] args)
	{
		for(int i=1;i<=5;i++)
		{
			try
			{
				System.out.println("try block"+i);
				if(i==2)
					System.out.println(10/0);
			}
			catch(ArithmeticException ae)
			{
				System.out.println("catch block");
				break;
			}
			finally
			{
				System.out.println("finally block");
			}
		}
	}
}
```

**Output:-**

```
try block1
finally block
try block2
catch block
finally block
```

**E.g.7**

```
class FinallyExample8
{
	public static void main(String[] args)
	{
		for(int i=1;i<=5;i++)
		{
			try
			{
				System.out.println("try block"+i);
				if(i==2)
					System.out.println(10/0);
			}
			catch(ArithmeticException ae)
			{
				System.out.println("catch block");
				break;
			}
			finally
			{
				System.out.println("finally block");
				continue;
			}
		}
	}
}
```

**Output:-**

```
try block1
finally block
try block2
catch block
finally block
try block3
finally block
try block4
finally block
try block5
finally block
```

**Note:- [ When finally block doesn't get execute]**

- When there is an infinite loop finally block doesn't get executed because control keeps on executing the same loop again and again and just because there is no condition to terminate it creates a situation same as deadlock.

**E.g. 8**

```
class FinallyExample9
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("catch");
			for (; ; );
		}
		finally
		{
			System.out.println("finally block");
		}
	}
}
```

**Output:-**

```
try
catch
```

★ **Deadlock:-**

- A thread waiting for another thread forever is known as deadlock.
- In the below example "join()" is used to pause the execution of current thread for duration forever (infinite duration).
- E.g.

```
class DeadlockExample
{
    public static void main(String[] args) throws InterruptedException
    {
		try
		{
			System.out.println("try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("catch");
			Thread.currentThread().join();
		}
		finally
		{
			System.out.println("finally block");
		}
    }
}
```

**Output:-**

```
try
catch
```

## Control flow of try-catch-finally (nested)

**E.g. 1**

```
class NestedExample1
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("outer try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("outer catch");
			try
			{
				System.out.println("inner try");
				System.out.println(10/0);
			}
			catch(NullPointerException npe)
			{
				System.out.println("inner catch");
			}
		}
		finally
		{
			System.out.println("outer finally");
		}
	}
}
```

**Output:-**

```
outer try
outer catch
inner try
outer finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at NestedExample1.main(NestedExample1.java:16)
```

**E.g. 2**

```
class NestedExample2
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("outer try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("outer catch");
			try
			{
				System.out.println("inner try");
				System.out.println(10/0);
			}
			catch(NullPointerException npe)
			{
				System.out.println("inner catch");
			}
			finally
			{
				System.out.println("inner finally");
			}
		}
		catch(Exception e)
		{
			System.out.println("Exception");
		}
		finally
		{
			System.out.println("outer finally");
		}
	}
}
```

**Output:-**

```
outer try
outer catch
inner try
inner finally
outer finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at NestedExample2.main(NestedExample2.java:16)
```

**E.g. 3**

```
class NestedExample3
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("try");
			System.out.println(10/0);
		}
		catch(ArithmeticException ae)
		{
			System.out.println("catch");
		}
		finally
		{
			System.out.println("finally");
			System.out.println(10/0);
		}
	}
}
```

**Output:-**

```
try
catch
finally
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at NestedExample3.main(NestedExample3.java:17)
```

## ★ Throwable:-

- Throwable is a root class of exception in java.
- All exception classes are derived from this “Throwable” class.
- Every exception object which is been created by a programmer or thrown by a method must be a “Throwable” type.
- There are two main subclass of Throwable
  1. Exception
  2. Error

**Methods of Throwable:-**

1. getMessage()
2. printStackTrace()
3. toString()

**1. getMessage():-**

- This method is used to get the description of exception.
- E.g. 1

```
class MethodExample{
    public static void main(String[] args) {
            try{
                    System.out.println("try");
                    System.out.println(args[1]);
            }
            catch(ArrayIndexOutOfBoundsException e) {
                    System.out.println("catch");
                    System.out.println(e.getMessage());
            }
    }
}
```

**Output:-**

```
try
catch
Index 1 out of bounds for length 0
```

E.g.2

```
class MethodExample2{
    public static void main(String[] args) {
            try{
                    System.out.println("try");
                    Object obj = new String();
                    StringBuffer sb = (StringBuffer)obj;
            }
            catch(ClassCastException e){
                    System.out.println("catch");
                    System.out.println(e.getMessage());
            }
    }
}
```

**Output:-**

```
try
catch
```

class java.lang.String cannot be cast to class java.lang.StringBuffer (java.lang.String and java.lang.StringBuffer are in module java.base of loader 'bootstrap')

2. **toString():-**
   - toString() belongs to Object class which is responsible for printing object reference, but the reference is of no use of programmer.
   - So it is always recommended to override it.
   - In Throwable class toString() is overrided and it displays ExceptionClassName with its description.

**E.g.1**

```
class ThrowableExample1
{
    public static void main(String[] args)
    {
            try
            {
                    System.out.println("try");
                    String str = null;
                    System.out.println(str.isEmpty());
            }
            catch(NullPointerException npe)
            {
                    System.out.println("catch");
                    System.out.println(npe);
                    System.out.println(npe.toString());
            }
    }
}
```

Same output

**Output:-**

```
try
catch
java.lang.NullPointerException: Cannot invoke "String.isEmpty()" because "<local1>" is null
java.lang.NullPointerException: Cannot invoke "String.isEmpty()" because "<local1>" is null
```

**E.g.2**

```
import java.util.Scanner;
class ThrowableExample2
{
    public static void main(String[] args)
    {
            try
            {
                    System.out.println("try");
                    int a = new Scanner(System.in).nextInt();
            }
            catch(Exception e)
            {
                    System.out.println("catch");
                    System.out.println(e.toString());
            }
    }
}
```

**Output:-**

```
try
a
catch
java.util.InputMismatchException
```

3. **printStackTrace():-**
   - This method displays ExceptionClassName, description and stack entries (i.e method sigtnature) where exception object propagates.
   - **E.g.**

```
class ThrowableExample3
{
    public static void main(String[] args)
    {
            m1();
    }
    public static void m1()
    {
            m2();
    }
    public static void m2()
    {
            try{
                    System.out.println(10/0);
            }
            catch(ArithmeticException e)
            {
                    e.printStackTrace();
            }
    }
}
```

**Output:-**

```
java.lang.ArithmeticException: / by zero
    at ThrowableExample3.m2(ThrowableExample3.java:14)
    at ThrowableExample3.m1(ThrowableExample3.java:9)
    at ThrowableExample3.main(ThrowableExample3.java:5)
```

Stack entries

## ★ Explicit Exception:-

- Explicit exception is nothing but throwing a user created exception (i.e.custom exception).
- To throw an exception explicitly we use a throw keyword.

### throw:-

- throw is keyword which is used to throw an exception explicitly.
- It is use to throw only “Throwable” types of object.
- Once an exception object is thrown the JVM checks whether the method has an exception handiling mechanism if not, exception will be handle by “Default Exception Handler”.
- Syntax:-

```
throw    new    exception    class_name(description of an exception);
```

keywords

- E.g.

```
throw    new    ArithmeticException(“cannot divide by number by zero”);
```

**E.g.1**

```
class ExplicitExceptionExample1
{
	public static void main(String[] args)
	{
		System.out.println("main starts");
		throw new ArithmeticException("Logical Mistake");
		System.out.println("main ends");
	}
}
```

**Output:-**

```
ExplicitExceptionExample1.java:7: error: unreachable statement
		System.out.println("main ends");
		^
```

- We are throwing an exception explicitly compiler is aware about it.
- So, the statements after an exception it will create “Unreachable Statements”.
- So, therefore we need to handle the exception either using try-catch or throws.

**E.g.2**

```
class ExplicitExceptionExample2
{
	public static void main(String[] args)
	{
		System.out.println("main starts");
		try
		{
			throw new ArithmeticException("Logical Mistake");
		}
		catch (ArithmeticException e)
		{
			System.out.println(e.getMessage());
		}
		System.out.println("main ends");
	}
}
```

**Output:-**

```
main starts
Logical Mistake
main ends
```

**E.g.3**

```
class ExplicitExceptionExample3{
	public static void main(String[] args) {
		throw new Exception("Exception created explicitly");                →CompileTimeError
		throw new ClassCastException("Cannot be nested");                   →CompileTimeError
		throw new NullPointerException("Object not created");               →CompileTimeError
		throw new Throwable("root class exception");                        →CompileTimeError
		throw new InterruptedException("thread get interrupted");   →CompileTimeError
		throw new NoSuchFieldException("file not found");           →CompileTimeError
	}
}
```

**Note:-**

- throw keyword throw only Throwable type of object it means that other than exception hierarchy, if we try to throw an object of any other class we will get a CTE i.e. incompatible type.
- To overcome these problem we need to extends that class with exception hierarchy to create our own "Custom exception".

## ★ Custom Exception:-

- To create customize message (description) we use custom exception.
- We can create user defined exception by extending that class with any of their class in exception hierarchy.
- We can create user defined exception as checked or unchecked exception.
- But it is always recommended to create user define exception as "**Unchecked Exception**" by extending with "**Runtime Exception**" or its **child classes**.
- **E.g.1**

```
import java.util.Scanner;
class AgeNotEligible extends RuntimeException
{
    AgeNotEligible(String desc)
    {
            super(desc);
    }
}
class CustomException1
{
    public static void main(String[] args)
    {
            Scanner sc = new Scanner(System.in);
            System.out.println("Enter an age : ");
            int age = sc.nextInt();
            if(age<18)
            {
                    try
                    {
                            throw new AgeNotEligible("Age is less than 18");
                    }
                    catch(AgeNotEligible ane)
                    {
                            System.out.println(ane.getMessage());
                    }
            }
            else
            {
                    System.out.println("Eligible for driving");
            }
    }
}
```

**E.g.2**

```
import java.util.Scanner;
class UserArithmeticException extends RuntimeException {
	UserArithmeticException(String desc){
		super(desc);
	}
}
class CustomException {
	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Number 1 : ");
		int num1 = sc.nextInt();
		System.out.println("Number 2 : ");
		int num2 = sc.nextInt();
		System.out.println("Enter operator : ");
		char ch = sc.next().charAt(0);
		if(ch!='/'){
			System.out.println("Other than divide operation will be performed.");
		}
		else{
			if(num2==0){
				throw new UserArithmeticException("number is zero cannot be divided");
			}
			else {
				System.out.println(num1/num2);
			}
		}
	}
}
```

**E.g.3**

```
class UserNullPointerException extends RuntimeException {
	UserNullPointerException(String desc){
		super(desc);
	}
}
class Student{
	String name;
}
class CustomException3{
	public static void main(String[] args) {
		Student obj = null;
		if(obj!=null){
			System.out.println(obj.name);
		}
		else {
			try{
				throw new UserNullPointerException("Cannot invoke name because obj is
null.");
			}
			catch(UserNullPointerException e){
				System.out.println(e.getMessage());
			}
		}
	}
}
```

**E.g.4**

```
class UserClassCastException extends RuntimeException{
	public UserClassCastException(String desc){
		super(desc);
	}
}
class CustomException4{
	public static void main(String[] args) {
		Object obj1 = new String("Hello");
		StringBuffer obj2 = castObject(obj1);
		System.out.println(obj2);
	}
	public static StringBuffer castObject(Object obj){
		StringBuffer sb = null;
		if(!(obj instanceof StringBuffer)){
			throw new UserClassCastException("Obj cannot be casted to StringBuffer");
		}
		else {
			sb = (StringBuffer)obj;
		}
		return sb;
	}
}
```

★ **throws:-**

- throws is a keyword use at end of method declaration to indicate that exception of given time may be thrown by the method.
- The main purpose of throws keyword is to deligate the responsibility of handling the exception through the calling method.
- In case of Unchecked Exception it is not require to use “throws” keyword.
- We can use throws keyword only for Throwable type of object otherwise we will get a compile time error i.e. “Incompatible type”.

**E.g.1**

```
class ThrowsExample {
	public static void main(String[] args) {
		System.out.println("main");
		try{
			m1(10,0);
		}
		catch(ArithmeticException ae){
			System.out.println("ArithmeticException handled");
		}
	}
	public static void m1(int num1,int num2) throws ArithmeticException {
		System.out.println("m1()");
		div(num1,num2);
	}
	public static void div(int num1,int num2) throws ArithmeticException {
		System.out.println("div");
		System.out.println(num1/num2);
	}
}
```

**E.g.2 – First way**

```
class ThrowsExample2{
	public static void main(String[] args) throws InterruptedException{
		String name = "Hello Java";
		for(int i=0;i<name.length();i++){
			System.out.println(name.charAt(i));
				Thread.sleep(1000);
		}
	}
}
```

**E.g.2 – Second way**

```
class ThrowsExample2{
	public static void main(String[] args){
		String name = "Hello Java";
		for(int i=0;i<name.length();i++){
			System.out.println(name.charAt(i));
			try{
				Thread.sleep(1000);
			}
			catch(InterruptedException ie){
				System.out.println("InterruptedException handled");
			}
		}
	}
}
```

**E.g.3**

```
import java.io.*;
class ThrowsExample3{
	public static void main(String[] args) throws Exception{
		FileWriter file = new FileWriter("abc.txt");
	}
}
```

**Note:-**

With the help of throws keyword we can handle multiple exception separated by comma(,).

**E.g.4**

```
class ThrowsExample4{
	public static void main(String[] args) {
		m1();
	}
	public static void m1() throws ArithmeticException,IndexOutOfBoundsException{
		System.out.println("m1()");
	}
}
```

**Note:-**

It is never recommended to handle exception using **throws** keyword as it just transfer the responsibility to the calling method and at the end it will be an abnormal termination.

## ★ Difference between throw and throws

| throw | throws |
|---|---|
| i. throw keyword is use to throw exception explicitly. | i. throws keyword is used to declare an exception. |
| ii. throw is followed by an instance. | ii. throws is followed by a class. |
| iii. throw is used within the method. | iii. throws is used with the method signature. |
| iv. throw cannot throw multiple exception. | iv. throws can declare multiple exception. |

## ★ Difference between final, finally, finalize

| final | finally | finalize |
|---|---|---|
| i. final is a keyword and non access modifier. | i. finally is a block use in exception handling. | i. finalize is a method present in Object class. |
| ii. It is use to restrict the inheritance of a class, overriding a method and re-initializing of a variable. | ii. It is used in exception to execute some crucial statement. | ii. It is used to perform cleanup activity before an object is being destroyed by GC (Garbage Collector). |
| iii. It is only applicable for variable, class and method. | iii. It is associated with try and catch block. | iii. It is a method applied to object. |
| | iv. After try and catch block, finally block get executed. | iv. finalize() method gets executed before an object gets destroyed. |

# ★ Arrays :-

## ★ Array:-

- Array is an index collection of fixed number of homogeneous data elements.
- Arrays are fixed in length(size).
- Arrays can store only homogeneous data element (same type of data).
- Arrays uses indexing to store the element.
- Arrays of object in java (Non-primitive data type).

**Array Declaration:-**

```
class Demo
{
    public static void main(String[] args)
    {
            int[] arr;
            int arr[];
            int [           ]arr;
            int[] a[][];
            byte[] a,b,c;
            short[][] a,b;
            char[] a[],b;
            boolean[] a[],b,c[];
            long  a[],b,c;
            float[] a[],[]b;
            double[][] a,b[],c[][];
    }
}
```

arr → 1D

arr → 1D

arr → 1D

a → 3D

a→ 1D, b→1D,c→1D

a → 2D, b→ 2D

a → 2D, b→1D

a → 2D, b→1D, c→ 2D

a → 1D, b → variable, c→ variable

X Compile Time Error

a→2D, b→3D,c→ 4D

**Note:-**

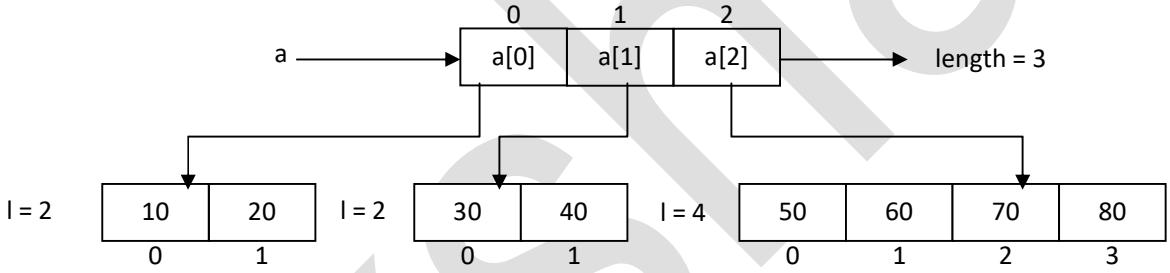
- We can store a huge number of data using single variable so that readability of code is improved.
- It's not at all possible to increase or decrease the size of an array once it is declared.
- Hence, we should use this concept only if we know the size of an array in advance.
- If we are declaring a multidimensional array at same time we are declaring multiple arrays in same line after the first variable we should use identifier and then an Array operator ( **[ ]** ) if required.
  ( **[ ]** → array operator is also a separator )

  E.g.
  ```
  int [] a [], []b;   //Wrong
  int [] a, b[];      //Correct
  ```
- Arrays in java are object so create them using new keyword.
- Syntax:-
  datatype[] var_name = new datatype [size];
  E.g:-
  int[] a = new int [10];

- If we try to declare an array without specifying size we get CompileTimeError i.e."Array Dimension Missing".
- We can declare a multidimensional array without specifying the next dimension size, but we have to specify it before using.
  E.g.
  int [][] arr = new int[10][];
  int [][] arr = new int[10][10];
- We can create an array of size "0" (zero), but we cannot store any element in it.
  E.g.
  int [] a = new int [0];
  Example of array size **zero** is **(String[] args).**
- We cannot create an array of negative length if we try, we get "**RuntimeException**" i.e. "**NegativeArraySizeException**".
- We can create an array of maximum capacity (length) i.e. 2147483647;
- **There are two possibilities→**
  1. If VM (Virtual Machine) don't have enough memory for creating an object it will get RuntimeError i.e. OutOfMemoryError.
  2. If our VM have enough memory object will be created.
- We can create an Array upto "255 dimensions".

## ★ Jagged Array:-

- An array of array varying length or size is known as the jagged array.
- E.g.

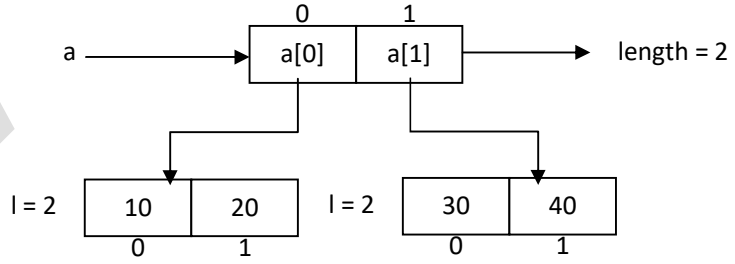



- Declaration :-

```
int [][] a = new int [3][];
a[0] = new int[2];
a[1] = new int[2];
a[2] = new int[4];
```

## ★ Matrix Array:-

- An array of array with same size is known as matrix array.
- In matrix array the number of rows and columns are same.
- E.g.

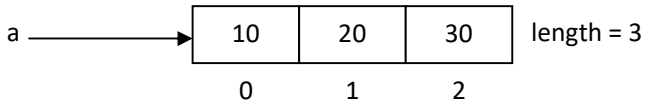



Declaration :-

```
int [][] a = new int [2][2];
```

## Array creation, declaration and initialization:-

**E.g. 1**




First way :-

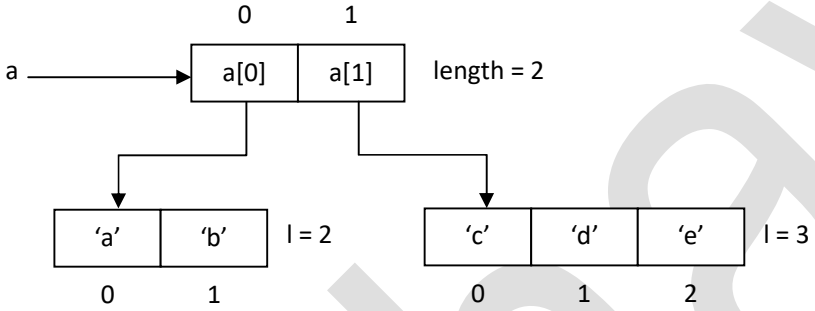
```
int[] a = new int[3];      //declaration
a[0] = 10;         //initialization
a[1] = 20;
a[2] = 30;
```

Second way :-

```
int [] a = {10,20,30};
```

---

**E.g. 2**




First way :-

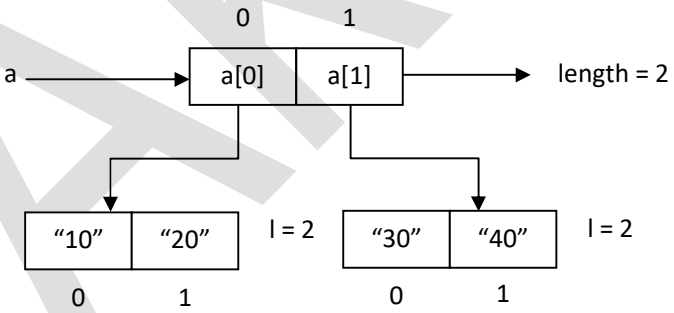
```
char[] a = new int[2][];
a[0] = new int[2];
a[1] = new int[3];
a[0][0] = 'a';     //initialization
a[0][1] = 'b';
a[1][0] = 'c';
a[1][1] = 'd';
a[1][2] = 'e';
```

Second way :-

```
char [][] a = { {'a','b'} , {'c','d','e'} };
```

---

**E.g. 3**



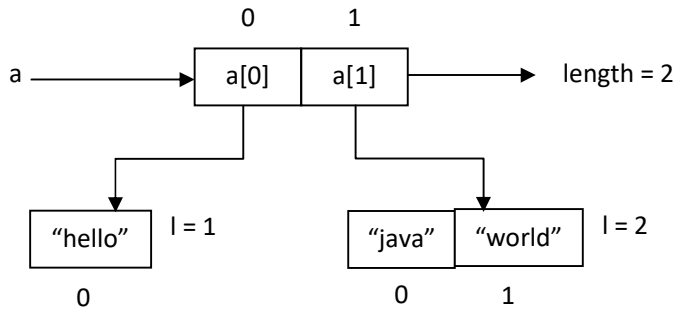

First way :-

```
String[][] a = new String[2][2];
a[0][0] = "10";
a[0][1] = "20";
a[1][0] = "30";
a[1][1] = "40";
```

Second way :-

```
String[][] a = { {"10", "20"}, {"30", "40"} };
```

**E.g. 4**




First way :-

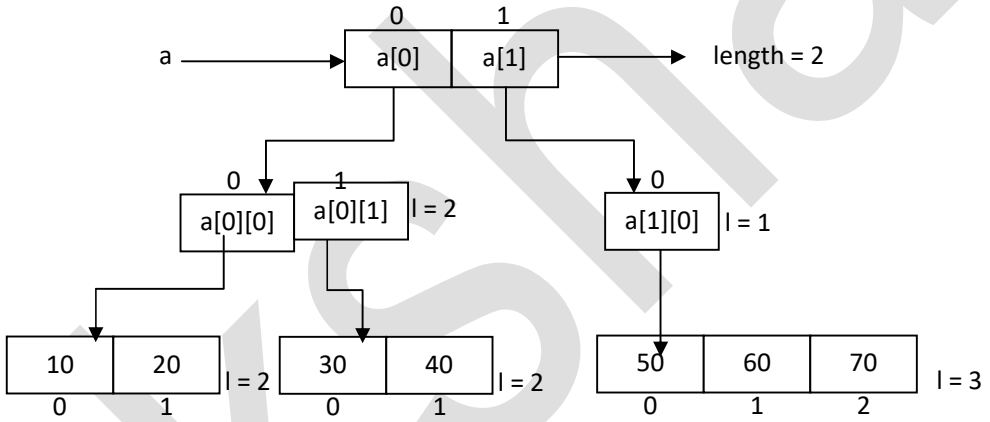
```
String[][] a = new String[2][];
a[0] = new String[1];
a[1] = new String[2];
a[0][0] = "hello";
a[1][0] = "java";
a[1][1] = "world";
```

Second way :-

```
String[][] a = { {"hello"}, {"java", "world"} };
```

---

**E.g. 5**




First way:-

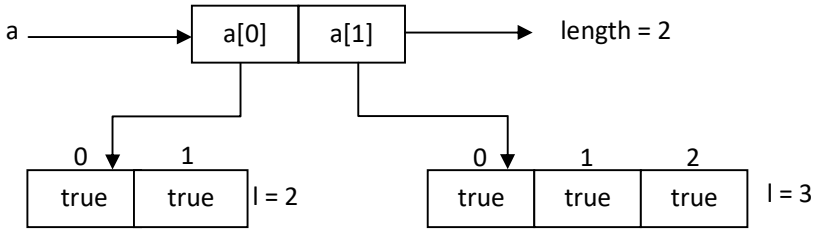
```
int[][][] a = new int[2][][];
int a[0] = new int[2][2];
inta[1] = new int[1][3];
a[0][0][0] = 10;
a[0][0][1] = 20;
a[0][1][0] = 30;
a[0][1][1] = 40;
a[1][0][0] = 50;
a[1][0][1] = 60;
a[1][0][2] = 70;
```

Second way:-

```
int[][][] a = { { {10,20},{30,40} }, { {50,60,70} } };
```

---

**E.g. 6**




First way:-

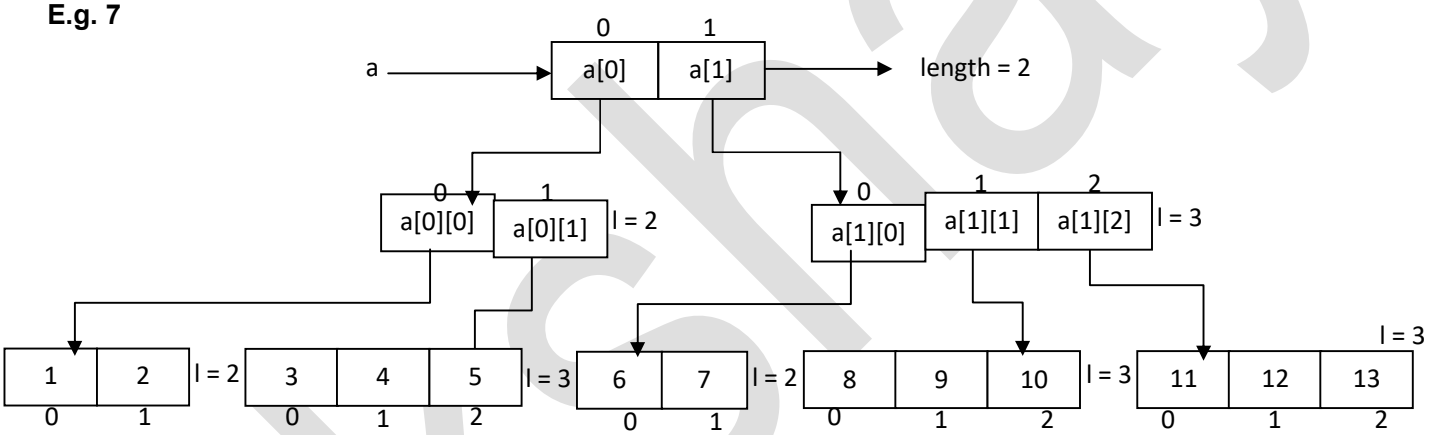
```
boolean[][] a = new boolean[2][];
a[0] = new boolean[2];
a[1] = new boolean[3];
a[0][0] = true;
a[0][1] = true;
a[1][0] = true;
a[1][1] = true;
a[1][2] = true;
```

Second way :-

```
boolean[][] a = { {true,true} , {true,true,true} };
```

---

**E.g. 7**



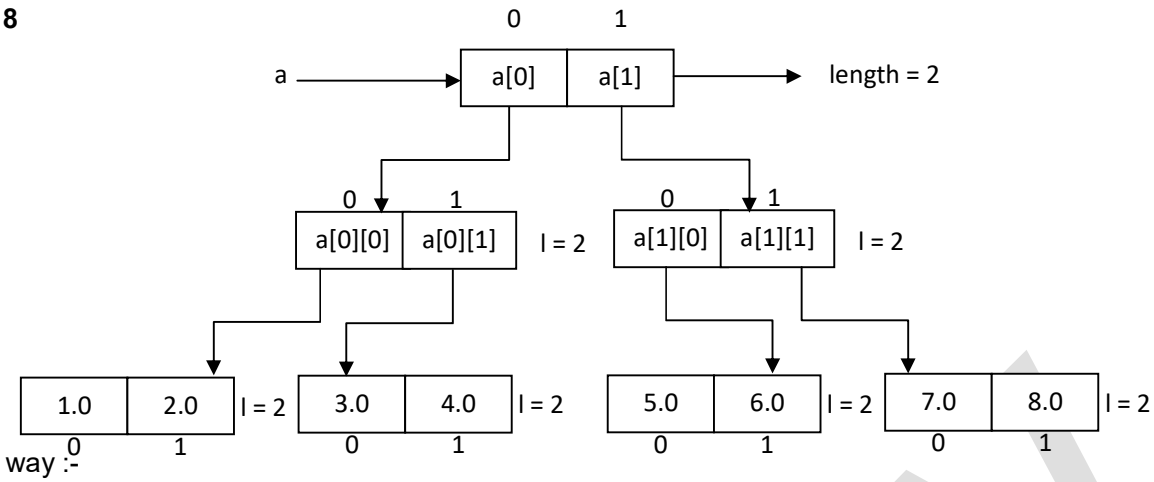

First way:-

```
byte[][][] a = new byte[2][][];
a[0] = new byte[2][];
a[0][0] = new byte[2];
a[0][1] = new byte[3];
a[1] = new byte[3][];
a[1][0] = new byte[2];
a[1][1] = new byte[3];
a[1][2] = new byte[3];
a[0][0][0] = 1;
a[0][0][1] = 2;
a[0][1][0] = 3;
a[0][1][1] = 4;
a[0][1][2] = 5;
a[1][0][0] = 6;
a[1][0][1] = 7;
a[1][1][0] = 8;
a[1][1][1] = 9;
a[1][1][2] = 10;
a[1][2][0] = 11;
a[1][2][1] = 12;
a[1][2][2] = 13;
```

Second way :-

```
byte[][][] a = { { {1,2},{3,4,5} }, { {6,7} , {8,9,10} , {11,12,13} } };
```

**E.g. 8**




First way :-

```
float[][][] a new float[2][2][2];
a[0][0][0] = 1.0f;
a[0][0][1] = 2.0f;
a[0][1][0] = 3.0f;
a[0][1][1] = 4.0f;
a[1][0][0] = 5.0f;
a[1][0][1] = 6.0f;
a[1][1][0] = 7.0f;
a[1][1][1] = 8.0f;
```

Second way :-

```
float[][][] a = { { {1.0f,2.0f} } , { {3.0f,4.0f} } , { {5.0f,6.0f} } , { {7.0f,8.0f} } };
```

★ **length variable:-**

- length is an built in variable which returns the length of an array.
- length is an non static variable present in every array.
- E.g.

```
class length
{
    public static void main(String[] args)
    {
            int [] arr = {1,2,3,4,5,6,7,8,9,10};
            System.out.println("Array : ");
            for (int i : arr)
            {
                    System.out.print(i+" ");
            }
            System.out.println();
            System.out.println("Length of array : "+arr.length);
    }
}
```

**Output:-**

```
Array :
1 2 3 4 5 6 7 8 9 10
Length of array : 10
```

**E.g. 1**

```
class Demo1{
	public static void main(String[] args) {
		int[] arr = {10,20,30,40,50,60,70,80,90,100};
		for(int i=0;i<arr.length;i++){
			System.out.print(arr[i]+" ");
		}
	}
}
```

**Output:-**

10 20 30 40 50 60 70 80 90 100

**E.g. 2**

**//finding the length of an array without using length variable**

```
class Demo2{
	public static void main(String[] args) {
		int[] arr = {10,20,30,40,50,60,70,80,90,100};
		int len = 0;
		for(int i : arr){
			len++;
		}
		System.out.println("Length : "+len);
	}
}
```

**Output:-**

Length : 10

**E.g. 3**

**//another example for finding the length of an array**

```
class Demo3{
	public static void main(String[] args) {
		int[] arr = {1,2,3,4,5,6,7,8,9,10};
		int leng = 0 ;
		for(int i=0; ; i++) {
			try{
				System.out.print(arr[i]+" ");
				leng++;
			}
			catch(ArrayIndexOutOfBoundsException e){
				break;
			}
		}
		System.out.println();
		System.out.println("Length of array : "+leng);
	}
}
```

**Output:-**

1 2 3 4 5 6 7 8 9 10
Length of array : 10

- Arrays are non primitive and for creating object in java we need a class because java is class based programming language.
- To fetch the class name of an array we have a built in method i.e. getClass()
- E.g.

```
class Demo4
{
  public static void main(String[] args) {
            byte[] a = new byte[1];
            short[] b = new short[1];
            int[] c = new int[1];
            long[] d = new long[1];
            float[] e = new float[1];
            double[] f = new double[1];
            char[] g = new char[1];
            boolean[] h = new boolean[1];
            String[] i = new String[1];
            Student[] j = new Student[1];
            System.out.println(a.getClass());
            System.out.println(b.getClass());
            System.out.println(c.getClass());
            System.out.println(d.getClass());
            System.out.println(e.getClass());
            System.out.println(f.getClass());
            System.out.println(g.getClass());
            System.out.println(h.getClass());
            System.out.println(i.getClass());
            System.out.println(j.getClass());
  }
}
class Student
{

}
```

**Output:-**

```
class [B
class [S
class [I
class [J
class [F
class [D
class [C
class [Z
class [Ljava.lang.String;
class [LStudent;
```

★ **Printing of one dimensional array :-**

```
int[] a = {1,2,3,4,5};
```

**E.g. 1 Using for loop**

```
class Demo5
{
  public static void main(String[] args)
{
        int[] a = {1,2,3,4,5};
        for(int i=0;i<a.length;i++)
        {
                System.out.print(a[i]+" ");
        }
  }
}
```

**Output:-**

1 2 3 4 5

**E.g. 2 Using while loop**

```
class Demo5
{
  public static void main(String[] args)
  {
        int[] a = {1,2,3,4,5};
        int i = 0 ;
        while(i<a.length)
        {
                System.out.print(a[i]+" ");
                i++;
        }
  }
}
```

**Output:-**

1 2 3 4 5

**E.g. 3 Using do while loop**

```
class Demo5
{
  public static void main(String[] args)
  {
        int[] a = {1,2,3,4,5};
        int i=0;
        do
        {
                System.out.print(a[i]+" ");
                i++;
        }while(i<a.length);
  }
}
```

**Output:-**

1 2 3 4 5

**E.g. 4 Using for each loop**

```
class Demo5
{
  public static void main(String[] args)
  {
        int[] a = {1,2,3,4,5};
        for (int i : a)
        {
              System.out.print(i+" ");
        }
  }
}
```

**Output:-**

1 2 3 4 5

**E.g. 5 Using toString()**

```
import java.util.Arrays;
class Demo5
{
  public static void main(String[] args)
  {
        int[] a = {1,2,3,4,5};
        System.out.println(Arrays.toString(a));
  }
}
```

**Output:-**

[1, 2, 3, 4, 5]

★ **toString():-**

- Arrays.toString() returns a String representation of the elements specified in that array.
- A string which consists of list of elements enclosed in [ ] (square brackets) and separated using a comma ( , ).
- It's a static method defined in java.util.Arrays package so before using it we need to import this class.

**Note:-**

Arrays.toString() method does not belongs to Object class.

**E.g.**

```
import java.util.Arrays;
class Demo5
{
    public static void main(String[] args)
    {
          char[] a = {'a','e','i','o','u'};
          System.out.println(Arrays.toString(a));
    }
}
```

**Output:-**

[a, e, i, o, u]

## ★ for each loop:-

- The for each loop in java was introduced in JDK 1.5.
- The internal implementation of for each loop is listIterator which makes the for each loop faster among all.
- It provides us an alternative approach for traversing an arrays or collection.
- This loop is specially designed to work on data structures like arrays and collection.
- The main advantage of using for each is that it eliminates the problem of ArrayIndexOutOfBounds Exception and makes code more readable.
- It is known as for each because it traverse elements one by one.
- E.g.

```
import java.util.*;
class Demo6
{
    public static void main(String[] args)
    {
            String[] str = {"ramesh","suresh","ganesh"};
            for(String name : str)
            {
                    System.out.print(name+"  ");
            }
            System.out.println();
            ArrayList<Integer>list = new ArrayList<>();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            list.add(50);
            for(Integer i : list)
            {
                    System.out.print(i+"  ");
            }
    }
}
```

**Output:-**

```
ramesh  suresh  ganesh
10  20  30  40  50
```

**Syntax:-**

```
for(datatype   var_name   :   array/collection var_name)
{
    //implementation
}
```

**Note:-**

- The drawback of for each loop is that it cannot traverse the elements in reverse order.
- It can be only used in Arrays and Collections.
- We cannot start a for each loop any where in the middle of data structure.
- Also we cannot skip an iteration as it does not conatins any index.

## Printing of multidimensional array:-

```
int[][] a = {{10,20,30,40},{50,60,70,80,90,100}};
```

**E.g.1 Using for loop**

```
class Demo7
{
  public static void main(String[] args)
  {
        int[][] a = {{10,20,30,40},{50,60,70,80,90,100}};
        for(int i=0;i<a.length;i++)
        {
              for(int j=0;j<a[i].length;j++)
              {
                    System.out.print(a[i][j]+"  ");
              }
              System.out.println();
        }
  }
}
```

**Output:-**

```
10  20  30  40
50  60  70  80  90  100
```

**E.g.2 Using while loop**

```
class Demo7
{
  public static void main(String[] args)
  {
        int[][] a = {{10,20,30,40},{50,60,70,80,90,100}};
        int i=0;
        while(i<a.length)
        {
              int j=0;
              while(j<a[i].length)
              {
                    System.out.print(a[i][j]+"  ");
                    j++;
              }
              System.out.println();
              i++;
        }
  }
}
```

**Output:-**

```
10  20  30  40
50  60  70  80  90  100
```

**E.g. 3 Using do while loop**

```
class Demo7{
  public static void main(String[] args) {
        int[][] a = {{10,20,30,40},{50,60,70,80,90,100}};
        int i=0;
        do{
              int j=0;
              do {
                    System.out.print(a[i][j]+"  ");
                    j++;
              }while(j<a[i].length);
              i++;
              System.out.println();
        }while(i<a.length);
  }
}
```

**Output:-**

```
10  20  30  40
50  60  70  80  90  100
```

**E.g. 4 Using for each loop**

```
class Demo8{
  public static void main(String[] args) {
        int[][] a = {{10,20,30,40},{50,60,70,80,90,100}};
        for(int[] i : a){
              for(int j : i){
                    System.out.print(j+" ");
              }
              System.out.println();
        }

  }
}
```

**Output:-**

```
10  20  30  40
50  60  70  80  90  100
```

**E.g. 5 Using deepToString()**

```
import java.util.*;
class Demo8 {
  public static void main(String[] args) {
        int[][] a = {{10,20,30,40},{50,60,70,80,90,100}};
        System.out.println(Arrays.deepToString(a));

  }
}
```

**Output:-**

```
[[10, 20, 30, 40], [50, 60, 70, 80, 90, 100]]
```

## ★ deepToString():-

- Arrays.deepToString() returns String representation of a multidimensional array.
- The String representation consists of list of arrays enclosed within square bracket ( [ ] ) separated using a comma ( , ).
- E.g.

```
import java.util.Arrays;
class Demo9
{
    public static void main(String[] args)
    {
            String[][] datalist = {{"1","ramesh"},{"2","suresh"},{"3","ganesh"}};
            System.out.println(Arrays.deepToString(datalist));
            for(String[] i : datalist)
            {
                    System.out.print(Arrays.toString(i));
            }
    }
}
```

**Output:-**

```
[[1, ramesh], [2, suresh], [3, ganesh]]
[1, ramesh][2, suresh][3, ganesh]
```

## ★ ArrayIndexOutOfBoundsException:-

- AIOOBE is a common issue in java that occurs when we try to access an element which is located at an invalid index (index which does not exists).
- E.g.

```
import java.util.Arrays;
class Demo9
{
    public static void main(String[] args)
    {
            int[] a = {1,2,3,4,5};
            for(int i=0;i<10;i++)
            {
                    System.out.println(a[i]);
            }
    }
}
```

**Output:-**

```
1
2
3
4
5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
      at Demo9.main(Demo9.java:9)
```

## ★ Anonymous Array:-

- An array in java without any name is called as anonymous array.
- Its an array which is created just for an instant use.
- We use anonymous array most oftenly when we need to pass it to a method as an argument.
- When the use of this array is completed it is destroyed by the garbage collector.
- We can also store an anonymous array inside a variable.
- **E.g. 1**

```
class AnonymousArray{
    public static void main(String[] args) {
            checkEvenOdd(new int[]{1,2,3,4,5});
    }
    public static void checkEvenOdd(int[] arr){
            for(int i : arr){
                    if(i%2==0)
                            System.out.println(i+" even");
                    else
                            System.out.println(i+" odd");
            }
    }
}
```

**Output:-**

```
1 odd
2 even
3 odd
4 even
5 odd
```

**E.g. 2**

```
class AnonymousArray{
    public static void main(String[] args) {
            int[] arr = new int[]{1,2,3,4,5};
            checkEvenOdd(arr);
    }
    public static void checkEvenOdd(int[] arr){
            for(int i : arr){
                    if(i%2==0)
                            System.out.println(i+" even");
                    else
                            System.out.println(i+" odd");
            }
    }
}
```

**Output:-**

```
1 odd
2 even
3 odd
4 even
5 odd
```

# Assignment

1. **Find even odd number (elements) from an array.**

```
//Find even and odd element from an array
class Question1
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,6,7,8,9};
		for (int i : arr)
		{
			if(i%2==0)
				System.out.println(i+" even  ");
			else
				System.out.println(i+" odd  ");
		}
	}
}
```

**Output:-**

```
1 odd
2 even
3 odd
4 even
5 odd
6 even
7 odd
8 even
9 odd
```

2. **Create user defined one dimensional array and store elements.**

```
//Create a user defined one dimensional array
import java.util.*;
class Question2
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the size of an array : ");
		int size = sc.nextInt();
		int[] arr = new int[size];
		for (int i=0;i<size;i++)
		{
			System.out.print("Enter the "+(i+1)+" element of array : ");
			arr[i] = sc.nextInt();
		}
		System.out.println("Array elements");
		for (int i : arr)
		{
			System.out.println(i);
		}
	}
}
```

**3. Create user defined two dimensional array.**

```
//create user defined two dimensional array
import java.util.*;
class Question3
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("First dimension of array : ");
		int size1 = sc.nextInt();
		int[][]arr = new int[size1][];

		for (int i=0;i<size1;i++)
		{
			System.out.print("Enter second dimension size for "+(i+1)+" arrray : ");
			int size2 = sc.nextInt();
			arr[i] = new int[size2];
		}

		for (int[] i : arr )
		{
			for(int j : i )
			{
				System.out.print(j+" ");
			}
		}

		System.out.println();

		for(int i=0;i<arr.length;i++)
		{
			for(int j=0;j<arr[i].length;j++)
			{
				System.out.print("Enter the "+(j+1)+" element of "+(i+1)+" array : ");
				int ele = sc.nextInt();
				arr[i][j] = ele;
			}
		}

		for (int[] i : arr )
		{
			for(int j : i )
			{
				System.out.print(j+" ");
			}
		}
	}
}
```

**4. Sum of elements of an array.**

```
//sum of elements of an array
import java.util.*;
class Question4{
	public static void main(String[] args) {
		int[] arr = {1,2,3,4,5,6,7,8,9,10};
		int sumOfArr=0;
		for (int i : arr) {
			sumOfArr+=i;
		}
		System.out.println("Sum of array elements : "+sumOfArr);
	}
}
```

**Output:-**

Sum of array elements : 55

**5. Product of elements of an array.**

```
//product of an array
import java.util.*;
class Question5{
	public static void main(String[] args) {
		int[] arr = {1,2,3,4,5};
		int productOfArray=1;
		for (int i : arr) {
			productOfArray*=i;
		}
		System.out.println("Product of array elements : "+productOfArray);
	}
}
```

**Output:-**

Product of array : 120

**6. Find average of an array.**

```
//find average of an array
import java.util.*;
class Question6{
	public static void main(String[] args) {
		int[] arr = {1,2,3,4,5,6,7,8,9,10};
		int sumOfArr=0;
		for (int i : arr) {
			sumOfArr+=i;
		}
		double average = sumOfArr/arr.length;
		System.out.println("Average of array is : "+average);
	}
}
```

**Output:-**

Average of array is : 5.0

**7. Find prime elements from an array.**

```
//find prime elements from an array
import java.util.*;
class Question7{
	public static void main(String[] args) {
		int[] arr = {1,2,3,4,5,6,7,8,9};

		for (int i : arr) {
			if(isPrimeNumber(i))
				System.out.print(i+ " ");
		}
	}
	public static boolean isPrimeNumber(int num){
		if(num==1)
			return false;
		for(int i=2;i<num;i++)
			if(num%i==0)
				return false;
		return true;

	}
}
```

**Output:-**

```
2 3 5 7
```

**8. Find palindrome name from an array.**

```
//find palindrome names from an array
import java.util.*;
class Question8{
	public static void main(String[] args) {
		String[] arr = {"akshay","nitin","rutvik","laal"};

		for (String name : arr) {
			isPalindromeName(name);
		}
	}
	public static void isPalindromeName(String name){
		String rev ="";
		int i=0;
		while(name.length()>i){
			rev=name.charAt(i)+rev;
			i++;
		}
		System.out.println(rev.equals(name)?name+" is a palindrome name":name+" is not a
palindrome name");
	}
}
```

**Output:-**

```
akshay is not a palindrome name
nitin is a palindrome name
rutvik is not a palindrome name
laal is a palindrome name
```

**9. Sum of even and odd numbers from an array.**

```
//sum of even and odd numbers from an array
class Question9
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,6,7,8,9};
		int sumOfEven = sumOfEven(arr);
		int sumOfOdd = sumOfOdd(arr);
		for (int i : arr)
		{
			System.out.print(i+"  ");
		}
		System.out.println();
		System.out.println("Sum of even number from array : "+sumOfEven);
		System.out.println("Sum of odd number from array : "+sumOfOdd);
	}
	public static int sumOfEven(int[] num)
	{
		int sumOfEven=0;
		for (int i : num)
		{
			if(i%2==0)
				sumOfEven+=i;
		}
		return sumOfEven;
	}
	public static int sumOfOdd(int[] num)
	{
		int sumOfOdd=0;
		for (int i : num)
		{
			if(i%2!=0)
				sumOfOdd+=i;
		}
		return sumOfOdd;
	}
}
```

**Output:-**

```
1  2  3  4  5  6  7  8  9
Sum of even number from array : 20
Sum of odd number from array : 25
```

**10. Merge one dimensional array.**

```
//merge one dimensional array
class Question10
{
	public static void main(String[] args)
	{
		int[] arr1 = {1,2,3,4,5};
		int[] arr2 = {6,7,8,9};
		System.out.println("First array : ");
		for (int i : arr1)
		{
			System.out.print(i+" ");
		}
		System.out.println();
		System.out.println("Second array : ");
		for (int i : arr2)
		{
			System.out.print(i+" ");
		}
		System.out.println();
		System.out.println("Third array before storing elements : ");
		int[] mergeArr = new int[arr1.length+arr2.length];
		for (int i : mergeArr)
		{
			System.out.print(i+" ");
		}

		for(int i=0,bi=0;i<mergeArr.length;i++)
		{
			if(i<arr1.length)
				mergeArr[i] = arr1[i];
			else
				mergeArr[i] = arr2[bi++];
		}
		System.out.println();
		System.out.println("Third array after storing elements : ");
		for (int i : mergeArr)
		{
			System.out.print(i+" ");
		}
	}
}
```

**Output:-**

```
First array :
1 2 3 4 5
Second array :
6 7 8 9
Third array before storing elements :
0 0 0 0 0 0 0 0 0
Third array after storing elements :
1 2 3 4 5 6 7 8 9
```

## 11. Merge one dimensional array in zigzag format.

```
//merge one dimensional array in zigzag format
import java.util.*;
class Question11
{
	public static void main(String[] args)
	{
		int[] arr1 = {10,20,30,40,50};
		int[] arr2 = {60,70,80};
		System.out.println("First array : ");
		for (int i : arr1)
		{
			System.out.print(i+" ");
		}
		System.out.println();
		System.out.println("Second array : ");
		for (int i : arr2)
		{
			System.out.print(i+" ");
		}
		int[] arr3 = new int[arr1.length+arr2.length];
		System.out.println();
		System.out.println("Third array before storing elements in zigzag : ");
		for (int i : arr3)
		{
			System.out.print(i+" ");
		}
		System.out.println();
		for(int i=0,cindex=0;i<arr1.length;i++)		//using cindex we store elements
		{
			arr3[cindex++] = arr1[i];		//after every store cindex increaments by one
			if(i<arr2.length)
				arr3[cindex++] = arr2[i];
		}
		System.out.println("Third array after storing elements in zigzag : ");
		for (int i : arr3)
		{
			System.out.print(i+" ");
		}
	}
}
```

**Output:-**

```
First array :
10 20 30 40 50
Second array :
60 70 80
Third array before storing elements in zigzag :
0 0 0 0 0 0 0 0
Third array after storing elements in zigzag :
10 60 20 70 30 80 40 50
```

**12. Convert two dimensional array to one dimensional array.**

```
//convert (merge) two dimensional array to one dimensional array
import java.util.*;
class Question12
{
	public static void main(String[] args)
	{
		int[][] a = {{10,20,30},{40,50,60,70,80,90}};
		System.out.println(Arrays.deepToString(a));
		int len=0;
		for(int[] i : a)
			len+=i.length;
		int[] newArr = new int[len];
		int indx = 0;
		for(int[] i : a)
			for(int ele : i)
				newArr[indx++]=ele;
		System.out.println(Arrays.toString(newArr));
	}
}
```

**Output:-**

```
[[10, 20, 30], [40, 50, 60, 70, 80, 90]]
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

**13. Convert first 10 elements from fibonacchi series to an array.**

```
//convert first 10 numbers from fibonacchi series to an array
import java.util.*;
class Question13
{
	public static void main(String[] args)
	{
		int[] fibonacciNumbers = new int[10];
      fibonacciNumbers[0] = 0;
      fibonacciNumbers[1] = 1;

      for (int i = 2; i < 10; i++) {
         fibonacciNumbers[i] = fibonacciNumbers[i - 1] + fibonacciNumbers[i - 2];
      }

      System.out.println(Arrays.toString(fibonacciNumbers));
	}
}
```

**Output:-**

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

## 14. Find positive and negative numbers and make separate array.

```
//find positive and negative number and make seperate array
class Question14
{
	public static void main(String[] args)
	{
		int[] arr = {-5,-4,-3,-2,-1,0,1,2,3,4,5};
		int len1 = 0 ;
		int len2 = 0 ;
		System.out.println("Array : ");
		for (int i : arr) {
			System.out.print(i+" ");
		}
		System.out.println();
		for (int i : arr)
		{
			if(i>=0)
				len1++;
			else
				len2++;
		}
		System.out.println("Positive number length : "+len1);
		System.out.println("Negative number length : "+len2);

		int[] posArr = new int[len1];
		int[] negArr = new int[len2];
		for(int i=0,pi=0,ni=0;i<arr.length;i++)
		{
			if(arr[i]>=0)
				posArr[pi++]=arr[i];
			else
				negArr[ni++]=arr[i];
		}
		System.out.println("Positive array : ");
		for (int i : posArr)
		{
			System.out.print(i+" ");
		}
		System.out.println();
		System.out.println("Negative array : ");
		for (int i : negArr)
		{
			System.out.print(i+" ");
		}
	}
}
```

**Output:-**

```
Array :
-5 -4 -3 -2 -1 0 1 2 3 4 5
Positive number length : 6
Negative number length : 5
Positive array :
0 1 2 3 4 5
Negative array :
-5 -4 -3 -2 -1
```

**Q. 1 Find frequency of an array**

```
//Find frequency of an array
import java.util.*;
class ArrayFrequency
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,4,3,2,1};
		int len = arr.length;
		boolean[] barr = new boolean[len];	//boolean array used for track of the array
		System.out.println(Arrays.toString(arr));
		System.out.println(Arrays.toString(barr));
		for(int i=0;i<len;i++)
		{
			int iele = arr[i];
			int icnt = 0;
			for(int j=0;j<len;j++)
			{
				int jele = arr[j];
				if(iele==jele && barr[j]==false)
				{
					icnt++;
					barr[j]=true;
				}
			}
			if(icnt!=0)
				System.out.println(iele+" : "+icnt);
		}
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 4, 3, 2, 1]
[false, false, false, false, false, false, false, false, false]
1 : 2
2 : 2
3 : 2
4 : 2
5 : 1
```

**Q.2 Find duplicate elements from an array.**

```
//Find duplicate elements from an array
import java.util.*;
class DuplicateArrayElement
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,4,3,2,1};
		int len = arr.length;
		boolean[] barr = new boolean[len];
		System.out.println(Arrays.toString(arr));
		System.out.println(Arrays.toString(barr));
		for(int i=0;i<len;i++)
		{
			int iele = arr[i];
			int icnt = 0;
			for(int j=0;j<len;j++)
			{
				int jele = arr[j];
				if(iele==jele && barr[j]==false)
				{
					icnt++;
					barr[j]=true;
				}
			}
			if(icnt>1)
				System.out.println(iele);
		}
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 4, 3, 2, 1]
[false, false, false, false, false, false, false, false, false]
1
2
3
4
```

**Q.3 Find unique elements from an array (non repeatable)**

```
//Find unique elements from an array
import java.util.*;
class UniqueArrayElement
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,4,3,2,1};
		int len = arr.length;
		boolean[] barr = new boolean[len];
		System.out.println(Arrays.toString(arr));
		System.out.println(Arrays.toString(barr));
		for(int i=0;i<len;i++)
		{
			int iele = arr[i];
			int icnt = 0;
			for(int j=0;j<len;j++)
			{
				int jele = arr[j];
				if(iele==jele && barr[j]==false)
				{
					icnt++;
					barr[j]=true;
				}
			}
			if(icnt==1)
				System.out.println(iele);
		}
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 4, 3, 2, 1]
[false, false, false, false, false, false, false, false, false]
5
```

**Q. 4 Find distinct elements from an array**

```
//Find distinct elements from an array
import java.util.*;
class DistinctArrayElement
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,4,3,2,1};
		int len = arr.length;
		boolean[] barr = new boolean[len];
		System.out.println(Arrays.toString(arr));
		System.out.println(Arrays.toString(barr));
		for(int i=0;i<len;i++)
		{
			int iele = arr[i];
			int icnt = 0;
			for(int j=0;j<len;j++)
			{
				int jele = arr[j];
				if(iele==jele && barr[j]==false)
				{
					icnt++;
					barr[j]=true;
				}
			}
			if(icnt>0)
				System.out.println(iele);
		}
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 4, 3, 2, 1]
[false, false, false, false, false, false, false, false, false]
1
2
3
4
5
```

**Q.5 Finding smallest element from array**

```
//Find smallest element from an array
import java.util.Arrays;
class SmallestElement
{
	public static void main(String[] args)
	{
		int[] arr = new int[10];
		for(int i=0,indx=0;i<10;i++)
		{
			int num = (int)(Math.random()*1000);
			if(num>100)
				arr[indx++]=num;
			else
				i--;
		}
		System.out.println(Arrays.toString(arr));
		int smallest = Integer.MAX_VALUE;
		for(int ele : arr)
		{
			if(smallest>ele)
				smallest=ele;
		}
		System.out.println(smallest);
	}
}
```

**Q. 6 Finding second smallest number form an array.**

```
//Find second smallest element from an array
import java.util.Arrays;
class SecondSmallestElement
{
	public static void main(String[] args)
	{
		int[] arr = new int[10];
		for(int i=0,indx=0;i<10;i++)
		{
			int num = (int)(Math.random()*1000);
			if(num>100)
				arr[indx++]=num;
			else
				i--;
		}
		System.out.println(Arrays.toString(arr));
		int smallest1 = Integer.MAX_VALUE;
		int smallest2 = Integer.MAX_VALUE;
		for(int ele : arr)
		{
			if(smallest1>ele)
			{
				smallest2 = smallest1;
				smallest1 = ele;
			}
		}
		System.out.println(smallest2);
	}
}
```

**Q. 7 Finding largest number from an array**

```
//Find largest element from an array
import java.util.Arrays;
class LargestElement
{
	public static void main(String[] args)
	{
		int[] arr = new int[10];
		for(int i=0,indx=0;i<10;i++)
		{
			int num = (int)(Math.random()*1000);
			if(num>100)
				arr[indx++]=num;
			else
				i--;
		}
		System.out.println(Arrays.toString(arr));
		int largest = Integer.MIN_VALUE;
		for(int ele : arr)
		{
			if(largest<ele)
				largest=ele;
		}
		System.out.println(largest);
	}
}
```

**Q. 8 Finding second largest number from an array**

```
//Find second largest element from an array
import java.util.Arrays;
class SecondLargestElement
{
	public static void main(String[] args)
	{
		int[] arr = new int[10];
		for(int i=0,indx=0;i<10;i++)
		{
			int num = (int)(Math.random()*1000);
			if(num>100)
				arr[indx++]=num;
			else
				i--;
		}
		System.out.println(Arrays.toString(arr));
		int largest1 = Integer.MIN_VALUE;
		int largest2 = Integer.MIN_VALUE;
		for(int ele : arr)
		{
			if(largest1<ele)
			{
				largest2 = largest1;
				largest1 = ele;
			}
		}
		System.out.println(largest2);
	}
}
```

# ★ Command Line Argument:-

- Command line argument in java are parameters pass to the program from the console (Command Line Interface).
- Basically this arguments are passed at the execution time.
- The parameters which are passed through the command line is received by main method's String[] in a form of elements.
- The data is stored in a form of a String if you need to use it in other forms you can use Wrapper class parse() method.
- If we are not passing any arguments from the command line, the length of args[] is zero.

**How to pass arguments from command line?**

Step-1:-
Compilation command:-
javac FileName.java
Step-2:-
Execution command:-
Java ClassName ele1 ele2 ele3 ….

E.g.

```
import java.util.*;
class CommandLineDemo
{
  public static void main(String[] args)
  {
        System.out.println(Arrays.toString(args));
        int[] num = new int[5];
        for(int i=0;i<num.length;i++)
        {
              num[i] = Integer.parseInt(args[i]);
        }
        System.out.println(Arrays.toString(num));
  }
}
```

**Output:-**

```
javac CommandLineDemo.java
java CommandLineDemo 10 20 30 40 50
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
```

1. **WAJP to find Armstrong elements from an array. Array must be initialized by using command line.**

```
import java.util.*;
class ArmstrongNumber
{
	public static void main(String[] args)
	{
		System.out.println(Arrays.toString(args));	//displaying the elements of (String[] args)
								// taken from cmd line
		int[] numbers = new int[args.length];		//creating array with the size of (String[] args)
		for(int i=0;i<args.length;i++)
		{
			numbers[i] = Integer.parseInt(args[i]);		//storing array elements in the int array
		}							//by coverting it into integer using wrapper class
		System.out.println(Arrays.toString(numbers));
		for(int ele : numbers)
			isArmstrong(ele);		//passing array elements to the method - isArmstrong()
	}
	public static void isArmstrong(int num)
	{
		int pow = (num+"").length();	//converting a number to string to getting length
		int sum = 0;
		for(int i=num;i!=0;i/=10)
		{
			sum+=Math.pow((i%10),pow);
		}
		if(num==sum)
			System.out.println(num);
	}
}
```

java ArmstrongNumber 153 123 1634 789 //command line argument numbers passed at execution time

**Output:-**

```
 [153, 123, 1634, 789]
[153, 123, 1634, 789]
153
1634
```

**2. WAJP to find second highest element frequency from an array.**

```
import java.util.*;
class SecondHighest
{
	public static void main(String[] args)
	{
		int[] arr = {5,5,2,2,1,3,1,4,4};
		Arrays.sort(arr);
		System.out.println(Arrays.toString(arr));
		int max1 = Integer.MIN_VALUE;
		int max2 = Integer.MIN_VALUE;
		for(int ele : arr)
		{
			if(max1<ele)
			{
				max2 = max1;
				max1 = ele;
			}
			else if(max2<max1 && ele!=max1)
				max2 = ele;
		}
		int cnt = 0;
		for (int ele : arr)
			if(max2==ele)
				cnt++;
		System.out.println(max2+" : "+cnt);
	}
}
```

**Output:-**

```
[1, 1, 2, 2, 3, 4, 4, 5, 5]
4 : 2
```

**3. WAJP of array rotation clockwise.**

```
import java.util.*;
class ArrayRotation
{
	public static void main(String[] args)
	{
		int[] a = {1,2,3,4,5};
		int rotation = 2;
		System.out.println(Arrays.toString(a));
		for(int i=1;i<=rotation;i++)
		{
			int temp = a[0];
			for(int j=1;j<a.length;j++)
				a[j-1] = a[j];
			a[a.length-1]=temp;
		}
		System.out.println(Arrays.toString(a));
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5]
[3, 4, 5, 1, 2]
```

**4. WAJP to find second smallest element frequency**

```
import java.util.*;
class SecondSmallestElementFrequency
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,6,7,7,6,6,1,1,2,2,3};
		Arrays.sort(arr);
		System.out.println(Arrays.toString(arr));
		int small1 = Integer.MAX_VALUE;
		int small2 = Integer.MAX_VALUE;
		for(int ele : arr)
		{
			if(small1>ele)
			{
				small2 = small1;
				small1 = ele;
			}
			else if(small2>ele && ele!=small1)
				small2 = ele;
		}
		int cnt=0;
		for (int ele : arr)
			if(small2==ele)
				cnt++;
		System.out.println(small2+" : "+cnt);
	}
}
```

**Output:-**

```
[1, 1, 1, 2, 2, 2, 3, 3, 4, 5, 6, 6, 6, 7, 7]
2 : 3
```

**5. WAJP of array rotation anticlockwise.**

```
import java.util.*;
class ArrayRotation2
{
	public static void main(String[] args)
	{
		int[] a = {1,2,3,4,5};
		int rotation = 1;
		System.out.println(Arrays.toString(a));
		for(int i=1;i<=rotation;i++)
		{
			int temp = a[a.length-1];
			for(int j=a.length-1;j>0;j--)
				a[j] = a[j-1];
			a[0]=temp;
		}
		System.out.println(Arrays.toString(a));
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5]
[5, 1, 2, 3, 4]
```

**6. Merge uncommon elements from array.**

```
import java.util.*;
class Q6
{
	public static void main(String[] args)
	{
		char[] a = {'A','A','B','C','D'};
		char[] b = {'A','B'};
		for (char ele : b)
		{
			for(int i=0;i<a.length;i++)
			{
				if(a[i]==ele)
					a[i]='_';
			}
		}
		for(int i=0;i<a.length;i++)
		{
			if(a[i]!='_')
				System.out.print(a[i]+" ");
		}
	}
}
```

**7. Convert String to character array.**

```
import java.util.*;
class StringToCharArray
{
	public static void main(String[] args)
	{
		String name = "AKSHAY";
		System.out.println(name);
		char[] ch = new char[name.length()];
		for(int i=0;i<name.length();i++)
		{
			ch[i] = name.charAt(i);
		}
		System.out.println(Arrays.toString(ch));
	}
}
```

## 8. Accenture Assessment program

```
import java.util.*;
class AssessmentAccentureQuestion
{
	public static void main(String[] args)
	{
		int[] sem1 = {40,60,35,48,69,80};
		int[] sem2 = {20,80,70,68,48,70};
		int[] res = new int[sem1.length];
		int no = 3;
		for(int i=0;i<sem1.length;i++)
		{
			res[i] = sem2[i]-sem1[i];	//logic to store difference of two arrays
		}
		Arrays.sort(res);	//arrays sorted in ascending order
		int totalMark = 0;
		for(int i = res.length-1,cnt = no;i>=0;i--)	//array started from last index
								//because array sorted in ascending order
			//so the highest element will be at the last index of array
		{
			if(res[i]>0 && cnt!=0)  //condition for checking the highest and
					// counter variable that keep track of highest mark subject
			{
				totalMark+=res[i];	//used to store totalmarks
				cnt--;
			}
		}
		if(totalMark>=35)
			System.out.println("Pass "+totalMark);
		else
			System.out.println("Fail "+totalMark);
	}
}
```

# ★ VARARGS:-

- varargs is a short name for variable arguments.
- In java a method can accept any number of arguments.
- An arguments which can accept any number of values is called as varargs.
- In order to define varargs three dots ( ... ) are used in the formal arguments of a method.
- ... → ellipsis
- E.g.

```
import java.util.*;
class VarargsExample
{
  public static void main(String[] args)
  {
        m1();
        m1(10);
        m1(10,20);
        m1(10,20,30);
        m1(10,20,30,40);
        m1(10,20,30,40,50);
  }
  public static int m1(int ... a)
  {
        System.out.println(Arrays.toString(a));
        System.out.println("_____");
  }
}
```

**Output:-**

```
[]
_____
[10]
_____
[10, 20]
_____
[10, 20, 30]
_____
[10, 20, 30, 40]
_____
[10, 20, 30, 40, 50]
_____
```

**Note:-**

- The three dots syntax tells the java compiler that the method can be called with 0 (zero) or any number of arguments.
- The variable declared for varargs is an array so we can access the elements using indexing.
- In case of no arguments the length of an array is zero.
- While defining varargs in a method, its declaration must be always at the last.
- A method can have only one varargs parameter.

**Overloading of varargs method.**

```
import java.util.*;
class VarargsExample2
{
	public static void main(String[] args)
	{
		m1();
		m1(10);
		m1(10,20);
		m1(10,20,30);
		m1(10,20,30,40);
		m1(10,20,30,40,50);
		m1("Akshay",10,20,30,40,50);
	}
	public static void m1(int ... a)
	{
		System.out.println(Arrays.toString(a));
		System.out.println("_____________________");
	}
	public static void m1(String str,int ... a)
	{
		System.out.println(str);
		System.out.println(Arrays.toString(a));
		System.out.println("_____________________");
	}
}
```

**Output:-**

```
[]
_____________________
[10]
_____________________
[10, 20]
_____________________
[10, 20, 30]
_____________________
[10, 20, 30, 40]
_____________________
[10, 20, 30, 40, 50]
_____________________
Akshay
[10, 20, 30, 40, 50]
_____________________
```

**WAJP to find the sum of user entered number.**

```
//WAJP for printing sum of user entered number
import java.util.*;
class SumOfElements
{
	public static void main(String[] args)
	{
		int op1,op2,op3,op4;
		op1 = add(10,20);
		op2 = add(10,20,30);
		op3 = add(10,20,30,40);
		op4 = add(10,20,30,40,50);
		System.out.println(op1);
		System.out.println(op2);
		System.out.println(op3);
		System.out.println(op4);
	}
	public static int add(int ... a)
	{
		int sum=0;
		for (int ele : a)
		{
			sum+=ele;
		}
		return sum;
	}
}
```

**Output:-**

```
30
60
100
150
```

# ★ Array sorting algorithm:-

- Sorting is a class of algorithm.
- In this we rearrange position of elements such that all of its element are either in ascending or descending order.
- A good sorting algorithm also needs to insure that elements having the same value should not change their positions.
- There are few popular sorting alogorithms such as,
  1. Bubble sort
  2. Selection sort
  3. Insertion sort
  4. Heap sort
  5. Merge sort

## 1. Bubble sort:-

- Bubble sort is a easiest sorting algorithm ever.
- It swaps adjacent elements if they are in the wrong order, this is done repeatedly.
- This process is repeated for every element in the list until no more swaps are needed.

  **Steps:-**
  i. Start from the first element of an array.
  ii. Compare it with the next element (adjacent element).
  iii. If the current element is greater than the next element swap them.
  iv. Move to the next pair and repeat the process until the end of an array.
  v. Repeat the entire process till length-1 from step ii to step iv.
  vi. Each iteration of the loop pushes the largest unsorted element to its correct position.

**Bubble sort algorithm for ascending sorting of an array.**

```
import java.util.*;
class BubbleSortAscending{
	public static void main(String[] args) {
		int[] arr = {9,6,5,3,4,7,8,2,1};
		System.out.println(Arrays.toString(arr));
		bubbleSort(arr);
		System.out.println(Arrays.toString(arr));
	}
	public static void bubbleSort(int[] arr)
	{
		for(int i=0;i<arr.length;i++)
		{
			for(int j=i+1;j<arr.length;j++)
			{
				if(arr[i]>arr[j])
				{
					int temp = arr[i];
					arr[i] = arr[j];
					arr[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
[9, 6, 5, 3, 4, 7, 8, 2, 1]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Bubble sort algorithm for descending sorting of an array.**

```
import java.util.*;
class BubbleSortDescending
{
	public static void main(String[] args)
	{
		int[] arr = {9,6,5,3,4,7,8,2,1};
		System.out.println(Arrays.toString(arr));
		bubbleSort(arr);
		System.out.println(Arrays.toString(arr));
	}
	public static void bubbleSort(int[] arr)
	{
		for(int i=0;i<arr.length;i++)
		{
			for(int j=i+1;j<arr.length;j++)
			{
				if(arr[i]<arr[j])
				{
					int temp = arr[i];
					arr[i] = arr[j];
					arr[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
[9, 6, 5, 3, 4, 7, 8, 2, 1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

**Note:-**

- The time complexity of bubble sort is $O(n^2)$ (worst time complexity).
- If an array is already sorted and only one swap is required the time complexity will be O(n).
- Bubble sort is most oftenly used for small datasets.
- Its not efficient for large dataset.

## Q. 1 Sort an int array in desecding order.

```
import java.util.*;
class BubbleSortDescending{
	public static void main(String[] args) {
		int[] arr = {9,6,5,3,4,7,8,2,1};
		System.out.println(Arrays.toString(arr));
		bubbleSort(arr);
		System.out.println(Arrays.toString(arr));
	}
	public static void bubbleSort(int[] arr){
		for(int i=0;i<arr.length;i++){
			for(int j=i+1;j<arr.length;j++){
				if(arr[i]<arr[j]){
					int temp = arr[i];
					arr[i] = arr[j];
					arr[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
[9, 6, 5, 3, 4, 7, 8, 2, 1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

## Q. 2 Find the first largest and smallest element from the array.

```
import java.util.*;
class LargestAndSmallest{
	public static void main(String[] args) {
		int[] a = {80,90,40,50,60,10,20,30,70};
		bubbleSort(a);
		System.out.println(Arrays.toString(a));
		System.out.println("Smallest: "+a[0]);
		System.out.println("Largest: "+a[a.length-1]);
	}
	public static void bubbleSort(int[] a){
		for(int i=0;i<a.length;i++){
			for(int j=i+1;j<a.length;j++){
				if(a[i]>a[j]){
					int temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
Smallest: 10
Largest: 90
```

**Q. 3 Find second largest element and second smallest element from an array.**

```
import java.util.*;
class SecondLargeSecondSmall
{
	public static void main(String[] args)
	{
		int[] a = {80,90,40,50,60,10,20,30,70,10,90,10,90};
		bubbleSort(a);
		System.out.println(Arrays.toString(a));
		int small1 = a[0];
		int large1 = a[a.length-1];
		int small2 = Integer.MAX_VALUE;
		int large2 = Integer.MIN_VALUE;
		for(int i : a)
		{
			if(small1<i&&i<small2){
				small2=i;
			}
			if(i<large1&&i>large2){
				large2=i;
			}
		}
		System.out.println("Second smallest element : "+small2);
		System.out.println("Second largest element : "+large2);
	}
	public static void bubbleSort(int[] a)
	{
		for(int i=0;i<a.length;i++)
		{
			for(int j=i+1;j<a.length;j++)
			{
				if(a[i]>a[j])
				{
					int temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
[10, 10, 10, 20, 30, 40, 50, 60, 70, 80, 90, 90, 90]
Second smallest element : 20
Second largest element : 80
```

**Q. 4 Student array example**

**i. sort using student id in ascending order**

```
import java.util.*;
class Student{
	int sid;
	String name;
	String branch;
	int yop;
	double cgpa;
	Student(int sid,String name,String branch,int yop,double cgpa){
		this.sid = sid;
		this.name = name;
		this.branch = branch;
		this.yop = yop;
		this.cgpa = cgpa;
	}
	@Override
	public String toString(){
		return sid+"."+name+", Branch : "+branch+", YOP : "+yop+", CGPA : "+cgpa;
	}
}
class SortExample1{
	public static void main(String[] args) {
		Student[] a = new Student[5];
		a[0] = new Student(3,"Akshay","CS",2024,8.87);
		a[1] = new Student(5,"Tanmay","IT",2022,9.0);
		a[2] = new Student(2,"Vishwajeet","CS",2023,9.5);
		a[3] = new Student(4,"Aditya","MECH",2020,7.5);
		a[4] = new Student(1,"RITIK","AIDS",2024,9.1);
		bubbleSort(a);
		for(Student i : a){
			System.out.println(i.toString());
		}
	}
	public static void bubbleSort(Student[] a){
		for(int i=0;i<a.length;i++){
			for(int j=i+1;j<a.length;j++){
				if(a[i].sid>a[j].sid){
					Student temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

1.RITIK, Branch : AIDS, YOP : 2024, CGPA : 9.1
2.Vishwajeet, Branch : CS, YOP : 2023, CGPA : 9.5
3.Akshay, Branch : CS, YOP : 2024, CGPA : 8.87
4.Aditya, Branch : MECH, YOP : 2020, CGPA : 7.5
5.Tanmay, Branch : IT, YOP : 2022, CGPA : 9.0

### ii. Sort using name in ascending order

```
import java.util.*;
class Student{
	int sid;
	String name;
	String branch;
	int yop;
	double cgpa;
	Student(int sid,String name,String branch,int yop,double cgpa){
		this.sid = sid;
		this.name = name;
		this.branch = branch;
		this.yop = yop;
		this.cgpa = cgpa;
	}
	@Override
	public String toString(){
		return sid+"."+name+", Branch : "+branch+", YOP : "+yop+", CGPA : "+cgpa;
	}
}
class SortExample1{
	public static void main(String[] args) {
		Student[] a = new Student[5];
		a[0] = new Student(3,"Akshay","CS",2024,8.87);
		a[1] = new Student(5,"Tanmay","IT",2022,9.0);
		a[2] = new Student(2,"Vishwajeet","CS",2023,9.5);
		a[3] = new Student(4,"Aditya","MECH",2020,7.5);
		a[4] = new Student(1,"RITIK","AIDS",2024,9.1);
		bubbleSort(a);
		for(Student i : a){
			System.out.println(i.toString());
		}
	}
	public static void bubbleSort(Student[] a){
		for(int i=0;i<a.length;i++){
			for(int j=i+1;j<a.length;j++){
				if(a[i].name.compareTo(a[j].name)>0){
					Student temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
4.Aditya, Branch : MECH, YOP : 2020, CGPA : 7.5
3.Akshay, Branch : CS, YOP : 2024, CGPA : 8.87
1.RITIK, Branch : AIDS, YOP : 2024, CGPA : 9.1
5.Tanmay, Branch : IT, YOP : 2022, CGPA : 9.0
2.Vishwajeet, Branch : CS, YOP : 2023, CGPA : 9.5
```

### iii. Sort using year of pass out in descending order

```
import java.util.*;
class Student{
	int sid;
	String name;
	String branch;
	int yop;
	double cgpa;
	Student(int sid,String name,String branch,int yop,double cgpa){
		this.sid = sid;
		this.name = name;
		this.branch = branch;
		this.yop = yop;
		this.cgpa = cgpa;
	}
	@Override
	public String toString(){
		return sid+"."+name+", Branch : "+branch+", YOP : "+yop+", CGPA : "+cgpa;
	}
}
class SortExample1{
	public static void main(String[] args) {
		Student[] a = new Student[5];
		a[0] = new Student(3,"Akshay","CS",2024,8.87);
		a[1] = new Student(5,"Tanmay","IT",2022,9.0);
		a[2] = new Student(2,"Vishwajeet","CS",2023,9.5);
		a[3] = new Student(4,"Aditya","MECH",2020,7.5);
		a[4] = new Student(1,"RITIK","AIDS",2024,9.1);
		bubbleSort(a);
		for(Student i : a){
			System.out.println(i.toString());
		}
	}
	public static void bubbleSort(Student[] a){
		for(int i=0;i<a.length;i++){
			for(int j=i+1;j<a.length;j++){
				if(a[i].yop<a[j].yop){
					Student temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
3.Akshay, Branch : CS, YOP : 2024, CGPA : 8.87
1.RITIK, Branch : AIDS, YOP : 2024, CGPA : 9.1
2.Vishwajeet, Branch : CS, YOP : 2023, CGPA : 9.5
5.Tanmay, Branch : IT, YOP : 2022, CGPA : 9.0
4.Aditya, Branch : MECH, YOP : 2020, CGPA : 7.5
```

### iv. Sort using CGPA in ascending order

```
import java.util.*;
class Student{
	int sid;
	String name;
	String branch;
	int yop;
	double cgpa;
	Student(int sid,String name,String branch,int yop,double cgpa){
		this.sid = sid;
		this.name = name;
		this.branch = branch;
		this.yop = yop;
		this.cgpa = cgpa;
	}
	@Override
	public String toString(){
		return sid+"."+name+", Branch : "+branch+", YOP : "+yop+", CGPA : "+cgpa;
	}
}
class SortExample1{
	public static void main(String[] args) {
		Student[] a = new Student[5];
		a[0] = new Student(3,"Akshay","CS",2024,8.87);
		a[1] = new Student(5,"Tanmay","IT",2022,9.0);
		a[2] = new Student(2,"Vishwajeet","CS",2023,9.5);
		a[3] = new Student(4,"Aditya","MECH",2020,7.5);
		a[4] = new Student(1,"RITIK","AIDS",2024,9.1);
		bubbleSort(a);
		for(Student i : a){
			System.out.println(i.toString());
		}
	}
	public static void bubbleSort(Student[] a){
		for(int i=0;i<a.length;i++){
			for(int j=i+1;j<a.length;j++){
				if(a[i].cgpa>a[j].cgpa){
					Student temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
4.Aditya, Branch : MECH, YOP : 2020, CGPA : 7.5
3.Akshay, Branch : CS, YOP : 2024, CGPA : 8.87
5.Tanmay, Branch : IT, YOP : 2022, CGPA : 9.0
1.RITIK, Branch : AIDS, YOP : 2024, CGPA : 9.1
2.Vishwajeet, Branch : CS, YOP : 2023, CGPA : 9.5
```

### v. Sort using cgpa in descending order

```
import java.util.*;
class Student{
	int sid;
	String name;
	String branch;
	int yop;
	double cgpa;
	Student(int sid,String name,String branch,int yop,double cgpa){
		this.sid = sid;
		this.name = name;
		this.branch = branch;
		this.yop = yop;
		this.cgpa = cgpa;
	}
	@Override
	public String toString(){
		return sid+"."+name+", Branch : "+branch+", YOP : "+yop+", CGPA : "+cgpa;
	}
}
class SortExample1{
	public static void main(String[] args) {
		Student[] a = new Student[5];
		a[0] = new Student(3,"Akshay","CS",2024,8.87);
		a[1] = new Student(5,"Tanmay","IT",2022,9.0);
		a[2] = new Student(2,"Vishwajeet","CS",2023,9.5);
		a[3] = new Student(4,"Aditya","MECH",2020,7.5);
		a[4] = new Student(1,"RITIK","AIDS",2024,9.1);
		bubbleSort(a);
		for(Student i : a){
			System.out.println(i.toString());
		}
	}
	public static void bubbleSort(Student[] a){
		for(int i=0;i<a.length;i++){
			for(int j=i+1;j<a.length;j++){
				if(a[i].cgpa<a[j].cgpa){
					Student temp = a[i];
					a[i] = a[j];
					a[j] = temp;
				}
			}
		}
	}
}
```

**Output:-**

```
2.Vishwajeet, Branch : CS, YOP : 2023, CGPA : 9.5
1.RITIK, Branch : AIDS, YOP : 2024, CGPA : 9.1
5.Tanmay, Branch : IT, YOP : 2022, CGPA : 9.0
3.Akshay, Branch : CS, YOP : 2024, CGPA : 8.87
4.Aditya, Branch : MECH, YOP : 2020, CGPA : 7.5
```

# 2. Selection sort:-

- Selection sort is a sorting algorithm that works by repeatedly selecting the smallest element from an unsorted array, and then it swaps it with the first unsorted element.

  **Steps:-**

  i. Store the index of first element.
  ii. Find the index of smallest element.
  iii. Swap the elements based on the indexes.
  iv. Repeat steps 1 to 3 until the arrays completely sorted.

  E.g.

```
import java.util.Arrays;
class SelectionSort
{
    public static void main(String[] args)
    {
            int[] a = {9,6,7,1,2,3,4,5,8};
            System.out.println(Arrays.toString(a));
            selectionSort(a);
            System.out.println(Arrays.toString(a));
    }
    public static void selectionSort(int[] a)
    {
            for(int i=0;i<a.length;i++)
            {
                    int indx = i;
                    for(int j=i+1;j<a.length;j++)
                    {
                            if(a[indx]>a[j])
                            {
                                    indx = j;
                            }
                    }
                    int temp = a[i];
                    a[i] = a[indx];
                    a[indx] = temp;
            }
    }
}
```

  **Output:-**

```
[9, 6, 7, 1, 2, 3, 4, 5, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Note:-**

- Selection sort is not efficient for large dataset, due to worst time complexity $O(n^2)$.
- It is used for small datasets and when we need to minimize the number of swaps.

**WAJP to sort a name array in descending using selection sort.**

```
import java.util.*;
class StringSorting
{
	public static void main(String[] args)
	{
		String[] name = {"Akshay","Aditya","Ritik","Tanmay","Sreyas"};
		System.out.println(Arrays.toString(name));
		selectionSort(name);
		System.out.println(Arrays.toString(name));
	}
	public static void selectionSort(String[] a)
	{
		for(int i=0;i<a.length;i++)
		{
			int indx=i;
			for(int j=i+1;j<a.length;j++)
			{
				if(a[indx].compareTo(a[j])<0)
				{
					indx = j;
				}
			}
			String temp = a[i];
			a[i] = a[indx];
			a[indx] = temp;
		}
	}
}
```

**Output:-**

[Akshay, Aditya, Ritik, Tanmay, Sreyas]
[Tanmay, Sreyas, Ritik, Akshay, Aditya]

1. **Insertion sort:-**
   - Insertion sort is a simple sorting algorithm that works similar to playing cards in your hands.
   - It builds sorted array one element at a time by comparing and inserting each element into its correct position.

   **Algorithm steps:-**
   i. Assume the first element is already sorted.
   ii. Pick the next element and compare it with the elements in the sorted portion.
   iii. Shift all the larger elements in the sorted portion towards the right to make space for the current element.
   iv. Now insert the current element into its correct position.
   v. Repeat the steps until the entire array is sorted.

**WAJP to sort the given array in ascending order using insertion sort algorithm.**
**int[] a = {7,9,1,3,5,4,2,6,8};**

```
import java.util.Arrays;
class InsertionSortAscending{
	public static void main(String[] args) {
		int[] a = {7,9,1,3,5,4,2,6,8};
		System.out.println(Arrays.toString(a));
		insertionSort(a);
		System.out.println(Arrays.toString(a));
	}
	public static void insertionSort(int[] a){
		for(int i=1;i<a.length;i++){
			int key = a[i];
			int j=i-1;
			while(j>=0 && a[j]>key){
				a[j+1] = a[j];
				j--;
			}
			a[j+1] = key;
		}
	}
}
```

**Output:-**

```
[7, 9, 1, 3, 5, 4, 2, 6, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**WAJP to sort the given array in descending order using insertion sort algorithm.**
**int[] a = {7,9,1,3,5,4,2,6,8};**

```
import java.util.Arrays;
class InsertionSortDescending{
	public static void main(String[] args) {
		int[] a = {7,9,1,3,5,4,2,6,8};
		System.out.println(Arrays.toString(a));
		insertionSort(a);
		System.out.println(Arrays.toString(a));
	}
	public static void insertionSort(int[] a){
		for(int i=1;i<a.length;i++){
			int key = a[i];
			int j=i-1;
			while(j>=0 && a[j]<key){
				a[j+1] = a[j];
				j--;
			}
			a[j+1] = key;
		}
	}
}
```

**Output:-**

```
[7, 9, 1, 3, 5, 4, 2, 6, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

## WAJP to sort an array using bubble sort (recursion).

```
//bubble sort algorithm using method recursion
import java.util.*;
class BubbleSortUsingMethodRecursion
{
	static int i=0;
	public static void main(String[] args)
	{
		int[] a = {5,4,6,3,7,2,8,1,9};
		System.out.println(Arrays.toString(a));
		recursionSort(a);
		System.out.println(Arrays.toString(a));
	}
	public static void recursionSort(int[] a)
	{
		for(int j=i+1;j<a.length;j++)
		{
			if(a[i]>a[j])
			{
				int temp = a[i];
				a[i] = a[j];
				a[j] = temp;
			}
		}
		i++;
		if(i==a.length)
			return;
		recursionSort(a);
	}
}
```

**Output:-**

```
[5, 4, 6, 3, 7, 2, 8, 1, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### ★ Arrays.sort():-

- Arrays.sort() is built in method in java which is used to sort an array efficiently.
- This method belongs to Arrays class and it is part of java.util package.
- It is fast and flexible way to sort primitive and non-primitive arrays.
- Arrays.sort() is an overloaded method.
- If we are using this method for a primitive array it uses Dual Pivot Quick Sort Algorithm.
- If an array is non-primitive i.e. object array it uses Tim Sort Algorithm which is combination of insertion sort and merge sort.

**Note:-**

- This sorting algorithm uses $O(n^2)$ time complexity in worst case.
- Also we cannot sort an array in descending manner.

**E.g.:-**

```
import java.util.Arrays;
class Demo2
{
	public static void main(String[] args)
	{
		String[] a ={"akshay","aditya","tanmay","shreyas"};
		System.out.println(Arrays.toString(a));
		Arrays.sort(a);
		System.out.println(Arrays.toString(a));
	}
}
```

**Output:-**

[akshay, aditya, tanmay, shreyas]
[aditya, akshay, shreyas, tanmay]

- **Dual Pivot Quick Sort Algorithm:-**
  - Its an optimise and traditional version of Quick sort algorithm.
  - Instead of using a single pivot it uses dual pivot effectively dividing the array into three parts.
  - Two pivot elements are selected P1 and P2 such that P1 should be less than equals to P2 if not swap them.
  - This pivots P1 and P2 divides the array into three parts,
    i. Elements smaller than P1 will be stored in first part.
    ii. Elements greater than P2 will be stored in third part and the elements between P1 and P2 will be stored in the middle part.
  - This steps will be repeated until the array is sorted.

- **Tim Sort Algorithm:-**
  - Its an combination of merge sort and insertion sort.
  - It is most efficient sorting algorithm for real world based datasets.
  - It uses a sub array i.e. called as run which is either already or can be sorted with less efforts.
  - Tim sort algorithm identifies the run in an array and process them instead of starting from scratch.
  - All the runs are merged using merge sort approach for creating a fully sorted array.

**Example of Arrays class and Comparator using Arrays.sort() sorting array in reverse order.**

```
import java.util.Arrays;
import java.util.Comparator;
class ArraysClassMethod
{
	public static void main(String[] args)
	{
		String[] a = {"z","x","a","f","i"};
		System.out.println(Arrays.toString(a));
		Arrays.sort(a);
		Arrays.sort(a,Comparator.reverseOrder());
		System.out.println(Arrays.toString(a));

		Integer[] b = {5,9,3,4,7,1,6,2,8};
		System.out.println(Arrays.toString(b));
		Arrays.sort(b);
		Arrays.sort(b,Comparator.reverseOrder());
		System.out.println(Arrays.toString(b));

	}
}
```

**Output:-**

```
[z, x, a, f, i]
[z, x, i, f, a]
[5, 9, 3, 4, 7, 1, 6, 2, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

**Note:-**

We cannot reverse the primitive int type that's why we are using corresponding Wrapper class.

**Arrays.sort(variable_name,int startIndex, int endIndex);**

**E.g.**

```
import java.util.Arrays;
class Demo3
{
	public static void main(String[] args)
	{
		int[] arr = {5,6,7,4,3,8,2,9,1};
		System.out.println(Arrays.toString(arr));
		Arrays.sort(arr,0,5);
		System.out.println(Arrays.toString(arr));
	}
}
```

**Output:-**

```
[5, 6, 7, 4, 3, 8, 2, 9, 1]
[3, 4, 5, 6, 7, 8, 2, 9, 1]
```

# ★ Searching Algorithms:-

- Searching algorithms are used to find location of element in a dataset (arrays, list, other data structures).
- There are few famous searching algorithms such as,
  1. Linear search
  2. Binary search
  3. Jump search
  4. Interpolation search

## 1. Linear search:-

- Linear search is one of the simpliest searching algorithm.
- It sequencially checks each element of an array until the desired element is found or entire array has been traversed.

**Algorithm steps:-**

i. Start from the first element of the array.
ii. Compare each element with the target element.
iii. If the target is found return the index of the element.
iv. If the target is not found after checking all the elements return -1.

**Program:-**

```
import java.util.*;
class LinearSearch
{
	public static void main(String[] args)
	{
		int [] arr = new int[30];
		for(int i=0;i<arr.length;i++)
		{
			arr[i] = i+1;
		}
		System.out.println(Arrays.toString(arr));
		int ele = 30;
		int pos = searchElement(arr,ele);
		System.out.println("Element "+ele+" Found at "+pos+" position.");
	}
	public static int searchElement(int[] arr,int ele)
	{
		for(int i=0;i<arr.length;i++)
		{
			if(ele==arr[i])
				return i;
		}
		return -1;
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
Element 30 Found at 29 position.
```

# 2. Binary search:-

- Binary search is an efficient algorithm used to find the position of target element in a sorted array.
- It works by repeatedly dividing the search space in half and checks for the middle element.

**Algorithm steps:-**

i. Set the search range with two pointers i.e. **start** (starting index of array) and **end** (last index of array).
ii. Find the middle element,
**start+end/2 = mid**
iii. Compare the middle element with the target,
   a. If the middle element is equals to the target return index (**mid**).
   b. If the target is smaller than the middle element,
   Search in the left half and update the end index by **mid-1**
   c. If the target is larger than the middle element,
   Seach in the right half and update the start index by **mid+1**
iv. Repeat this step until the range is invalid i.e. **start<=end**

**Note:-**
Binary search used O(logn) time complexity for large data sets.

**Program:-**

```
import java.util.Arrays;
class BinarySearch
{
	public static void main(String[] args)
	{
		int[] arr = {1,2,3,4,5,6,7,8,9};
		int ele = 1;
		int pos = binarySearch(arr,ele);
		System.out.println("Element "+ele+" found at "+pos+" position.");
	}
	public static int binarySearch(int[] arr,int ele)
	{
		int start = 0;
		int end = arr.length-1;
		while(start<=end)
		{
			int mid = (start+end)/2;
			if(arr[mid]<ele)
				start = mid+1;
			else if(arr[mid]>ele)
				end = mid-1;
			else
				return mid;
		}
		return -1;
	}
}
```

**Output:-**

```
Element 1 found at 0 position.
```

# ★ String class:-

- String is a predefined, built in and final class in java which is derived in java.lang package.
- String is a collection of characters that are enclosed within double quotes (“ ”).
- Strings are case sensitive.
- It is immutable in nature (i.e. cannot be changed).
- Strings are immutable because of class loading, security, synchronization, hashcode, chaching.
- Strings can be created using 3 classes,
  i. String
  ii. StringBuffer
  iii. StringBuilder

★ **Why String is immutable?**

→

**i. Security:-**

- Strings are used to hold case sensitive data such as connection URL, Username, Password, File paths, etc.
- If Strings were mutable someone can modify there content after they have been passed.

**ii. SCP (String Constant Pool):-**

- Java uses special area i.e. String Constant Pool (heap area) to occupies memory usage.
- When we create a String using String syntax i.e.
  String str = “hello”;
  The JVM checks if an identical string already exists in the pool if not a new String will be created or else the reference of existing object is returned.

**iii. Synchronization:-**

- Strings are immtable because of which thread safe.
- When the String is shared between multiple threads its immutability gurantees that its value cannot be changed.
- This eliminates the need of synchronization and makes String more simplier and efficient for multithreading.

**E.g. 1**

```
class Demo
{
  public static void main(String[] args)
  {
        String a = "hello";
        String b = "hello";
        String c = new String("hello");
        System.out.println(a==b);
        System.out.println(a==c);
        System.out.println(b==c);
        System.out.println(a.equals(b));
        System.out.println(a.equals(c));
  }
}
```

**Output:-**

```
true
false
false
true
true
```

**E.g. 2**

```
class Demo
{
	public static void main(String[] args)
	{
		String a = new String("hello world");
		String b = "hello "+"world";
		String c = "hello world";
		String d = "hello ".concat("world");
		System.out.println(a==b);
		System.out.println(b==c);
		System.out.println(b==d);
		System.out.println(a==d);
		System.out.println(a.equals(b));
		System.out.println(b.equals(c));
		System.out.println(c.equals(d));
	}
}
```

**Output:-**

```
false
true
false
false
true
true
true
```

**Note:-**

- Strings internally stores the data in a form of character array i.e. char[], before JDK 1.9.
- From JDK 1.9 onwoards for memory utilization Strings are stored in form of byte[].
- **String is a final class.**

### ★ Constructuors of String class:-

1. public String()
2. public String(String str)
3. public String(StringBuffer sb)
4. public String(StringBuilder sb)
5. public String(char[] arr)
6. public String(char[] arr, int offset, int count)
7. public String(byte[] arr)
8. public String(byte[] arr, int , int )

**1. public String():-**

- It initialize a newly created String object, that it represent an empty character sequence.
- Note that use of this constructor is unessessary since, Strings are immutable.
- E.g.

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

2. **public String(String str):-**
   - It initialize a newly created String object so that it represents the same sequence of character as specified in the argument.
   - E.g.

```
class StringArgumentConstructor
{
    public static void main(String[] args)
    {
            String str = new String("Akshay");
            System.out.println(str);
    }
}
```

**Output:-**
Akshay

3. **public String(StringBuffer sb):-**
   - It allocates a new String that contain the sequence of character currently contained in the StringBuffer argument.
   - E.g.

```
class StringBufferArgumentConstructor
{
    public static void main(String[] args)
    {
            StringBuffer sb = new StringBuffer("Akshay");
            String str = new String(sb);
            System.out.println(str);
    }
}
```

**Output:-**
Akshay

4. **public String(StringBuiler sb):-**
   - It allocates a new String that contain the sequence of character currently contained in the StringBuilder agument.
   - E.g.

```
class StringBuilderArgumentConstructor
{
    public static void main(String[] args)
    {
            StringBuilder sb = new StringBuilder("Akshay");
            String str = new String(sb);
            System.out.println(str);
    }
}
```

**Output:-**
Akshay

5. **public String(char[] arr):-**
   - It allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
   - E.g.

```
import java.util.*;
class CharArrayArgumentConstructor
{
    public static void main(String[] args)
    {
            char[] arr = {'A','k','s','h','a','y'};
            System.out.println(Arrays.toString(arr));
            String str = new String(arr);
            System.out.println(str);
    }
}
```

**Output:-**

```
[A, k, s, h, a, y]
Akshay
```

6. **public String(char[] arr,int offset, int count):-**
   - It allocates a new String that contains character from a subarray of the specified character array argument.
   - The offset argument is the start index and the count is the number of character.
   - E.g.

```
class CharArrayArgumentConstructorTwo
{
    public static void main(String[] args)
    {
            char[] arr = {'M','A','H','A','R','A','S','H','T','R','A'};
            String str = new String(arr,0,4);
            System.out.println(str);
    }
}
```

**Output:-**

```
MAHA
```

7. **public String(byte[] arr):-**
   - It initialize a new String object by converting all the specified argument of a given array into its Unicode value.
   - E.g.

```
import java.util.Arrays;
class ByteArrayArgumentConstructor
{
    public static void main(String[] args)
    {
            byte[] arr = {65,107,115,104,97,121};
            System.out.println(Arrays.toString(arr));
            String str = new String(arr);
            System.out.println(str);
    }
}
```

**Output:-**

```
[65, 107, 115, 104, 97, 121]
Kasha
```

8. **public String(byte[] arr,int ,int):-**
   - It constructs a new String by converting (decoding) all the elements into its Unicode character of a specified subarray.
   - E.g.

```
import java.util.Arrays;
class ByteArrayArgumentConstructorTwo
{
    public static void main(String[] args)
    {
            byte[] arr = {65,107,115,104,97,121};
            System.out.println(Arrays.toString(arr));
            String str = new String(arr,0,2);
            System.out.println(str);
    }
}
```

**Output:-**

```
[65, 107, 115, 104, 97, 121]
Ak
```

## ★ Methods of String:-

1. public int length()
2. public boolean isEmpty()
3. public char charAt(int index)
4. public int codePointAt(int index)
5. public int codePointBefore(int index)
6. public int codePointCount(int start, int end)
7. public boolean equals(Object obj)
8. public boolean contentEquals(StringBuffer sb)
9. public int compareTo(String str)
10. public boolean startsWith(String str)
11. public boolean endsWith(String str)
12. public int hashCode()
13. public int indexOf(int asciiValue)
14. public int indexOf(int asciiValue, int startIndex)
15. public int lastIndexOf(int asciiValue)
16. public int lastIndexOf(int asciiValue, int startIndex)
17. public String subString(int startIndex)
18. public String subString(int startIndex, int endIndex)
19. public String concat(String str)
20. public String replace(char oldChar, char newChar)
21. public String replaceFirst(String regex, String replacement)
22. public String replaceAll(String regex, String replacement)
23. public String[] split(String regex)
24. public String toLowerCase()
25. public String toUpperCase()
26. public String trim()
27. public char[] toCharArray()
28. public boolean equalsIgnoreCase(String str)

1. **public int length():-**
   - This method returns the length of this String i.e. equals to the number of characters contains in the String.
   - E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str="";
            for (char ch : arr)
                    str+=ch;
            return str;
    }
    public int length(){
            return arr.length;
    }
}
class UserLength
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.length());
    }

}
```

**Output:-**

6

2. **public boolean isEmpty():-**
   - This method returns true if the length of String is zero or else retrns false if the length is greater than zero.
   - E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str="";
            for (char ch : arr) {
                    str+=ch;
            }
            return str;
    }
    public boolean isEmpty(){
            if(arr.length!=0)
                    return false;
            return true;
    }
}
class UserIsEmpty
{
    public static void main(String[] args)
    {
            UserString str1 = new UserString();
            System.out.println(str1.isEmpty());
            UserString str2 = new UserString("Akshay");
            System.out.println(str2.isEmpty());
    }
}
```

**Output:-**

```
true
false
```

### 3. public char charAt(int index):-

- It returns the char value from a specified index an index ranges from 0 to length -1.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str ="";
            for (char ch : arr)
                    str+=ch;
            return str;
    }
    public char charAt(int index){
            if(index<0||index>=arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index");
            return arr[index];
    }
}
class UserCharAt
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.charAt(0));
    }
}
```

**Output:-**

```
A
```

### 4. public int codePointAt(int index):-

- It returns the characters Unicode values from the specified index.
- The index refers to a character value that ranges from 0 to length-1.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for (char ch : arr)
                    str+=ch;
            return str;
    }
    public int codePointAt(int index){
            if(index<0 || index>=arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index : "+index);
            return arr[index]+0;
    }
}
class UserCodePointAt
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.codePointAt(0));
    }
}
```

**Output:-**

65

5. **public int codePointBefore(int index):-**
   - It return a character Unicode value which is before the specified index.
   - The index refers to char values ranges from 1 to length.
   - E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for (char ch : arr)
                    str+=ch;
            return str;
    }
    public int codePointBefore(int index){
            if(index<1 || index>arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index : "+index);
            return arr[index-1]+0;
    }
}
class UserCodePointBefore
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.codePointBefore(3));
    }
}
```

**Output:-**

115

### 6. public int codePointCount(int startIndex, int endIndex):-

- This method returns the number of character within the specified range where start index character is included and end index character is excluded.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for (char ch : arr)
                    str+=ch;
            return str;
    }
    public int codePointCount(int startIndex,int endIndex){
            if(startIndex<0 || endIndex>arr.length || startIndex>endIndex)
            throw new IndexOutOfBoundsException("Range ["+startIndex+","+endIndex+"]
                                                                    out of bounds");
            return endIndex-startIndex;
    }
}
class UserCodePointCount
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.codePointCount(2,5));
    }
}
```

**Output:-**

3

### 7. public boolean equals(Object obj):-

- This method compairs this String with specified object argument.
- It returns true if the Objects contains same character sequence of else return false.
- E.g.

```
final class UserString{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch : arr)
                    str+=ch;
            return str;
    }
    @Override
    public boolean equals(Object obj){
            String str1 = (String)obj;
            UserString str2 = new UserString(str1);
            if(this.length()!=str2.length())
                    return false;
            int indx=0;
            for (char ch : arr) {
                    if(ch==str2.charAt(indx++))
                            continue;
                    else
                            return false;
            }
            return true;
    }
    public int length(){
            return arr.length;
    }
    public char charAt(int indx){
            if(indx<0 || indx>=arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index : "+indx);
            return arr[indx];
    }
}
class UserEquals{
    public static void main(String[] args) {
            UserString str = new UserString("Akshay");
            System.out.println(str.equals("Akshay"));
    }
}
```

**Output:-**

```
true
```

### 8. public boolean equalsIgnoreCase(String str):-

- This method compaires this String with another String ignoring rare consederation.
- It returns true if both the Object conatins same character sequence ignoring case or else returns false.
- E.g.

```
final class UserString{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch : arr)
                    str+=ch;
            return str;
    }
    @Override
    public boolean equals(Object obj){
            UserString str2 = (UserString)obj;
            if(this.length()!=str2.length())
                    return false;
            int indx=0;
            for (char ch : arr) {
                    if(ch==str2.charAt(indx++))
                            continue;
                    else
                            return false;
            }
            return true;
    }
    public int length(){
            return arr.length;
    }
    public char charAt(int indx){
            if(indx<0 || indx>=arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index : "+indx);
            return arr[indx];
    }
    public boolean equalsIgnoreCase(UserString obj){
            return this.toUpperCase().equals(obj.toUpperCase());
    }
    public UserString toUpperCase(){
            String str="";
            for(char ch : arr){
                    if(ch>=97&&ch<=122)
                            str+=(char)(ch-32);
                    else
                            str+=ch;
            }
            return new UserString(str);
    }
}
class UserEqualsIgnoreCase{
    public static void main(String[] args) {
```

```
        UserString str1 = new UserString("AKSHAY");
        UserString str2 = new UserString("akshay");
        System.out.println(str1.equalsIgnoreCase(str2));
    }
}
```

**Output:-**

true

**9. public boolean contentEquals(StringBuffer sb):-**

- this method compares this String with the specified buffer object, and returns true if both conatins same character sequence or else return false.
- E.g.

```
final class UserString{
    char[] arr;
    UserString(){
        arr = new char[0];
    }
    UserString(String str){
        arr = new char[str.length()];
        for(int i=0;i<arr.length;i++)
            arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
        String str = "";
        for(char ch : arr)
            str+=ch;
        return str;
    }
    public boolean contentEquals(StringBuffer sb){
        if(arr.length!=sb.length())
            return false;
        for(int i=0;i<arr.length;i++)
            if(arr[i]!=sb.charAt(i))
                return false;
        return true;
    }
    public boolean contentEquals(StringBuilder sb){
        if(arr.length!=sb.length())
            return false;
        for(int i=0;i<arr.length;i++)
            if(arr[i]!=sb.charAt(i))
                return false;
        return true;
    }
}
class UserContentEquals{
    public static void main(String[] args) {
        UserString str1 = new UserString("Akshay");
        StringBuilder str2 = new StringBuilder("Akshay ");
        System.out.println(str1.contentEquals(str2));
    }
}
```

**Output:-**

true

**10. public int compareTo(String str):-**

- compareTo() method compares two Strings Pexigraphically -1.
- The comparison is based on Unicode values of each character in String.
- It returns zero if both the String values are equal.
- It returns one of 1st String greater than 2nd.

- It returns positive value i.e. substraction of Unicode of both String.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch : arr)
                    str+=ch;
            return str;
    }
    public int length(){
            return arr.length;
    }
    public char charAt(int indx){
            if(indx<0 || indx>=arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index : "+indx);
            return arr[indx];
    }
    @Override
    public boolean equals(Object obj){
            String str1 = (String)obj;
            UserString str2 = new UserString(str1);
            if(this.length()!=str2.length())
                    return false;
            int indx=0;
            for (char ch : arr) {
                    if(ch==str2.charAt(indx++))
                            continue;
                    else
                            return false;
            }
            return true;
    }
    public int compareTo(String str){
            if(arr.length!=str.length())
                    return -1;
            for(int i=0;i<arr.length;i++)
                    if(arr[i]!=str.charAt(i))
                            return arr[i]-str.charAt(i);
            return 0;
    }
}
class UserCompareTo{
    public static void main(String[] args) {
            UserString str = new UserString("AKSHAY");
            System.out.println(str.compareTo("AKSHAY"));
    }
}
```

**Output:-**

```
0
```

### 11. public boolean startsWith(String str):-

- This method checks if this string starts with specified prefix characters sequence.
- E.g.

```
final class UserString{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str="";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public boolean startsWith(String str){
            if(str.length()>arr.length)
                    return false;
            for(int i=0;i<str.length();i++){
                    if(str.charAt(i)!=arr[i])
                            return false;
            }
            return true;
    }
}
class UserStartsWith{
    public static void main(String[] args) {
            UserString str = new UserString("Akshay");
            System.out.println(str.startsWith("Ak"));
    }
}
```

**Output:-**

```
true
```

### 12. public boolean endsWith(String str):-

- This method checks if this string ends with specified suffix characters sequence.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str="";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public boolean endsWith(String str){
            if(str.length()>arr.length)
                    return false;
            for(int i=arr.length-str.length(),j=0;i<arr.length;i++,j++){
                    if(str.charAt(j)!=arr[i])
                            return false;
            }
            return true;
    }
}
class UserEndsWith
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.endsWith("ay"));
            System.out.println(str.endsWith("a"));
    }
}
```

**Output:-**

```
true
false
```

### 13. public int indexOf(int asciiValue):-

- This method returns the index within the String of the first occurrence of the specified character.
- If the specified character is not present it returns -1.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i]=str.charAt(i);
    }
    @Override
    public String toString(){
            String str="";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public int indexOf(int asciiValue){
            for(int i=0;i<arr.length;i++)
                    if(arr[i]==asciiValue)
                            return i;
            return -1;
    }
}
class UserIndexOf
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.indexOf('y'));
    }
}
```

**Output:-**

5

### 14. public int indexOf(int asciiValue, int startIndex):-

- This method returns the index within the string of the first occurrence of the specified character starting from the specified index (startValue).
- If the specified character is not present after the startValue index, it returns -1.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public int indexOf(int asciiValue, int startIndex){
            if (startIndex < 0 || startIndex >= arr.length)
                    return -1;
    for (int i = startIndex; i < arr.length; i++)
            if (arr[i] == (char) asciiValue)
                    return i;
    return -1;
    }
}
class UserIndexOf2
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            System.out.println(str.indexOf('a',2));
    }
}
```

**Output:-**

4

### 15. public int lastIndexOf(int asciiValue):-

- This method returns the index within the string of the last occurrence of specified character.
- If the specified character is not present it returns -1.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public int lastIndexOf(int asciiValue){
            for(int i=arr.length-1;i>=0;i--)
                    if(arr[i]==asciiValue)
                            return i;
            return -1;
    }
}
class UserLastIndexOf
{
    public static void main(String[] args)
    {
            UserString str = new UserString("AkshayAkshay");
            System.out.println(str.lastIndexOf('k'));
    }
}
```

**Output:-**

7

### 16. public int lastIndexOf(int asciiValue, int startIndex):-

- This method returns the index within the string of the last occurrence of the specified character, searching backward starting from the specified index (startIndex).
- If the specified character is not present before or at the startIndex, it returns -1.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public int lastIndexOf(int asciiValue, int startIndex) {
            if (startIndex < 0 || startIndex >= arr.length)
                            return -1;
            for (int i = startIndex; i >= 0; i--)
                            if (arr[i] == asciiValue)
                                    return i;
            return -1;
         }

}
class UserLastIndexOf2
{
    public static void main(String[] args)
    {
            UserString str = new UserString("AkshayAkshay");
            System.out.println(str.lastIndexOf('k',6));
    }
}
```

**Output:-**

1

### 17. public String toUpperCase():-

- This method returns all the lowercase characters into uppercase characters by ignoring the rest of the characters and returns a new String object.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString toUpperCase(){
            String str="";
            for(char ch: arr){
                    if(ch>=97&&ch<=122)
                            str+=(char)(ch-32);
                    else
                            str+=ch;
            }
            return new UserString(str);
    }
}
class UserToUpperCase
{
    public static void main(String[] args)
    {
            UserString str = new UserString("akshay");
            System.out.println(str.toUpperCase());
    }
}
```

**Output:-**

```
AKSHAY
```

### 18. public String toLowerCase():-

- This method converts all the uppercase characters into lowercase characters by ignoring the rest of the characters and returns a new String object.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString toLowerCase(){
            String str="";
            for(char ch: arr){
                    if(ch>=65&&ch<=90)
                            str+=(char)(ch+32);
                    else
                            str+=ch;
            }
            return new UserString(str);
    }
}
class UserToLowerCase
{
    public static void main(String[] args)
    {
            UserString str = new UserString("AKSHAY");
            System.out.println(str.toLowerCase());
    }
}
```

**Output:-**

```
akshay
```

### 19. public String trim():-

- This method returns a string value without prefixed and suffixed white spaces (blank) around.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString trim(){
            int start = 0;
            int end = arr.length-1;
            while(start<=end&&arr[start]==' ')
                    start++;
            while(end>=start&&arr[end]==' ')
                    end--;
            return substring(start,end+1);
    }
    public UserString substring(int start, int end){
            if(start>end)
                    throw new IndexOutOfBoundsException("Start is greater than end index");
            String newStr="";
            for(int i=start;i<end;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }
}
class UserTrim
{
    public static void main(String[] args)
    {
            UserString str = new UserString("    Akshay  Mali    ");
            System.out.println(str.trim());
    }
}
```

**Output:-**

```
Akshay  Mali
```

### 20. public String concat(String str):-

- This method concats the specified String to the end of this String.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString concat(String str){
            String str1 = "";
            for(char ch : arr)
                    str1+=ch;
            return new UserString(str1+str);
    }
}
class UserConcat
{
    public static void main(String[] args)
    {
            UserString str1 = new UserString("Akshay");
            System.out.println(str1.concat("Mali"));
    }
}
```

**Output:-**

```
AkshayMali
```

### 21. public String substring(int start):-

- This method returns a string i.e. a substring of the string begins with the character specified at startIndex and extends till the end of the String.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString substring(int start){
            if(start<0||start>=arr.length)
                    throw new StringIndexOutOfBoundsException("Start is greater than end index");
            String newStr="";
            for(int i=start;i<arr.length;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }
}
class UserSubstring
{
    public static void main(String[] args)
    {
            UserString str = new UserString("ABCDakshay");
            str = str.substring(4);
            System.out.println(str);
    }
}
```

**Output:-**

```
akshay
```

### 22. public String substring(int start, int end):-

- This method returns a substring of the string that begins at the specified start index and extends up to, but does not include, the end index.
- The start index must be non-negative, and the end index must be less than or equal to the length of the string. Additionally, start must be less than or equal to end.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString substring(int start,int end){
            if(start>end)
                    throw new IndexOutOfBoundsException("Start is greater than end index");
            String newStr="";
            for(int i=start;i<end;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }
}
class UserSubstring2
{
    public static void main(String[] args)
    {
            UserString str = new UserString("AkshayMali");
            str = str.substring(6,10);
            System.out.println(str);
    }
}
```

**Output:-**

```
Mali
```

### 23. public String[] split(String regex):-

- Split method splits this string from a specified regular expression into a String type array.
- E.g.

```
import java.util.*;
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public UserString[] split(String regex){
            char ch=regex.charAt(0);
            ArrayList <String> list = new ArrayList<>();
            String a = "";
            for(int i=0;i<arr.length;i++){
                    char ch1=arr[i];
                    if(ch1!=ch){
                            a+=ch1;
                    }
                    else{
                            list.add(a);
                            a="";
                    }
            }
            if(a!="")
                    list.add(a);
            UserString[] arr = new UserString[list.size()];
            int index=0;
            for(String ele:list)
                    arr[index++] = new UserString(ele);
            return arr;
    }
}
class UserSplit
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Hi how are you");
            UserString[] arr = str.split(" ");
            System.out.println(Arrays.toString(arr));
    }
}
```

**Output:-**

```
[Hi, how, are, you]
```

### 24. public String replace(char oldChar, char newChar):-

- This method returns new String by replacing all occurrences of old characters in this String with new character.
- E.g.

```
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public String replace(char ch1,char ch2)
    {
            String str="";
            for(char ch: arr)
                    str+=ch;
            for(int i=0;i<str.length();i++)
                    if(str.charAt(i)==ch1)
                            str = str.substring(0,i)+ch2+str.substring(i+1,str.length());
            return str;
    }
    public UserString substring(int start,int end){
            if(start>end)
                    throw new IndexOutOfBoundsException("Start is greater than end index");
            String newStr="";
            for(int i=start;i<end;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }
    public char charAt(int indx){
            if(indx<0||indx>=arr.length)
                    throw new StringIndexOutOfBoundsException("Wrong Index");
            return arr[indx];
    }
}
class UserReplace{
    public static void main(String[] args) {
            UserString str1 = new UserString("Bottle");
            UserString str2 = new UserString("i");
            System.out.println(str1.replace('t','i'));
    }
}
```

**Output:-**

```
Boiile
```

### 25. public char[] toCharArray():-

- This method returns a new character array that contains character elements from the specified String.
- E.g.

```
import java.util.Arrays;
final class UserString
{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch: arr)
                    str+=ch;
            return str;
    }
    public char[] toCharArray(){
            return arr;
    }
}
class UserToCharArray
{
    public static void main(String[] args)
    {
            UserString str = new UserString("Akshay");
            char[] arr = str.toCharArray();
            System.out.println(Arrays.toString(arr));
    }
}
```

**Output:-**

```
[A, k, s, h, a, y]
```

### 26. public String replaceFirst(String regex, String replacement):-

- This method returns a new string by replacing the first occurrence of the specified substring (defined by regex) in this string with the provided replacement string.
- E.g.

```
final class UserString{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch : arr)
                    str+=ch;
            return str;
    }
    public UserString replaceFirst(String regex, String replacement){
            String str = this.toString();
            if(!(str.contains(regex)))
                    return new UserString(str);
            int index = str.indexOf(regex);
            String before = str.substring(0,index);
            String after = str.substring(index+regex.length());
            return new UserString(before+replacement+after);
    }
    public UserString substring(int start){
            if(start<0||start>=arr.length)
                    throw new StringIndexOutOfBoundsException("Start is greater than
                                                                    end index");
            String newStr="";
            for(int i=start;i<arr.length;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }

    public UserString substring(int start, int end){
            if(start>end)
                    throw new IndexOutOfBoundsException("Start is greater than end index");
            String newStr="";
            for(int i=start;i<end;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }
}
class UserReplaceFirst{
    public static void main(String[] args) {
            UserString str = new UserString("Hello how are you");
            System.out.println(str.replaceFirst("how","hey"));
    }
}
```

**Output:-**

```
Hello hey are you
```

### 27. public String replaceAll(String regex, String replacement):-

- This method returns a new string by replacing all occurrences of the specified substring or pattern (defined by regex) in this string with the provided replacement string.
- E.g.

```
final class UserString{
    char[] arr;
    UserString(){
            arr = new char[0];
    }
    UserString(String str){
            arr = new char[str.length()];
            for(int i=0;i<arr.length;i++)
                    arr[i] = str.charAt(i);
    }
    @Override
    public String toString(){
            String str = "";
            for(char ch : arr)
                    str+=ch;
            return str;
    }
    public UserString replaceAll(String regex, String replacement) {
            String str = this.toString();
            String newStr = "";
            int start = 0;
            while (str.contains(regex)) {
                            int index = str.indexOf(regex);
                            newStr += str.substring(0, index) + replacement;
                    str = str.substring(index + regex.length());
            }
            newStr += str;
            return new UserString(newStr);
    }

    public UserString substring(int start, int end){
            if(start>end)
                    throw new IndexOutOfBoundsException("Start is greater than end index");
            String newStr="";
            for(int i=start;i<end;i++)
                    newStr+=arr[i];
            return new UserString(newStr);
    }
}
class UserReplaceAll{
    public static void main(String[] args) {
            UserString str1 = new UserString("Hi how how you");
            System.out.println(str1.replaceAll("how","hey"));
    }
}
```

**Output:-**

```
Hi hey hey you
```

**Assignment :-**

**Q. 1. Write a java program to reverse a String if the String contains special characters their position should not be changed.**

**Ans.**

```
class SpecialStringReverse
{
	public static void main(String[] args)
	{
		String str = "ab@cd#e";
		System.out.println(str);
		String str1 = str.replaceAll("[^A-Za-z]","");
		str1 = (new StringBuffer(str1).reverse()).toString();
		for(int i=0;i<str.length();i++)
		{
			char ch = str.charAt(i);
			if(!((ch>=65&&ch<=90)||(ch>=97&&ch<=122)))
				str1 = str1.substring(0,i)+ch+str1.substring(i);
		}
		System.out.println(str1);
	}
}
```

**Output:-**

```
ab@cd#e
ed@cb#a
```

**Q. 2. Write a java program to check the String is anagram or not.**

**Ans.**

```
import java.util.*;
class AnagramString
{
	public static void main(String[] args)
	{
		String str1 = "pot";
		String str2 = "top";
		char[] arr1 = str1.toCharArray();
		char[] arr2 = str2.toCharArray();
		boolean flag = anagramString(arr1,arr2);
		System.out.println(flag?str1+" its an anagram":str1+" its not an anagram");
	}
	public static boolean anagramString(char[] ch1,char[] ch2)
	{
		if(ch1.length!=ch2.length)
			return false;
		Arrays.sort(ch1);
		Arrays.sort(ch2);
		int indx=0;
		for (char i : ch1) {
			if(i!=ch2[indx++])
				return false;
		}
		return true;
	}
}
```

**Output:-**

```
pot its an anagram
```

**Q. 3. Write a java program to check the String is panagram or not.**
**Ans.**

```
class PanagramString
{
	public static void main(String[] args)
	{
		String str = "abcdefghijklmnopqrstuvwyz";	//x is missing
		boolean flag = true;
		for(char i='a';i<='z';i++)
		{
			if(!(str.contains(i+"")))
			{
				flag = false;
				break;
			}
		}
		System.out.println(flag?"its an panagram":"its not an panagram");
	}
}
```

**Output:-**

```
its not an panagram
```

**Q. 4. Find an highest frequency word from a String sentence.**
**Ans.**

```
class HighestFrequencyOfWordFromString{
	public static void main(String[] args) {
		String str = "hello java hello java hello java hello";
		System.out.println(str);
		String[] arr = str.split(" ");
		boolean[] barr = new boolean[arr.length];
		String highFrequencyWord = "";
		int max = 0;
		for(int i=0;i<arr.length;i++){
			String iele = arr[i];
			int cnt=0;
			for(int j=0;j<arr.length;j++){
				String jele = arr[j];
				if(iele.equals(jele) && barr[j]==false){
					cnt++;
					barr[j]=true;
				}
			}
			if(cnt>max){
				max = cnt;
				highFrequencyWord = iele;
			}
		}
		System.out.println(highFrequencyWord+" : "+max);
	}
}
```

**Output:-**

```
hello java hello java hello java hello
hello : 4
```

**Q. 5. Write a java program to find highest element from a String.**
**Ans.**

```
class HighestFrequencyOfElement{
	public static void main(String[] args) {
		String str = "Mississippii";
		boolean[] arr = new boolean[str.length()];
		char highFrequencyElement ='0';
		int max =0;
		for(int i=0;i<str.length();i++){
			char iele = str.charAt(i);
			int cnt = 0;
			for(int j=0;j<str.length();j++){
				char jele = str.charAt(j);
				if(iele==jele && arr[j]==false){
					cnt++;
					arr[j] = true;
				}
			}
			if(cnt>max){
				max = cnt;
				highFrequencyElement = iele;
			}
		}
		System.out.println(highFrequencyElement+" : "+max);
	}
}
```

**Output:-**

```
i : 5
```

**Q. 6. Write a java program to find distinct element from the String. (Hello → Helo)**
**Ans.**

```
class DistinctElementsOfString{
	public static void main(String[] args) {
		String str = "Hello";
		System.out.println(str);
		boolean[] arr = new boolean[str.length()];
		String newStr = "";
		for(int i=0;i<str.length();i++){
			char iele = str.charAt(i);
			int cnt = 0;
			for(int j=0;j<str.length();j++){
				char jele = str.charAt(j);
				if(iele==jele && arr[j]==false){
					cnt++;
					arr[j] = true;
				}
			}
			if(cnt>0)
				newStr+=str.charAt(i);
		}
		System.out.println(newStr);
	}
}
```

**Output:-**

```
Hello
Helo
```

**Q. 7. Write a java program to find duplicate element from the String. (Hello → l)**
**Ans.**

```
class DuplicateElementFromString{
	public static void main(String[] args) {
		String str = "Hello";
		System.out.println(str);
		boolean[] arr = new boolean[str.length()];
		String newStr = "";
		for(int i=0;i<str.length();i++){
			char iele = str.charAt(i);
			int cnt = 0;
			for(int j=0;j<str.length();j++){
				char jele = str.charAt(j);
				if(iele==jele && arr[j]==false){
					cnt++;
					arr[j] = true;
				}
			}
			if(cnt>1)
				newStr+=str.charAt(i);
		}
		System.out.println(newStr);
	}
}
```

**Output:-**

```
Hello
l
```

**Q. 8. Write a java program to find unique element from String. (Hello → Heo)**
**Ans.**

```
class UniqueElementFromString{
	public static void main(String[] args) {
		String str = "Hello";
		System.out.println(str);
		boolean[] arr = new boolean[str.length()];
		String newStr = "";
		for(int i=0;i<str.length();i++){
			char iele = str.charAt(i);
			int cnt = 0;
			for(int j=0;j<str.length();j++){
				char jele = str.charAt(j);
				if(iele==jele && arr[j]==false){
					cnt++;
					arr[j] = true;
				}
			}
			if(cnt==1)
				newStr+=str.charAt(i);
		}
		System.out.println(newStr);
	}
}
```

**Output:-**

```
Hello
Heo
```

**Q. 9. Write a java program to find distinct word from a String sentence.**
**Ans.**

```
import java.util.*;
class DistinctStringFromStringSentence
{
	public static void main(String[] args)
	{
		String str = "Hello java hello java hello bye";
		String[] arr = str.split(" ");
		System.out.println(Arrays.toString(arr));
		boolean[] barr = new boolean[arr.length];
		String newStr = "";
		for(int i=0;i<arr.length;i++)
		{
			String iele = arr[i];
			int cnt = 0;
			for(int j=0;j<arr.length;j++)
			{
				String jele = arr[j];
				if(iele.equals(jele) && barr[j]==false)
				{
					cnt++;
					barr[j] = true;
				}
			}
			if(cnt>0)
				newStr += arr[i] + " ";
		}
		System.out.println(newStr);
	}
}
```

**Output:-**

```
[Hello, java, hello, java, hello, bye]
Hello java hello bye
```

**Q. 10. Write a java program to find duplicate element from a String sentence.**
**Ans.**

```
import java.util.*;
class DuplicateStringFromStringSentence
{
	public static void main(String[] args)
	{
		String str = "Hello java hello java hello";
		String[] arr = str.split(" ");
		System.out.println(Arrays.toString(arr));
		boolean[] barr = new boolean[arr.length];
		String newStr = "";
		for(int i=0;i<arr.length;i++)
		{
			String iele = arr[i];
			int cnt = 0;
			for(int j=0;j<arr.length;j++)
			{
				String jele = arr[j];
				if(iele.equals(jele) && barr[j]==false)
				{
					cnt++;
					barr[j] = true;
				}
			}
```

```
                    if(cnt>1)
                        newStr += arr[i] + " ";
                }
                System.out.println(newStr);
        }
}
```

**Output:-**

```
[Hello, java, hello, java, hello]
java hello
```

**Q. 11.Write a java program to find unique words from String sentence.**

**Ans.**

```
import java.util.*;
class UniqueStringFromStringSentence
{
        public static void main(String[] args)
        {
                String str = "Hello java hello java hello";
                String[] arr = str.split(" ");
                System.out.println(Arrays.toString(arr));
                boolean[] barr = new boolean[arr.length];
                String newStr = "";
                for(int i=0;i<arr.length;i++)
                {
                        String iele = arr[i];
                        int cnt = 0;
                        for(int j=0;j<arr.length;j++)
                        {
                                String jele = arr[j];
                                if(iele.equals(jele) && barr[j]==false)
                                {
                                        cnt++;
                                        barr[j] = true;
                                }
                        }
                        if(cnt==1)
                                newStr += arr[i] + " ";
                }
                System.out.println(newStr);
        }
}
```

**Output:-**

```
[Hello, java, hello, java, hello]
Hello
```

# ★ StringBuffer class:-

## ★ Introduction:-

- StringBuffer is a predefined built in class.
- It is derived in java.lang package.
- StringBuffers are similar to String which contains character sequence enclosed within double quotes.
- It is mutable in nature (It means that it can be modified).
- StringBuffer is synchronized and it is thread safe.
- If our frequent operation is to insert, remove / replace we use StringBuffer.
- E.g.

```
class Example1{
    public static void main(String[] args) {
            StringBuffer sb = new StringBuffer();
            System.out.println(sb.length());        //0
            System.out.println(sb.capacity());      //16
            sb.append("1234567890123456");
            System.out.println(sb.length());        //16
            System.out.println(sb.capacity());      //16
            sb.append("1");                         //(16+1)*2 = 34
            System.out.println(sb.length());        //17
            System.out.println(sb.capacity());      //34
    }
}
```

**Output:-**

```
0
16
16
16
17
34
```

- E.g.

```
class Example2{
        public static void main(String[] args) {
                StringBuffer sb = new StringBuffer("1234");
                System.out.println(sb);                     //1234
                System.out.println(sb.length());            //4
                System.out.println(sb.capacity());  //20
                sb.append("5678901234567890");
                System.out.println(sb.length());            //20
                System.out.println(sb.capacity());  //20
                sb.append("21");                    //(20+1)*2 = 42
                System.out.println(sb.length());            //22
                System.out.println(sb.capacity());  //42
        }
}
```

**Output:-**

```
1234
4
20
20
20
22
42
```

- If we are creating a StringBuffer object using a no argument constructor, the initial capacity will be 16 character sequence.
- Capacity is nothing but number of characters StringBuffer can hold.
- Once the buffer capacity is full and if we try to append or insert new characters in it, internally the buffer object grows.
- Same Buffer object increases its capacity (no new object is created).
- Formula to calculate the capacity of buffer

  **newCapacity = (initialCapacity + 1) * 2**
  newCapacity = ( 16 + 1 ) * 2
  newCapacity = 34

## ★ Constructors of StringBuffer:-

1. public StringBuffer()
2. public StringBuffer(int capacity)
3. public StringBuffer(String str)

### 1. public StringBuffer():-

- It constructs a StringBuffer with no characters in it and the initial capacity is of 16 character sequence.
- E.g.

```
class Example3
{
    public static void main(String[] args)
    {
        StringBuffer sb = new StringBuffer();
        System.out.println(sb.length());            //0
        System.out.println(sb.capacity());    //16
        sb.append("12345678901234567");
        System.out.println(sb.capacity());    //(16 + 1) * 2 = 34
        System.out.println(sb.length());            //17
    }
}
```

**Output:-**

```
0
16
34
17
```

### 2. public StringBuffer(int capacity):-

- It constructs a StringBuffer with no characters in it, with a specified initial capacity.
- E.g.

```
class Example4
{
	public static void main(String[] args)
	{
		StringBuffer sb = new StringBuffer(12);
		System.out.println(sb.capacity());	// 12
		sb.append("1234567890123");
		System.out.println(sb.capacity());	// ( 12 + 1) * 2 = 26
	}
}
```

**Output:-**

```
12
26
```

- StringBuffer sb = new StringBuffer(-2);
  RuntimeException NegativearraySizeException
- The internal implementation of a StringBuffer is char[].
- Whenever we create a StringBuffer object internally it creates a char[] and stores the characters in it in form of elements.
- So whenever we specifying capacity of a buffer indirectly we are specifying the size of an array.
- We cannot create a Buffer of negative capacity as we cannot create an array negative size.

### 3. public StringBuffer(String str):-

- It constructrs a StringBuffer with initial String initialied in it.
- The initial capacity of Buffer is 16 + the length of character sequence specified.
- E.g. 1

```
class Example5
{
	public static void main(String[] args)
	{
		StringBuffer sb = new StringBuffer("hello");
		System.out.println(sb.capacity());	//16 + 4 = 21
	}
}
```

**Output:-**

```
21
```

- E.g. 2

```
class Example6
{
	public static void main(String[] args)
	{
		String str = new String("Akshay");
		StringBuffer sb = new StringBuffer(str);
		System.out.println(sb.capacity());	//16 + 5 = 22
	}
}
```

**Output:-**

```
22
```

## ★ Methods of StringBuffer:-

1. public synchronized int length();
2. public synchronized int capacity();
3. public synchronized void ensureCapacity(int);
4. public synchronized void trimToSize();
5. public synchronized java.lang.StringBuffer append(char[], int, int);
6. public synchronized java.lang.StringBuffer append(boolean);
7. public synchronized java.lang.StringBuffer append(char);
8. public synchronized java.lang.StringBuffer append(int);
9. public synchronized java.lang.StringBuffer append(long);
10. public synchronized java.lang.StringBuffer append(float);
11. public synchronized java.lang.StringBuffer append(double);
12. public synchronized java.lang.StringBuffer append(java.lang.Object);
13. public synchronized java.lang.StringBuffer append(java.lang.String);
14. public synchronized java.lang.StringBuffer append(java.lang.StringBuffer);
15. synchronized java.lang.StringBuffer append(java.lang.AbstractStringBuilder);
16. public synchronized java.lang.StringBuffer insert(int, char);
17. public java.lang.StringBuffer insert(int, int);
18. public java.lang.StringBuffer insert(int, long);
19. public java.lang.StringBuffer insert(int, float);
20. public java.lang.StringBuffer insert(int, double);
21. public synchronized java.lang.StringBuffer insert(int, java.lang.String);
22. public synchronized java.lang.StringBuffer insert(int, java.lang.Object);
23. public synchronized char charAt(int);
24. public synchronized java.lang.StringBuffer delete(int, int);
25. public synchronized java.lang.StringBuffer deleteCharAt(int);
26. public synchronized java.lang.StringBuffer reverse();
27. public synchronized void setCharAt(int, char);

### 1. public synchronized int length();

- This method returns the count of characters present in the StringBuffer.
- E.g.

```
class Example7
{
        public static void main(String[] args)
        {
                StringBuffer sb = new StringBuffer("Akshay");
                for(int i=0;i<sb.length();i++)
                {
                        System.out.println(sb.charAt(i));
                }
        }
}
```

**Output:-**

```
A
k
s
h
a
y
```

**2. public synchronized int capacity():-**

- This method returns the capacity of StringBuffer object.
- The capacity is the amount of storage we can use in StringBuffer that is number of character sequence.
- E.g.

```
class CapacityExample{
    public static void main(String[] args) {
            StringBuffer sb = new StringBuffer("HELLO");
            System.out.println(sb.capacity());
            sb.append("12345678901234567890");
            System.out.println(sb.capacity());
            StringBuffer sb1 = new StringBuffer(10);
            System.out.println(sb1.capacity());
            sb1.append("12345678901");
            sb1.trimToSize();
            System.out.println(sb1.capacity());
    }
}
```

**Output:-**

```
21
44
10
11
```

**3. public synchronized void trimToSize():-**

- This method attempts to reduce storage use for character sequence.
- If the buffer is larger than necessary to hold its current sequence of character then it may be resize to become more space efficient.
- In simple words, we invoke trimToSize() method the buffer capacity will be equivalent to its length (count of characters).
- E.g.

```
class TrimToSizeExample{
    public static void main(String[] args) {
            StringBuffer sb = new StringBuffer(90);
            for(int i=1,j=1;i<=40;i++,j++){
                    sb.append(j);
                    if(j==9)
                            j=1;
            }
            System.out.println(sb);
            System.out.println("Capacity : "+sb.capacity());
            sb.trimToSize();
            System.out.println("After trim to size method : ");
            System.out.println("Length : "+sb.length());
            System.out.println("Capacity : "+sb.capacity());
    }
}
```

**Output:-**

```
1234567892345678923456789234567892345678
Capacity : 90
After trim to size method :
Length : 40
Capacity : 40
```

### 4. public synchronized void ensureCapcity(int newCapacity):-

- It ensures that capacity is atleast equals to the specified minimum capacity.
- If the current capacity is less than the argument then, it will increment the size of buffer object equivalent to specified capacity.
- E.g

```
class EnsureCapacityExample{
    public static void main(String[] args) {
            StringBuffer sb = new StringBuffer();
            sb.ensureCapacity(300);
            for(int i=1,j=65;i<=200;i++){
                    sb.append((char)j);
                    if(j==90)
                            j=65;
            }
            System.out.println(sb.capacity());
            StringBuffer sb1 = new StringBuffer(100);
            sb1.ensureCapacity(50);
            for(int i=0,j=65;i<=50;i++,j++){
                    sb.append((char)j);
                    if(j==90)
                            j=65;
            }
            System.out.println(sb1.capacity());
    }
}
```

**Output:-**
300
100

### 5. public synchronized StringBuffer append( ):-

int, double, float, long, char, boolean,
Object, String, StringBuffer, StringBuilder

- This method is used to append [ concatenation ] to specified argument data at the end of character sequence.
- Return type of this method is StringBuffer.
- These method is an example of method overloading.
- E.g.

```
class AppendExample
{
    public static void main(String[] args)
    {
            StringBuffer sb = new StringBuffer();
            sb.append(10);
            sb.append(1123L);
            sb.append(1.1f);
            sb.append("e");
            System.out.println(sb);
    }
}
```

**Output:-**
1011231.1e

### 6. public synchronized StringBuffer reverse():-

- This method reverse the character sequence and replace it with original character and returns a StringBuffer object.
- E.g.

```
class ReverseExample
{
    public static void main(String[] args)
    {
            String str = "madam";
            StringBuffer sb = new StringBuffer(str);
            sb.reverse();
            if(str.contentEquals(sb))
                    System.out.println(str+" is a palindrome.");
            else
                    System.out.println(str+" is not a palindrome.");
    }
}
```

**Output:-**

madam is a palindrome.

### 7. public synchronized StringBuffer deleteCharAt(int index):-

- It remove the character form the specified position [index] from these sequence.
- The return type of these method is StringBuffer.
- E.g.

```
class DeleteCharAtExample
{
    public static void main(String[] args)
    {
            StringBuffer sb = new StringBuffer("elephant");
            for(int i=0;i<sb.length();i++)
            {
                    char ch = sb.charAt(i);
                    if(ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u')
                            sb.deleteCharAt(i);
            }
            System.out.println(sb);
    }
}
```

**Output:-**

lphnt

### 8. public synchronized StringBuffer insert( ):-

(int , double) (int , float) (int , int) (int , long)
(int , char) (int , String) (int , Object) (int , boolean)

- This method is used to insert data that is specified given index.
- Return type of this method is StringBuffer.
- Its an example of method overloading.
- E.g.

```
class InsertExample
{
    public static void main(String[] args)
    {
            StringBuffer sb = new StringBuffer();
            sb.append(" JAVA");
            sb.insert(0,"HELLO");
            int index = sb.indexOf(" ");
            sb.setCharAt(index,'-');
            sb.insert(0,"");
            sb.append("");
            System.out.println(sb);
    }
}
```

**Output:-**
"HELLO-JAVA"

### 9. public synchronized void setCharAt(int, char):-

- This method replaces the character specified index with the new character.
- E.g.

```
class SetCharAtExample
{
    public static void main(String[] args)
    {
            StringBuffer sb = new StringBuffer("1234567");
            System.out.println(sb+" : ");
            for(int i=0;i<sb.length();i++)
            {
                    int ch = sb.charAt(i)-48;
                    if(ch==0)
                            sb.setCharAt(i,'N');
                    if(ch%2!=0)
                            sb.setCharAt(i,'O');
                    else
                            sb.setCharAt(i,'E');
            }
            System.out.println(sb);
    }
}
```

**Output:-**
1234567 :
OEOEOEO

★ **WAJP to convert number to String.**

```
class NumberToBinary
{
    public static void main(String[] args)
    {
            int num = 24;
            String bin = "";
            for(int i=num;num!=0;num/=2)
            {
                    bin = (num%2)+bin;
            }
            System.out.println(bin);
    }
}
```

**Output:-**
11000

★ **Difference between StringBuffer and StringBuilder**

| StringBuffer | StringBuilder |
|---|---|
| i. Method in StringBuffer are synchronized. | i. Methods in StringBuilder are not synchronized. |
| ii. At a time only one thread is allowed to operate on StringBuffer object. | ii. At a time multiple threads are allowed to operate on StringBuilder object. |
| iii. Waiting time of thread is more. | iii. Waiting time of thread is less. |
| iv. StringBuffers are slower. | iv. StringBuilders are faster. |
| v. StringBuffers was introduced in JDK 1.0 version. | v. StringBuilders are introduced in JDK 1.5 version. |
| vi. StringBuffers have capacity. | vi. StringBuilders doesn't have capacity. |

★ **Difference between String and StringBuffer**

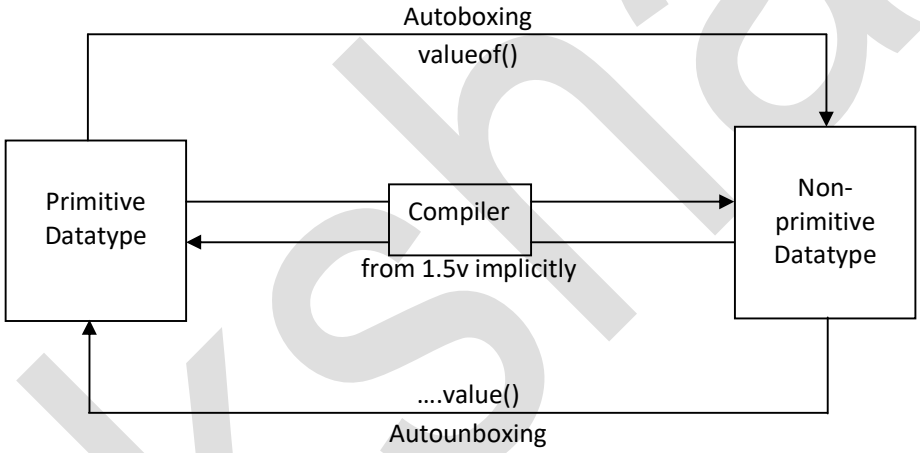
| String | StringBuffer |
|---|---|
| i. Strings are immutable in nature. | i. StringBuffers are mutable in nature. |
| ii. String cannot be modified. | ii. StringBuffer can be modified. |
| iii. For every operation it creates a new object. | iii. For every operation same object is modified and return. |
| iv. Everytime it creates a new object and consumes more memory. | iv. Returns the same object and consumes less memory. |
| v. Performance wise String is slow. | v. StringBuffers are faster. |
| vi. Strings are highly secured. | vi. StringBuffers are not secured. |

# ★ Wrapper class:-

- Java's wrapper classes wraps the primitive datatype, that's why they are known as Wrapper class.
- All the wrapper classes are final because no other class extend it.
- All the wrapper classes in java are immutable.
- Wrapper claases are fundamental in java / they help java program to be completely object oriented.
- Wrapper claases provides the mechianism to convert primitive datatype into object (non-primitive datatype) and object into primitive datatype.
- Wrapper classes are present in java.lang package.
- From version 1.5 this coversion is done by the compiler implicitly.

### ★ Why we need wrapper class in java?

i. Instead of using primitive datatype we can use its corresponding non-primitive datatype in that sense wrapper classes helps the java program to be 100% object oriented.
ii. To bind values of different primitive datatype into object (which helps to perform complex operation) while working with collection.
iii. To provide different utility methods that can be used with primitive datatypes.
iv. Since, primitive datatype can't be given null value but wrapper classes can be used to assign null value to any primitive datatype.
v. They help with synchronization because multipthreading in java requires object.
vi. Working with collection framework become easy with wrapper class.




| Primitive Datatype | Wrapper classes Non primitive Datatype | Constructors |
|---|---|---|
| byte | Byte | byte / String |
| short | Short | Short / String |
| int | Integer | int / String |
| long | Long | long / String |
| float | Float | float double String |
| double | Double | double String |
| char | Character | char |
| boolean | Boolean | boolean String |

## ⋆ Autoboxing:-

- Implicit conversion of primitive datatype into its corresponding wrapper class is known as autoboxing.
- This conversion is done by compiler internally.
- E.g. 1)

```
class ExampleAutoboxing
{
    public static void main(String[] args)
    {
            int a = 10;
            System.out.println(a);
            Integer integer = a;
            System.out.println(integer);
            Integer integer2 = Integer.valueOf(a);
            System.out.println(integer2);
            String b = "10";
            Byte byte1 = Byte.valueOf(b);
            System.out.println(byte1);
    }
}
```

**Output:-**

```
10
10
10
10
```

E.g. 2)

```
class ExampleAutoboxing2
{
    public static void main(String[] args)
    {
            Integer integer = 127;
            Integer integer1 = 127;
            System.out.println(integer==integer1);
            Integer integer2 = 130;
            Integer integer3 = 130;
            System.out.println(integer2==integer3);
            Integer integer4 = 130;
            Integer integer5 = 130;
            System.out.println(integer4.equals(integer3));
    }
}
```

**Output:-**

```
true
false
true
```

**Note:-**

- If we try to store a data within a range of -128 to 127 inside any of wrapper class the data will store inside SCP it means same value will have same reference.
- And if we try to compare them it is going to compare reference.
- The value exceeding the cache memory (-128 to 127) will be stored in form of a object inside heap not in SCP and the references will be different that's why we compare them using == we will get false.
- But we can compare them using equals() which is meant for content comparison.

- E.g.

```
class ExampleAutoboxing3
{
    public static void main(String[] args)
    {
            float a = 1.1f;
            double d = 1.2;
            String str = "1.1";
            Float float1 = a;
            Float float2 = a;
            Float float7 = Float.valueOf(a);
            Float float3 = new Float(a);
            System.out.println(float3);
            Float float4 = new Float(a);
            System.out.println(float4);
            Float float5 = (float) a;
            System.out.println(float5);
            Float float6 = new Float(str);
    }
}
```

**Output:-**

```
ExampleAutoboxing3.java:11: warning: [removal] Float(float) in Float has been deprecated and marked for removal
            Float float3 = new Float(a);
                           ^
ExampleAutoboxing3.java:13: warning: [removal] Float(float) in Float has been deprecated and marked for removal
            Float float4 = new Float(a);
                           ^
ExampleAutoboxing3.java:17: warning: [removal] Float(String) in Float has been deprecated and marked for removal
            Float float6 = new Float(str);
                           ^
3 warnings
1.1
1.1
1.1
```

- E.g.

```
class ExampleAutoboxing4
{
        @SuppressWarnings("removal")
        public static void main(String[] args)
        {
                boolean b = false;
                Boolean boolean1 = b ;
                Boolean boolean2 = Boolean.valueOf(b);
                Boolean boolean3 = new Boolean(b);
                Boolean boolean4 = new Boolean("");
                Boolean boolean5 = new Boolean("false");
                Boolean boolean6 = new Boolean("TRUE");
                Boolean boolean7 = new Boolean("TrUe");
                Boolean boolean8 = new Boolean("GALAT");
                Boolean boolean9 = new Boolean("0");
                Boolean boolean10 = new Boolean("1");
                System.out.println(boolean1);
                System.out.println(boolean2);
                System.out.println(boolean3);
                System.out.println(boolean4);
```

```
                System.out.println(boolean5);
                System.out.println(boolean6);
                System.out.println(boolean7);
                System.out.println(boolean8);
                System.out.println(boolean9);
                System.out.println(boolean10);
        }
}
```

**Output:-**

```
false
false
false
false
false
true
true
false
false
false
```

**Note:-**

- Every wrapper class will be having range similar to its primitive datatype.
- E.g.
- byte :- -128 to 127
- Byte :- -128 to 127

★ **valueOf():-**

- The valueOf() method is static method present in every wrapper class.
- Used to convert primitive to non-primitive.
- valueOf() is an example of method overloading.

**Methods:-**

```
public  static  Integer  valueOf(int);
public  static  Integer  valueOf(Stirng);
public  static  Integer  valueOf(Stirng, int->radix);
```

**The range of radix is 2 to 36.**

## ★ Autounboxing:-

- The implicit conversion from non-primitive to primitive datatype is known as autoboxing.
- It is done by compiler implicitly from JDK 1.5 version.
- If we need to do this conversion explicitly we use method ….value().

**….value():-**

- This method used to convert non-primitive to primitive.
- Used in unboxing explicit conversion.
- This method is non static method present in every wrapper class.
- Syntax:-
  reference_variable.primitive_datatypevalue();
- E.g.
  a.bytevalue();
  a.intvalue();
  a.charvalue();

**E.g. 1)**

```
class ExampleAutounboxing
{
@SuppressWarnings("removal")
public static void main(String[] args)
{
        Integer integer = new Integer(10);
        int a = integer;
        int b = integer.intValue();
        int c = (int) integer;
        int d = new Integer(integer);
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
}
}
```

**Output:-**

```
10
10
10
10
```

**E.g. 2)**

```
class ExampleAutounboxing1
{
	@SuppressWarnings("removal")
	public static void main(String[] args)
	{
		Boolean boolean1 = new Boolean(false);
		boolean a = boolean1;
		boolean b = boolean1.booleanValue();
		boolean c = (boolean)boolean1;
		boolean d = new Boolean(boolean1);
		System.out.println(a);
		System.out.println(b);
		System.out.println(c);
		System.out.println(d);
	}
}
```

**Output:-**

```
false
false
false
false
```

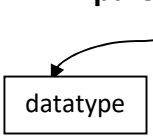
**E.g. 3)**

```
class ExampleAutounboxing2
{
	@SuppressWarnings("removal")
	public static void main(String[] args)
	{
		Character ch = 'a';
		char a = ch;
		char b = ch.charValue();
		char c = new Character(ch);
		char d = (char)ch;
		System.out.println(a);
		System.out.println(b);
		System.out.println(c);
		System.out.println(d);
	}
}
```

**Output:-**

```
a
a
a
a
```

★ **parse<u>xxxx</u>():-**




- We can use the parsexxxx() method convert a String into primitive (corresponding primitive datatype).
- The methods are static.
- E.g.

```
class ParseMethodExample
{
        public static void main(String[] args)
        {
                String a = "10";
                byte b = Byte.parseByte(a);
                boolean c = Boolean.parseBoolean(a);
                double d = Double.parseDouble(a);
                System.out.println(a);
                System.out.println(b);
                System.out.println(c);
                System.out.println(d);
        }
}
```

**Output:-**

```
10
10
false
10.0
```

★ **toString():-**

- The toString() method used to convert wrapper object or primitive datatype to String.
- The toString() method present in every wrapper class.
- E.g.

```
class ToStringMethodExample
{
    @SuppressWarnings("removal")
    public static void main(String[] args)
    {
            int a = 10;
            Integer integer = new Integer(100);
            String str = integer.toString();
            Integer integer1 = a;
            String str1 = integer1.toString();
            System.out.println(str);
            System.out.println(str1);
    }
}
```

**Output:-**

```
100
10
```

★ **Write a java program to check whether the number is palindrome or not and take input from the command line.**

```
class Palindrome
{
    public static void main(String[] args)
    {
            int num1 = Integer.parseInt(args[0]);
            int num2 = num1;
            int rev = 0;
            for( int i = num1 ; i!=0 ; i/=10 )
            {
                    int rem = i%10;
                    rev = rev*10+rem;
            }
            System.out.println(rev==num1?"Palindrome":"Not Palindrome");
    }
}
```

javac Palindrome.java
java Palindrome 777
**Output:-**
Palindrome

★ **Packages:-**

A package is a group of similar type of classes and sub packages, interface.

★ **Subpackage:-**

Package present inside another package is called as subpackage.
java.util.regex;
java.util.function;

★ **Packages in java categories as:-**

i. Built in
ii. Userdefine

**Note:-**

public and private modifiers are not allowed in package declaration.

# ★ Collection Framework:-

★ **Why do we need collection framework in java?**

- If we want to store multiple data or group of objects together we usually use Arrays but it has some limitations.
- **Disadvantages of Arrays (limitations):-**
  i. We cannot store heterogeneous data.
  ii. If we need to use Arrays, we should know the base size in advance.
  iii. Array consumes more memory but from performance point of view they are faster in execution.
  iv. Java doesn't provide much support for Arrays as they are not implemented using data structures.
  v. So we have to write complex code and logic which consumes time.
- **Conclusion:-**
  So to overcome all the above problems we have collections in java.

★ **Difference between Arrays and Collections.**

| Arrays | Collections |
|---|---|
| i. Arrays are index collection of fixed number of homogenous data elements. | i. Collections is group of individual object represents as single entity. |
| ii. Arrays are fixed in size. | ii. Collections are not fixed in size. |
| iii. To use Arrays we should know the size in adavance. | iii. To use collection we don't need to know the size in adavance. |
| iv. We can store both primitive and non primitive data in it. | iv. We can only store non primitive data in it. |
| v. Arrays are faster. | v. Collections are slower. |
| vi. Arrays consumes more memory. | vi. Collections consumes less memory. |
| vii. Codes are complex in Arrays. | vii. Codes are consize in collections. |
| viii. It does not provides much built in supports for operations. | viii. It provides built in supports for operations. |

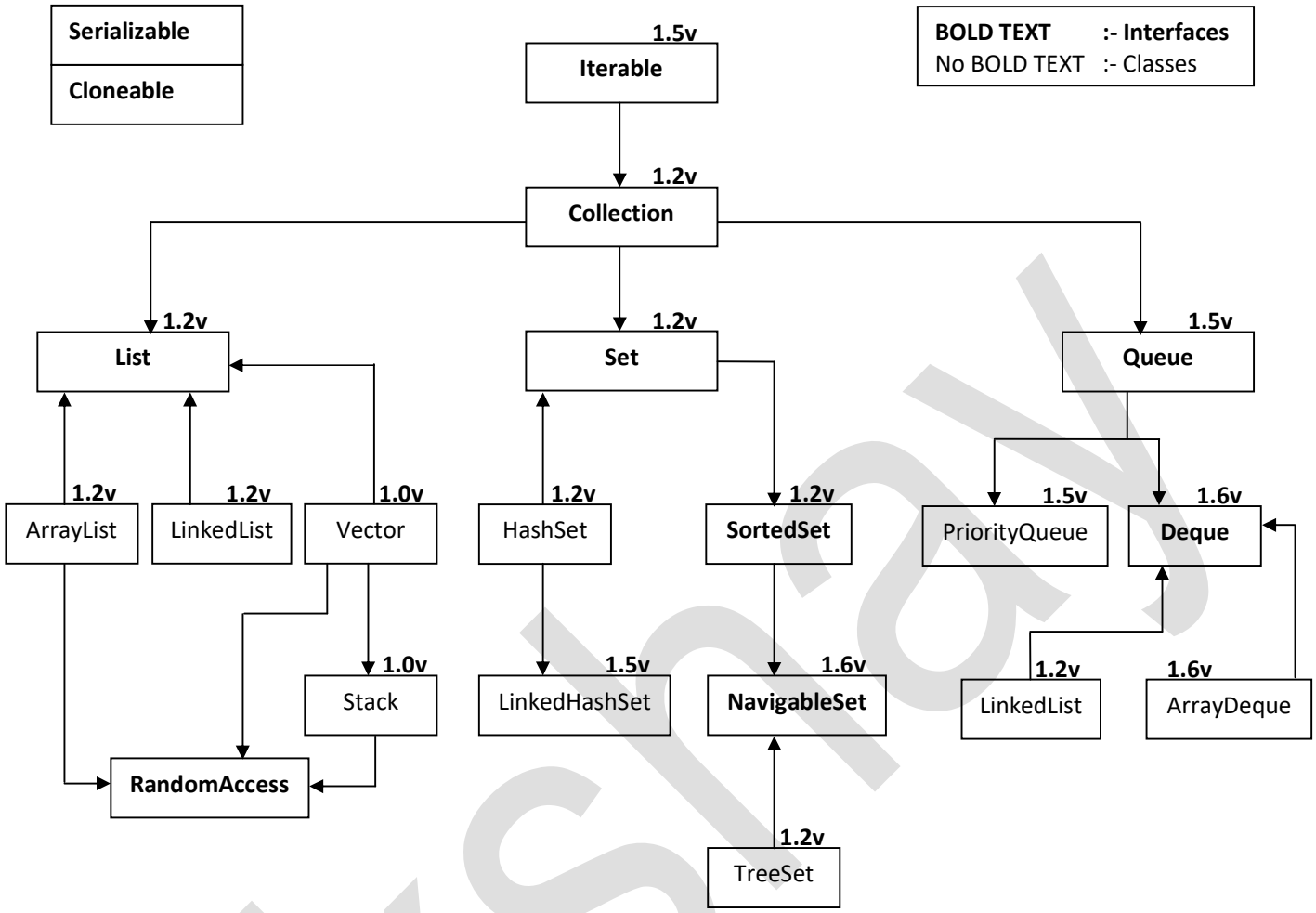
★ **Collection Framework:-**

- Collection framework represents an architecture for storing, arranging and manipulating group of objects in java.
- It has two main interfaces in it,
  i. Collection Interface
  ii. Map Interface

★ **collection :-**

- The group of individual objects representing as a single entity is known as collection.

# Collection Interface




# Classification Of Collection

**Legacy Classes:-**
1. Vector
2. Stack
3. HashTable
4. Dictionary (AC)
5. Properties

**Legacy Interface:-**
1. Enumeration

**Cursors:-**
1. Enumeration
2. Iterator
3. ListIterator

# ★ Collection Interface:-

- Collection interface is a member of java's collection framework.
- It is derived in java.util package.
- The Collection interface is used to pass collection of objects.
- It provides the standard method and behaviours for working with group of objects.
- It extends **Iterable** interface and inherits **iterator()** enabling collections to be used in advance for loop.
- The Collection interface is not directly implemented by any other class.
- However it is implemented indirectly while its sub types (sub interfaces) like List, Set, Queue.

## ★ Methods of Collection interface:-

1. public abstract int size();
2. public abstract boolean isEmpty();
3. public abstract boolean add(E ele);
4. public abstract boolean addAll(Collection c);
5. public abstract boolean remove(Object obj);
6. public abstract boolean removeAll(Collection c);
7. public abstract boolean retainAll(Collection c);
8. public abstract void clear();
9. public abstract Iterator <E> iterator();
10. public abstract T[] toArray(T[]);
11. public abstract Object[] toArray();
12. public Stream <E> stream();

### 1. public abstract boolean add(E ele):-

- This method is used to insert an element inside the collection and the element is inserted at the end of this collection.
- **E.g. 1)**

```
import java.util.*;
class AddMethodExample1
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection coll = new ArrayList();
            System.out.println(coll);
            for(int i = 10; i<=100 ; i+=10)
            {
                    coll.add(i);
            }
            System.out.println(coll);
    }
}
```

**Output:-**

```
[]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

- **E.g. 2)**

```
import java.util.*;
class AddMethodExample2
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection coll = new TreeSet();
            System.out.println(coll.add(1));
            System.out.println(coll.add(3));
            System.out.println(coll.add(2));
            // System.out.println(coll.add("1"));    //RuntimeException
            Collection coll1 = new HashSet();
            System.out.println(coll1.add(10));
            System.out.println(coll1.add(30));
            System.out.println(coll1.add(20));
            System.out.println(coll1.add("20"));
    }
}
```

**Output:-**

```
true
true
true
true
true
true
true
```

**RuntimeException:- Cannot store heterogenous data in TreeSet.**

- add() method returns true if the collection accepts the element or else returns false if it is refused by the collection.
- Exception thrown by this method are,
  i. ClassCastException
  ii. NullPointerException
  iii. IllegalArgumentException

## 2. public abstract boolean addAll(Collection c):-

- This method is used to add group of objects inside a collection.
- It will add all the elements at the end of this collection.
- It returns true if collection accepts the element or else returns false.
- Exception thrown by the method are,
  i. ClassCastException
  ii. NullPointerException
  iii. IllegalArgumentException
- **E.g. 1)**

```
import java.util.*;
class AddAllMethodExample1
{
@SuppressWarnings("unchecked")
        public static void main(String[] args)
        {
                Collection coll = new HashSet();
                coll.add(10);
                coll.add(20);
                System.out.println(coll);
                Collection coll1 = new ArrayList();
                coll1.add("fifty");
                coll1.add(20);
                coll1.add("sixty");
                System.out.println(coll1);
                System.out.println(coll.addAll(coll1));
                System.out.println(coll);
        }
}
```

**Output:-**

```
[20, 10]
[fifty, 20, sixty]
true
[sixty, 20, fifty, 10]
```

- **E.g. 2)**

```
import java.util.*;
class AddAllMethodExample2
{
        @SuppressWarnings("unchecked")
        public static void main(String[] args)
        {
                Collection coll = new ArrayList();
                coll.add(10);
                coll.add(20);
                System.out.println(coll);
                Collection coll1 = new ArrayList();
                coll1.add("fifty");
                coll1.add(20);
                coll1.add("sixty");
                System.out.println(coll1);
                System.out.println(coll.addAll(coll1));
                System.out.println(coll);
        }
}
```

**Output:-**

```
[10, 20]
[fifty, 20, sixty]
true
[10, 20, fifty, 20, sixty]
```

- **E.g. 3)**

```
import java.util.*;
class AddAllMethodExample3
{
        @SuppressWarnings("unchecked")
        public static void main(String[] args)
        {
                Collection coll = new TreeSet();
                coll.add(10);
                coll.add(20);
                System.out.println(coll);
                Collection coll1 = new ArrayList();
                coll1.add("fifty");
                coll1.add(20);
                coll1.add("sixty");
                System.out.println(coll1);
                System.out.println(coll.addAll(coll1));
                System.out.println(coll);
        }
}
```

**Output:-**

```
[10, 20]
[fifty, 20, sixty]
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java.lang.String (java.lang.Integer and java.lang.String are in module java.base of loader 'bootstrap')
    at java.base/java.lang.String.compareTo(String.java:142)
    at java.base/java.util.TreeMap.put(TreeMap.java:849)
    at java.base/java.util.TreeMap.put(TreeMap.java:569)
    at java.base/java.util.TreeSet.add(TreeSet.java:259)
    at java.base/java.util.AbstractCollection.addAll(AbstractCollection.java:338)
    at java.base/java.util.TreeSet.addAll(TreeSet.java:313)
    at AddAllMethodExample3.main(AddAllMethodExample3.java:16)
```

### 3. public abstract boolean remove(Object obj):-

- This method removes single instance of specified element from this collection.
- return type of this method is boolean.
- It returns true if the first instance of this specified element is been removed from this collection or else returns false.
- **E.g.**

A java program convert a heterogeneous collection to homogeneous collection.

```
import java.util.*;
class HetroToHomo
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection coll = new ArrayList();
            coll.add(10);
            coll.add("20");
            coll.add(30);
            coll.add("40");
            coll.add(true);
            coll.add(50);
            coll.add('S');
            coll.add("Hello");
            coll.add(10.00);
            System.out.println(coll);
            Collection coll1 = new ArrayList();
            for (Object e : coll)
            {
                    if(e instanceof Integer)
                            coll1.add(e);
            }
            System.out.println(coll1);
    }
}
```

**Output:-**

```
[10, 20, 30, 40, true, 50, S, Hello, 10.0]
[10, 30, 50]
```

**4. public abstract boolean removeAll(Colleciton c):-**

- This method removes all the elements from this collection that are present in the specified collection.
- The return type of this method is boolean.
- If the group of objects are present, it will remove them will return true or else false.
- This method throws following exceptions,
  i. NullPointerException
  ii. ClassCastException
- **E.g. 1)**

```
import java.util.*;
class RemoveAllExample1
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection coll = new ArrayList();
            coll.add(10);
            coll.add("20");
            coll.add(30);
            coll.add("40");
            coll.add(50);
            coll.add("hello");
            coll.add(true);
            coll.add('S');
            System.out.println(coll);
            Collection coll1 = new ArrayList();
            for(Object obj : coll)
            {
                    if(obj instanceof Integer)
                            coll1.add(obj);
            }
            System.out.println(coll1);
            System.out.println(coll.removeAll(coll1));
            System.out.println(coll);
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50, hello, true, S]
[10, 30, 50]
true
[20, 40, hello, true, S]
```

**E.g. 2)**
Plindrome Name

```
import java.util.*;
class RemoveAllExample2
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection<String> coll = new ArrayList<String>();
            coll.add("ramesh");
            coll.add("nayan");
            coll.add("suresh");
            coll.add("nitin");
            coll.add("mukesh");
            coll.add("naman");
            coll.add("mahesh");
            System.out.println(coll);
            Collection pal = palindrome(coll);
            System.out.println(pal);
            coll.removeAll(pal);
            System.out.println(coll);
    }
    @SuppressWarnings("unchecked")
    public static Collection<String> palindrome(Collection<String> coll)
    {
            Collection<String> pal = new ArrayList();
            for(String str : coll)
            {
                    if(!(str.contentEquals(new StringBuffer(str).reverse())))
                            pal.add(str);
            }
            return pal;
    }
}
```

**Output:-**

```
[ramesh, nayan, suresh, nitin, mukesh, naman, mahesh]
[ramesh, suresh, mukesh, mahesh]
[nayan, nitin, naman]
```

5. **public abstract boolean contains(Object obj):-**
   - This method returns true if this collection contains the specified elements.
   - It returns true only if the collection contains the elements or else return false.
   - E.g.

```
import java.util.*;
class ContainsMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection coll = new ArrayList();
            coll.add(10);
            coll.add(20);
            coll.add(null);
            coll.add("30");
            System.out.println(coll.contains(null));
            System.out.println(coll.contains("30"));
            System.out.println(coll.contains(true));

            Collection coll1 = new TreeSet();
            coll1.add(10);
            coll1.add(20);
            // System.out.println(coll1.contains(null));     //NullPointerException
            // System.out.println(coll1.contains("30"));     //ClassCastException
            // System.out.println(coll1.contains(true));     //ClassCastException
            System.out.println(coll1.contains(10));
    }
}
```

**Output:-**

```
true
true
false
true
```

### 6. public abstract boolean containsAll(Collection coll):-

- This method returns true if this collection contains all the elements from the specified collection or else return false.
- Exception thrown by containsAll()
  i. ClassCastException
  ii. NullPointerException
- E.g.

```
import java.util.*;
class ContainsAllMethodExample{
        @SuppressWarnings("unchecked")
        public static void main(String[] args) {
                Collection coll = new ArrayList();
                coll.add(10);
                coll.add(20);
                coll.add(null);
                coll.add("30");
                Collection coll1 = new ArrayList();
                coll1.add(10);
                coll1.add(20);
                coll1.add(null);
                coll1.add("30");
                coll1.add(true);
                System.out.println(coll1.containsAll(coll));
        }
}
```

**Output:-**

```
true
```

### 7. public abstract int size():-

- This method returns the number of elements present inside this collection.
- E.g.

```
import java.util.*;
class SizeMethodExample{
        @SuppressWarnings("unchecked")
        public static void main(String[] args) {
                List coll = new ArrayList();
                coll.add(10);
                coll.add(20);
                coll.add(30);
                coll.add(40);
                coll.add(50);
                System.out.println(coll.size());
                for(int i=0;i<coll.size();i++){
                        System.out.println(coll.get(i));
                }
        }
}
```

**Output:-**

```
5
10
20
30
40
50
```

**8. public abstract boolean retainAll(Collection c):-**

- This method retains only the elements that are contains in this specified collection.
- In simple words it removes collection elements that are not present in this specified collection argument.
- This method throws,
  i. NullPointerException
  ii. ClassCastException
  iii. UnsupportedOperationException

★ **Write a java program to remove even elements from the collection.**
**Input :- [1,2,3,4,5,6,7,8,9]**
**Output :- [1,3,4,7,9]**

```
import java.util.*;
class RemoveEvenElements
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection<Integer> coll = new ArrayList();
            coll.add(1);
            coll.add(2);
            coll.add(3);
            coll.add(4);
            coll.add(5);
            coll.add(6);
            coll.add(7);
            coll.add(8);
            coll.add(9);
            System.out.println(coll);
            Collection<Integer> odd = removeEven(coll);
            coll.retainAll(odd);
            System.out.println(coll);
    }
    @SuppressWarnings("unchecked")
    public static Collection<Integer> removeEven(Collection<Integer> coll)
    {
            Collection<Integer> odd = new ArrayList();
            for (Integer i : coll)
            {
                    if(i%2!=0)
                            odd.add(i);
            }
            return odd;
    }
}
```

**Output:-**
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 3, 5, 7, 9]

**9. public abstract boolean clear():-**

- This method removes all the elements from this collection.
- It throws
  i. UnsupportedOperationException
- E.g.

```
import java.util.*;
class ClearMethodExample
{
        @SuppressWarnings("unchecked")
        public static void main(String[] args)
        {
                Collection coll = new ArrayList();
                coll.add(10);
                coll.add(20);
                coll.add(30);
                coll.add(40);
                coll.add(50);
                System.out.println(coll);
                coll.clear();
                System.out.println(coll);
        }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
[]
```

**10. public abstract boolean isEmpty():-**

- This method returns true if this collection contains no elements in it or else return false.
- E.g.

```
import java.util.*;
class IsEmptyMethodExample
{
        @SuppressWarnings("unchecked")
        public static void main(String[] args)
        {
                Collection coll = new ArrayList();
                System.out.println(coll.isEmpty());
                coll.add(10);
                System.out.println(coll.isEmpty());
        }
}
```

**Output:-**

```
true
false
```

### 11. public abstract Object[] toArray():-

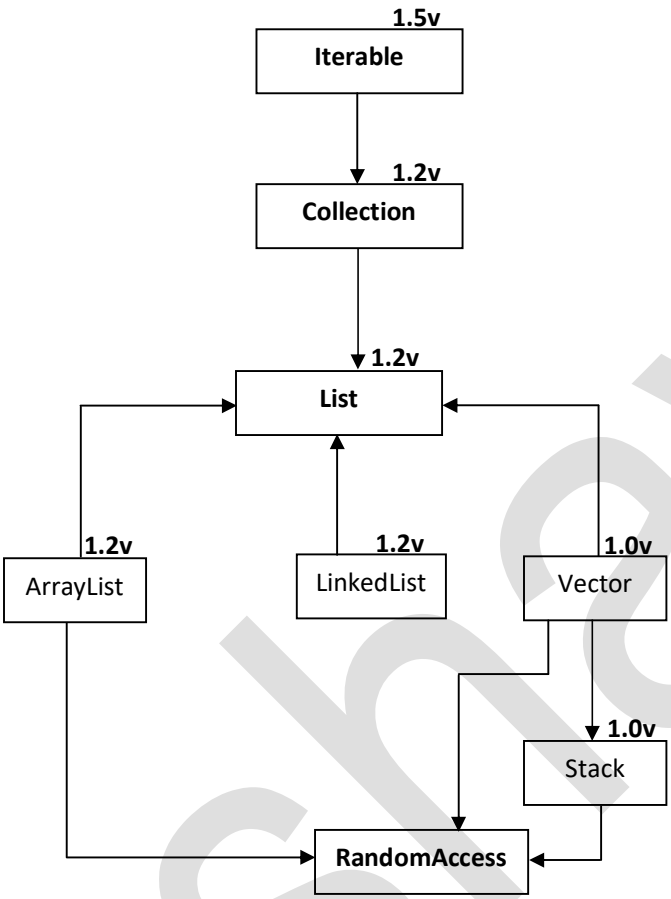
- This method returns an Object type of array containing all of the elements from this collection.
- The return type of this method is an Object[], the reason behind it is that it stores a heterogeneous data in it.
- E.g.

```
import java.util.*;
class ToArrayExample1
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Collection coll = new ArrayList();
            coll.add(10);
            coll.add(20);
            coll.add(30);
            coll.add(40);
            Object[] arr = coll.toArray();
            int [] arr1 = new int[arr.length];
            for(int i=0;i<arr.length;i++)
            {
                    arr1[i] =(Integer) arr[i];
            }
            System.out.println(Arrays.toString(arr1));
    }
}
```

**Output:-**

[10, 20, 30, 40]

# ★ List interface :-




## ★ List interface:-

- List is a child interface of Collection interface.
- Its an ordered collection of elements (object).
- It maintains the insertion order of elements.
- It allows us to store duplicate elements and multiple null values.
- It was introduced in JDK 1.2 version.
- It has four implementation classes,
  1. ArrayList
  2. LinkedList
  3. Vector
  4. Stack

### Methods of List interface:-

1. public abstract void add(int index,E ele);
2. public abstract boolean addAll(int index, Collection c);
3. public abstract E remove(int index);
4. public abstract int indexOf(Object obj);
5. public abstract int lastIndexOf(Object obj);
6. public abstract E get(int index);
7. public abstract E set(int index, E ele);
8. public abstract ListIterator<E> listIterator();

1. **public abstract void add(int index, E ele);**
   - This method inserts the specified element at a specified position in this List.
   - E.g.

```
import java.util.*;
class addMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            System.out.println(list);
            list.add(2,50);
            System.out.println(list);
    }
}
```

**Output:-**

```
[10, 20, 30, 40]
[10, 20, 50, 30, 40]
```

2. **public abstract void allAll(int index, Collection c);**
   - It inserts all of the elements in this specified collection into this list at the specified position.
   - E.g.

```
import java.util.*;
class addAllMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list1 = new ArrayList();
            list1.add(10);
            list1.add(20);
            list1.add(30);
            list1.add(40);
            List list2 = new ArrayList();
            list2.add("akshay");
            list2.add("java");
            list2.add("hello");
            list2.add("notebook");
            System.out.println(list1.addAll(3,list2));
            System.out.println(list1);
    }
}
```

**Output:-**

```
true
[10, 20, 30, akshay, java, hello, notebook, 40]
```

### 3. public abstract E remove(int index);

- This method removes the element which is at the specified position in this List.
- After removing it returns the same element.
- E.g.

```
import java.util.*;
class removeMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            list.add(50);
            System.out.println(list);
            System.out.println(list.remove(1));
            System.out.println(list);
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
20
[10, 30, 40, 50]
```

### 4. public abstract E set(int index, E ele);

- This method replaces the element from the specified position in this list with the specified element.
- This method returns the previous element which has been replaced.
- E.g.

```
import java.util.*;
class setMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            list.add(50);
            System.out.println(list);
            System.out.println(list.set(2,70));
            System.out.println(list);
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
30
[10, 20, 70, 40, 50]
```

5. **public abstract E get(int index):-**
   - This method returns the element from the specified position in this List.
   - The return type of this method is E (type of object).
   - E.g.

```
import java.util.*;
class getMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            list.add(50);
            System.out.println(list);
            System.out.println(list.get(2));
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
30
```

6. **public abstract int indexOf(Object obj);**
   - This method returns the index of the first occurance of the specified element in this List or else it returns -1 if the List does not contains the element.
   - E.g.

```
import java.util.*;
class indexOfMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            list.add(50);
            System.out.println(list);
            System.out.println(list.indexOf(40));
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
3
```

7. **public abstract int lastIndexOf(Object obj);**
   - This method returns the index of the last occurance of the specified element in this List or else returns -1 if the List does not contains the element.
   - E.g.

```
import java.util.*;
class lastIndexOfMethodExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add(20);
            list.add(30);
            list.add(40);
            list.add(50);
            System.out.println(list);
            System.out.println(list.lastIndexOf(20));
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
1
```

# 1. ArrayList :-

- ArrayList is an implementation class of List interface.
- ArrayList was introduced in JDK 1.2 version and derived in java.util package.
- The internal implementation of ArrayList is growable array (data structure).
- It stores the data using indexing.
- It preserves the insertion order.
- It allows us to store duplicate elements.
- Also store multiple null values.
- Since ArrayList is implemented using a growable array which used indexing so we can perform fast searching of elements.
- For this fast searching of elements ArrayList implements RandomAccess interface (marker interface).
- ArrayList is not synchronized because of which multiple threads can work at a time and it is faster in performance compared to Vector.
- As it is not synchronized it's not thread safe.
- The initial capacity of an ArrayList is 10.
- We can store both homogeneous and heterogeneous data elements in it.
- Other interfaces which are implemented by ArrayList are,
  i. Collection,
  ii. Iterable,
  iii. Serializable,
  iv. Cloneable

★ **Constructors of ArrayList:-**

1. public ArrayList();
2. public ArrayList(int capacity);
3. public ArrayList(Collection c);

**1. public ArrayList():-**

- It constructs an empty list with an initial capacity of 10.
- E.g.

```
import java.util.*;
class noArgumentConstructorExample
{
    public static void main(String[] args)
    {
            List<Integer> list = new ArrayList<>();
            System.out.println(list.size());
            for(int i=10;i<=100;i+=10)
                    list.add(i);
            System.out.println(list.size());
            list.add(110);
            //(initialCapacity*3/2)+1 = newCapacity
            //(10*3/2)+1 = 16
            System.out.println(list.size());
    }
}
```

**Output:-**

```
0
10
11
```

**Note:-**

- Initial capacity of ArrayList is 10, once the List is full and if we try to store the next element it will create a new ArrayList with a capacity of,

$$\left(\frac{\text{initialCapacity} * 3}{2}\right) + 1 = \text{newCapacity}$$

i.e.

$$\left(\frac{10 * 3}{2}\right) + 1 = 16$$

- The new ArrayList have capacity of 16 and will copy all elements from old array list into newly created ArrayList.

**2. public ArrayList(int initialCapcity):-**

- It constructs an empty list with a specified initial capacity.
- E.g.

```
import java.util.*;
class capacityArgumentConstructorExample
{
    public static void main(String[] args)
    {
            List<Integer> list = new ArrayList<>(20);
            for(int i=10;i<=200;i+=10)
                    list.add(i);
            System.out.println(list.size());          //(initialCapacity*3/2)+1
            list.add(210);                            //(20*3/2)+1 = 31
            System.out.println(list.size());
    }
}
```

**Output:-**

```
20
21
```

**Note:-**

- To alter the capacity of an ArrayList we can use following method,

**i. ensureCapacity(int capacity):-**

- This method create new ArrayList with specified capacity.

**ii. trimToSize():-**

- This method reduce the capacity of ArrayList equivalent to the number of elements present in it.

**E.g. 1)**

```
import java.util.*;
class ensureCapacityExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<>(100);
            for(int i=1;i<=100;i++)
                    list.add(i);
            list.ensureCapacity(300);
            for(int i=1;i<=200;i++)
                    list.add(i);
            System.out.println(list);
    }
}
```

**Output:-**

[1,2,3,4,……..,200]

**E.g. 2)**

```
import java.util.*;
class TrimToSizeMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<>(100);
            for(int i=1;i<=90;i++)
                    list.add(i);
            list.trimToSize();
            System.out.println(list);
    }
}
```

**Output:-**

[1, 2, 3, 4, ……., 90]

### 3. public ArrayList(Collection c):-

- It constructs a list containing elements of a specified collection.
- Basically it is used to convert any other type of collection into ArrayList.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructorExample1
{
    public static void main(String[] args)
    {
            TreeSet<Integer> ts = new TreeSet<>();
            for(int i=1;i<=10;i++)
                    ts.add((int)(Math.random()*10));
            System.out.println(ts);

            ArrayList<Integer> al = new ArrayList<>(ts);
            System.out.println(al);

            LinkedHashSet<Integer> lhs = new LinkedHashSet<>(ts);
            lhs.addAll(ts);

            ArrayList<Integer> al1 = new ArrayList<>(lhs);
            System.out.println(al1);

            PriorityQueue<Integer> pq = new PriorityQueue<>(ts);

            ArrayList<Integer> al2 = new ArrayList<>(pq);
            System.out.println(al2);

    }
}
```

### ★ Disadvantages of ArrayList:-

- If we want to perform operation like insertion / removing of an element from list it is going to consume lot of time because it performs **shifting** operation on removing an element / after insertion.

**Note:-**

**To overcome this problem we have LinkedList.**

## -: Implementation of User Created ArrayList in java :-

```
import java.util.*;
@SuppressWarnings("unchecked")
class UserArrayList<E>
{
	public static final int DEFAULT_CAPACITY = 10;
	int index = 0;
	E[] arr;
	@SuppressWarnings("unchecked")
	public UserArrayList()
	{
		arr = (E[]) new Object[DEFAULT_CAPACITY];
	}
	public UserArrayList(int cap)
	{
		if(cap<0)
			throw new NegativeArraySizeException("invalid capacity : "+cap);
		arr = (E[]) new Object[cap];
	}
	public UserArrayList(Collection<E> coll)
	{
		arr = (E[]) coll.toArray();
		index = coll.size();
	}

	@Override
	public String toString()
	{
		if(index==0)
			return"[]";
		String op = "[";
		for(int i=0;i<index-1;i++)
		{
			op+=arr[i]+", ";
		}
		op = op+arr[index-1]+" ]";
		return op;
	}
	public boolean add(E ele)
	{
		if(arr.length<=size())
		{
			int newCapacity = newCapacity(arr.length);
			E[] newArr = (E[]) new Object[newCapacity];
			for(int i=0;i<arr.length;i++)
			{
				newArr[i] = arr[i];
			}
			arr = newArr;
		}
		arr[index++] = ele;
		return true;
	}
	public int size()
	{
		return index;
	}
	public E remove(int index)
	{
```

```
		if(index<0 || index>=size())
			throw new IndexOutOfBoundsException("USER YOU HAVE ENTERED A WRONG
INDEX "+index);
		E ele = arr[index];
		arr[index] = null;
		for(int i = index;i<arr.length-1;i++)
		{
			arr[i] = arr[i+1];
		}
		arr[this.index] = null;
		this.index--;
		return ele;
	}
	public int indexOf(E ele)
	{
		for(int i=0;i<size();i++)
		{
			if(arr[i].equals(ele))
				return i;
		}
		return -1;
	}
	public int lastIndexOf(E ele)
	{
		for(int i=size()-1;i>=0;i--)
		{
			if(arr[i].equals(ele))
				return i;
		}
		return -1;
	}
	public int newCapacity(int oldCapacity)
	{
		return ((oldCapacity*3)/2)+1;
	}
	public E get(int index)
	{
		if(index<0 || index>=size())
			throw new IllegalArgumentException("WRONG INDEX "+index);
		return arr[index];
	}
	public E set(int index,E ele)
	{
		if(index<0 || index>=size())
			throw new IllegalArgumentException("WRONG INDEX "+index);
		E temp = arr[index];
		arr[index] = ele;
		return temp;
	}
	public void addAll(Collection<E> coll)
	{
		// E[] newArr = (E[]) new Object[arr.length];
		if(arr.length<=(coll.size()+size()))
		{
			int newCap = newCapacity(arr.length);
			E[] newArr = (E[]) new Object[newCap];
			if(newArr.length<=(coll.size()+size()))
			{
				int newCap1 = newCapacity(newCap);
				newArr = (E[]) new Object[newCap1];
```

```
                        }
                        for(int i=0;i<size();i++)
                                newArr[i] = arr[i];
                        arr = newArr;
                }
                // for(int i=0;i<size();i++){
                //      newArr[i] = (E) arr[i];
                // }
                Object[] objArr = coll.toArray();
                for(int i=0,j=size();i<coll.size();i++,j++)
                {
                        arr[j] = (E) objArr[i];
                }
                // arr = newArr;
                this.index +=coll.size();
        }
        public void removeAll(Collection<E> coll)
        {
                E[] newArr = (E[]) new Object[arr.length];
                int newIndex = 0;
                for(int i=0;i<size();i++)
                {
                        if(!coll.contains(arr[i]))
                        {
                                newArr[newIndex++] = arr[i];
                        }
                }
                arr = newArr;
                index = newIndex;
        }
        public void add(int index,E ele)
        {
                if(index<0)
                        throw new IllegalArgumentException("WRONG INDEX "+index);
                if(arr.length<=(size()+1))
                {
                        int newCap = newCapacity(arr.length);
                        E[] newArr = (E[]) new Object[newCap];
                        for(int i=0;i<size();i++)
                                newArr[i] = arr[i];
                        arr = newArr;
                }
                for(int i=size();i>=index;i--)
                {
                        E temp = arr[i];
                        arr[i] = arr[i+1];
                        arr[i+1] = temp;
                }
                arr[index] = ele;
                this.index++;
        }
}
```

★ **Explanation:-**

1. **public String toString():-**
   - The toString() returns the String representation of an ArrayList.
   - If the ArrayList contains no elements in it. It returns an empty square brackets or else return the elements enclosed in square bracket separated by comma.

2. **public boolean add(E ele):-**
   - add() is used to add the element at the last in the ArrayList.
   - It returns true if the element is added to the collection or else returns false.
   - Every time we add the element the index is updated with the value of one and it checks the capacity of collection.
   - If the capacity is less or equal to size then new ArrayList is created with capacity of (initialCapacity*3)/2 +1.
   - Previous array element copied into new ArrayList.

3. **public E remove(int index):-**
   - remove() used to remove the element from ArrayList which is present at the specified index.
   - This method returns the removed element.
   - When er remove the element from ArrayList the index is updated by decrement of one.

4. **public int size():-**
   - size() is used to return the size of ArrayList i.e. it returns the number of elements present at the ArrayList.

5. **public int indexOf(E ele):-**
   - indexOf() is used to get the first occurance of the specified element.
   - It checks the element is present in the ArrayList or not if present returns the index or else returns -1.

6. **public int lastIndexOf(E ele):-**
   - lastIndexOf() is used to get the last occurance of the specified element.
   - It checks the element from last of the ArrayList.
   - It iterate the ArrayList in reverse, checks for the element for present or not.
   - If the element is present it returns the index or else returns -1.

7. **public int newCapacity(int oldCapacity):-**
   - It returns an integer value based on the formula i.e. (oldCapacity*3)/2 +1.

8. **public E set(int index, E ele):-**
   - set() is used to set the specified element at the specified position in ArrayList.
   - Firstly it checks that the given index is valis or not i.e. index should not be less than zero or greater than or equals to the size() of ArrayList.
   - The given index element is stored in temporary variable and the given element is assigned to the given index of ArrayList.
   - Temporary variable is returned.

9. **public E get(int index):-**
   - get() is used to the element which specified at the given index from ArrayList.
   - Firstly it checks that the given index is valis or not i.e. index should not be less than zero or greater than or equals to the size() of ArrayList.
   - Returns the element specied at the given index.

### 10. public void addAll(Collection<E> coll):-

- addAll() is used to add all the elements to the ArrayList from the specified collection.
- Firstly it checks the current capacity of ArrayList is enough for storing elements or not.
- If not it increments the capacity of ArrayList based on the formula **(currentCapacity*3)/2+1.**
- Creates a newArray with the new capacity and copy the ArrayList and then dereferenced to ArrayList.
- At last given collection array is converted into Object[] and typecasted to ArrayList at the last.
- The index variable of ArrayList is updated by **oldArrayList size() + collectionSize().**

### 11. public void removeAll(Collection <E> coll):-

- removeAll() is used to remove all the elements from ArrayList which are present in specified collection.
- First it creates a new array with ArrayList size and new index variable initialized to zero.
- Then the method iterate a loop on the ArrayList size, it checks that the current ArrayList element is present in the collection or not with the help of **contains()**.
- If not it is inserted into new array and new index updated by increase of one value.
- After the loop the new array is dereferenced to ArrayList and ArrayList index is updated to the new index.

### 12. public void add(int index, E ele):-

- This method is used to add an element at a specified position in the ArrayList.
- First it checks that the given index is valid or not i.e. it should not be less than zero or it should not be greater than or equals to the size() of ArrayList.
- Then it checks that if the ArrayList have enough capacity to store the element or not i.e capacity<=size()+1.
- If no then the newCapacity array will be created and the old array is copied to the new array.
- Then new array is dereferenced to the ArrayList array.
- Then method iterate a loop from the last and keep swapping of elements till to the index specified.
- After the loop the element is inserted to the specified index.
- And the ArrayList index variable is updated by increment of one.

**Note:-**

- We cannot create generic type of array in java, so while implementing ArrayList we created an Object type of array and typecasted into generic array i.e.

**(E[]) new Object[size];**

## ★ RandomAccess interface:-

- RandomAccess interface is a marker interface.
- There are few classes in Collection which implements this interface such as,
  i. ArrayList,
  ii. Stack,
  iii. Vector
- It is used for fast searching of elements, with constant time complexity.
- A class implementing RandomAccess provides a meta data that it can perform fast searching of elements.

**Note:-**

- RandomAccess is implemented by those classes which uses indexing for storing or retrival of elements with generics linear time complexity.

# 2. LinkedList:-

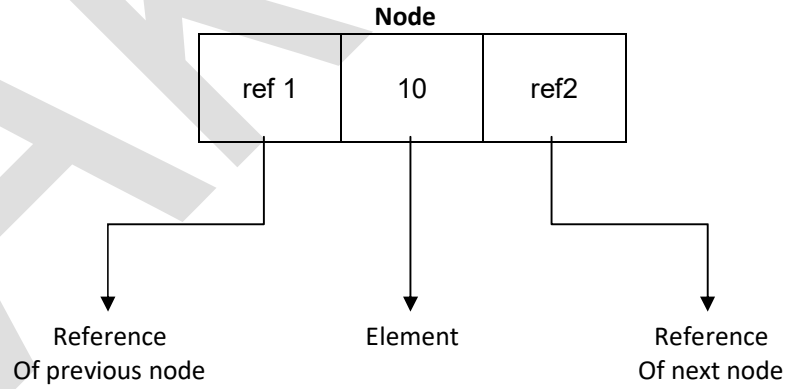
- LinkedList is an implementation class of List interface.
- LinkedList was introduced in JDK 1.2 version and derived in java.util package.
- It preserves the insertion order.
- We can store duplicate elements as well as multiple null values.
- LinkedList is not synchronized.
- Internal implementation of LinkedList is doubly LinkedList in java.
- It stores the data in a form of node (inner class).
- LinkedList stores the data (node) at non-contiguous location and its best suitable for insertion or removing of element because it does not perform shifting operation.
- It generates O(1) time complexity for removing or insertion of operation.
- Other interfaces which are implemented by LinkedList are,
  i. Collection,
  ii. Iterable,
  iii. Serializable,
  iv. Cloneable,
  v. Deque
- LinkedList can be implemented in three ways,
  1. Singaly Linked List
  2. Doubly Linked List
  3. Circular Linked List

★ **Doubly Linked List:-**

- In computer science a doubly linked list is link data structure that consists of a set of sequentially linked records called as node.

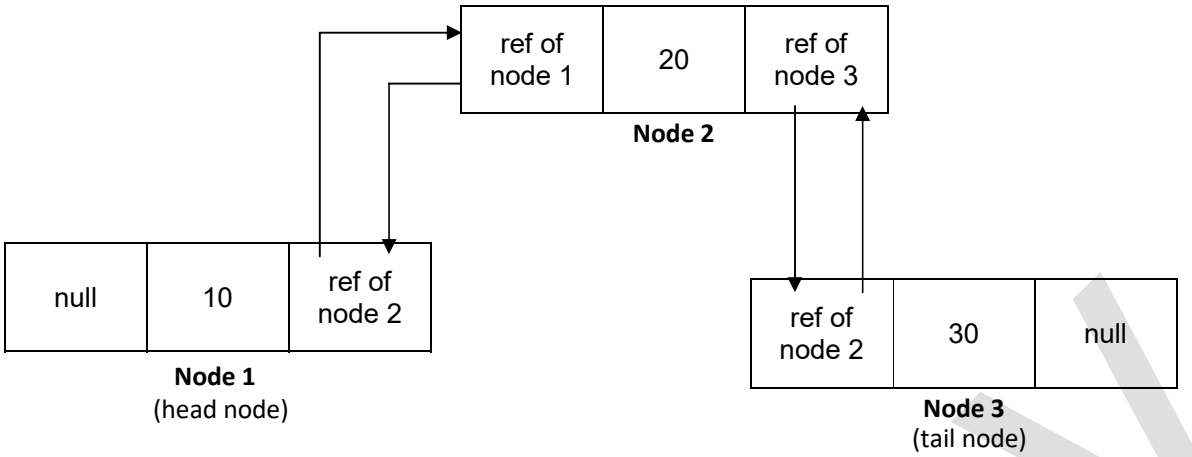
★ **Node:-**

- A node in java is an inner class which is present inside LinkedList.
- It mainly consists of three parts,
  1. Reference of previous node
  2. Element
  3. Reference of next node

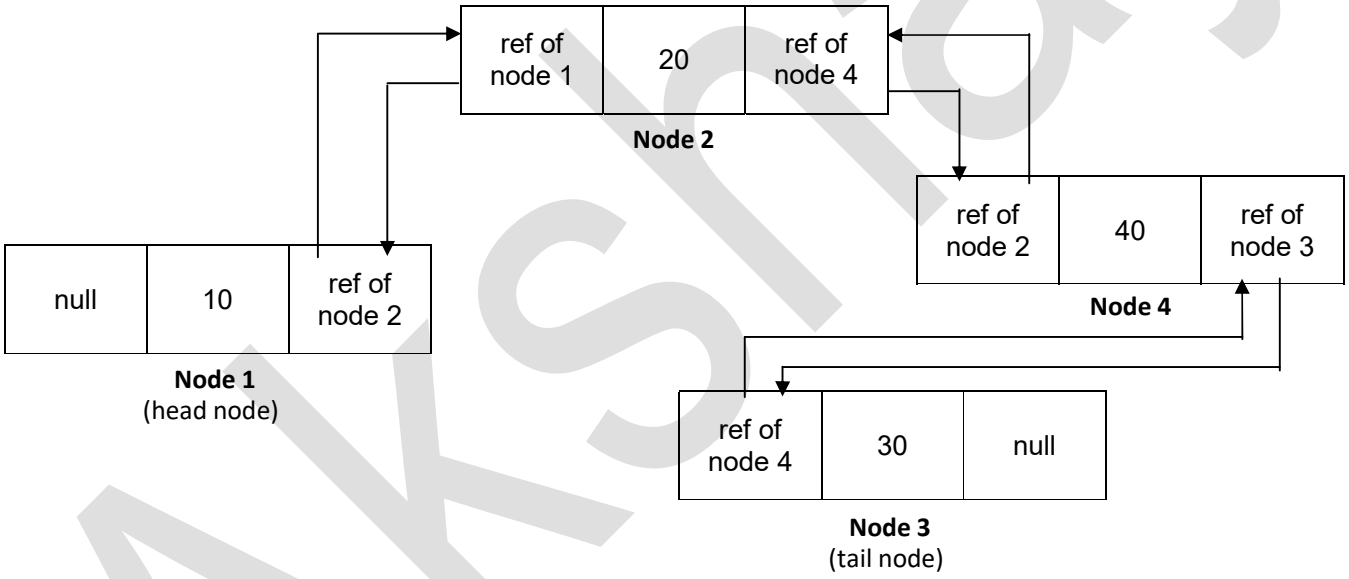



**Representation Of Node**

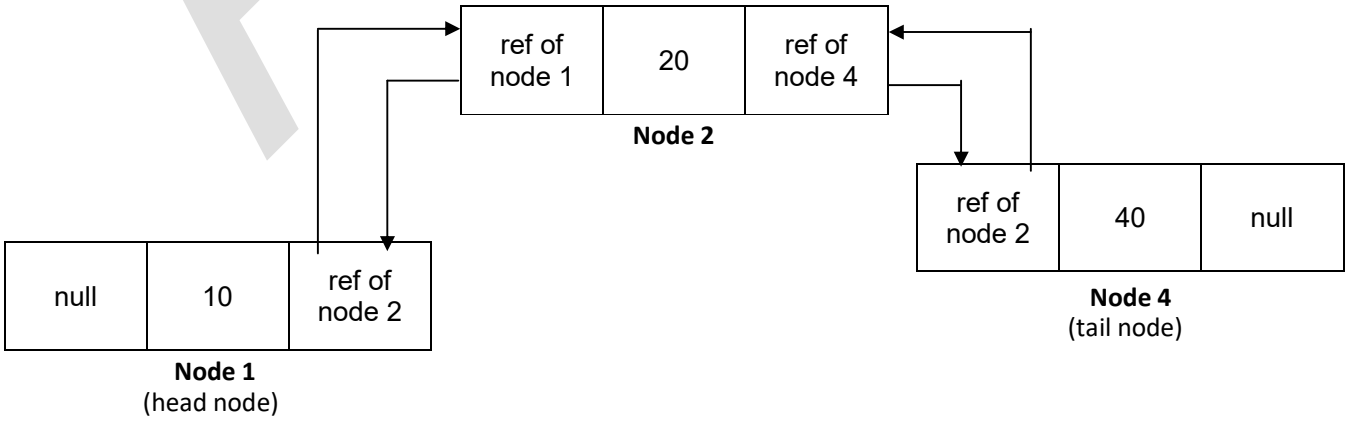
★ **Representation of LinkedList:-**




★ **Representation of LinkedList after adding an element:-**




★ **Representation of LinkedList after removing an element:-**



## ★ Constructors of LinkedList:-

1. public LinkedList()
2. public LinkedList(Collection c)

### 1. public LinkedList():-

- It is used to construct empty list which doesn't have any elements in it.
- E.g.

```
import java.util.*;
class NoArgumentConstructorExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            LinkedList list = new LinkedList();
        System.out.println(list);
            list.add(10);
            list.add(20);
            list.add(30);
            System.out.println(list);
    }
}
```

**Output:-**

```
[]
[10, 20, 30]
```

### 2. public LinkedList(Collection c):-

- It constructs a new list that contains all the elements from specified collection.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructorExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            ArrayList list1 = new ArrayList();
            list1.add(10);
            list1.add(20);
            list1.add(30);
            list1.add(40);
            list1.add(50);
            System.out.println(list1);
            LinkedList list2 = new LinkedList(list1);
            System.out.println(list2);
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
```

## ★ Methods of LinkedList:-

1. public E getFirst()
2. public E getLast()
3. public E removeFirst()
4. public E removeLast()
5. public void addFirst(E ele)
6. public void addLast(E ele)
7. public boolean contains(Object obj)

### 1. public void addFirst(E ele):-

- This method inserts the specified element at the beginning of this List.
- E.g.

```
import java.util.LinkedList;
class AddFirstExample
{
    public static void main(String[] args)
    {
            LinkedList<String> list = new LinkedList<>();
            list.add("B");
            list.add("C");
            list.add("D");
            System.out.println("Original LinkedList : "+list);
            list.addFirst("A");
            System.out.println("LinkedList after addFirst : "+list);
    }
}
```

**Output:-**

```
Original LinkedList : [B, C, D]
LinkedList after addFirst : [A, B, C, D]
```

### 2. public addLast(E ele):-

- This method inserts the specified element to the end of this List.
- E.g.

```
import java.util.LinkedList;
class AddLastExample
{
    public static void main(String[] args)
    {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            System.out.println("Original LinkedList : "+list);
            list.addLast("D");
            System.out.println("LinkedList after addLast : "+list);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C]
LinkedList after addLast : [A, B, C, D]
```

### 3. public E getFirst():-

- This method returns the first element in this List.
- E.g.

```
import java.util.LinkedList;
class AddLastExample
{
    public static void main(String[] args)
    {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            System.out.println("Original LinkedList : "+list);
            list.addLast("D");
            System.out.println("LinkedList after addLast : "+list);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C]
First element of the LinkedList : A
LinkedList after addFirst : [Start, A, B, C]
New first element : Start
```

### 4. public E getLast():-

- This method returns the last element in this List.
- E.g.

```
import java.util.LinkedList;
class GetLastExample
{
    public static void main(String[] args)
    {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            System.out.println("Original LinkedList : "+list);
            String lastElement = list.getLast();
            System.out.println("Last element of the LinkedList : "+lastElement);
            list.addLast("D");
            System.out.println("LinkedList after addLast : "+list);
            String newLastElement = list.getLast();
            System.out.println("New first element : "+newLastElement);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C]
Last element of the LinkedList : C
LinkedList after addLast : [A, B, C, D]
New first element : D
```

5. **public E remove(int index):-**
   - This method removes the element at the specified position in this list.
   - E.g.

```
import java.util.LinkedList;
class RemoveExample{
    public static void main(String[] args) {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            list.add("D");
            System.out.println("Original LinkedList : "+list);
            String removeElement = list.remove(1);
            System.out.println("LinkedList after removing element at index 1 : "+list);
            System.out.println("Removed Element : "+removeElement);
            removeElement = list.remove(2);
            System.out.println("LinkedList after removing element at index 2 : "+list);
            System.out.println("Removed Element : "+removeElement);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C, D]
LinkedList after removing element at index 1 : [A, C, D]
Removed Element : B
LinkedList after removing element at index 2 : [A, C]
Removed Element : D
```

6. **public void add(int index, E ele):-**
   - This method inserts the specified element at the specified position in this list.
   - E.g.

```
import java.util.LinkedList;
class AddExample{
    public static void main(String[] args) {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("D");
            System.out.println("Original LinkedList : "+list);
            list.add(2,"C");
            System.out.println("LinkedList after adding 'C' at index 2 : "+list);
            list.add(0,"Start");
            System.out.println("LinkedList after adding 'Start' at index 0 : "+list);
            list.add(5,"End");
            System.out.println("LinkedList after adding 'End' at index 5 : "+list);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, D]
LinkedList after adding 'C' at index 2 : [A, B, C, D]
LinkedList after adding 'Start' at index 0 : [Start, A, B, C, D]
LinkedList after adding 'End' at index 5 : [Start, A, B, C, D, End]
```

7. **public E removeFirst():-**
   - This method remove and returns the first element from this list.
   - E.g.

```
import java.util.LinkedList;
class RemoveFirstExample{
    public static void main(String[] args) {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            System.out.println("Original LinkedList : "+list);
            String removedElement = list.removeFirst();
            System.out.println("LinkedList after removeFirst : "+list);
            System.out.println("Removed Element : "+removedElement);
            removedElement = list.removeFirst();
            System.out.println("LinkedList after another removeFirst : "+list);
            System.out.println("Removed Element : "+removedElement);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C]
LinkedList after removeFirst : [B, C]
Removed Element : A
LinkedList after another removeFirst : [C]
Removed Element : B
```

8. **public E removeLast():-**
   - This method remove and returns the last elemtn from this list.
   - E.g.

```
import java.util.LinkedList;
class RemoveLastExample{
    public static void main(String[] args) {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            System.out.println("Original LinkedList : "+list);
            String removedElement = list.removeLast();
            System.out.println("LinkedList after removeLast : "+list);
            System.out.println("Removed Element : "+removedElement);
            removedElement = list.removeLast();
            System.out.println("LinkedList after another RemoveLastExample : "+list);
            System.out.println("Removed Element : "+removedElement);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C]
LinkedList after removeLast : [A, B]
Removed Element : C
LinkedList after another removeLast : [A]
Removed Element : B
```

### 9. public boolean removeFirstOccurrence(Object obj):-

- Removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
- E.g.

```
import java.util.LinkedList;
class RemoveFirstOccurrenceExample{
    public static void main(String[] args) {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            list.add("D");
            list.add("B");
            System.out.println("Original LinkedList : "+list);
            boolean isRemoved = list.removeFirstOccurrence("B");
            System.out.println("LinkedList after removing first occurrence of 'B': "+list);
            System.out.println("Was the element removed? "+isRemoved);
            isRemoved = list.removeFirstOccurrence("Z");
            System.out.println("LinkedList after removing first occurrence of 'Z': "+list);
            System.out.println("Was the element removed? "+isRemoved);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C, D, B]
LinkedList after removing first occurrence of 'B': [A, C, D, B]
Was the element removed? true
LinkedList after removing first occurrence of 'Z': [A, C, D, B]
Was the element removed? false
```

### 10. public boolean removeLastOccurrence(Object obj):-

- This removes the last occurrence of the specified element in this list (when traversing list from head to tail).
- E.g.

```
import java.util.LinkedList;
class RemoveLastOccurrenceExample{
    public static void main(String[] args) {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            list.add("D");
            list.add("B");
            System.out.println("Original LinkedList : "+list);
            boolean isRemoved = list.removeLastOccurrence("B");
            System.out.println("LinkedList after removing last occurrence of 'B': "+list);
            System.out.println("Was the element removed? "+isRemoved);
            isRemoved = list.removeLastOccurrence("Z");
            System.out.println("LinkedList after removing last occurrence of 'Z': "+list);
            System.out.println("Was the element removed? "+isRemoved);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C, D, B]
LinkedList after removing last occurrence of 'B': [A, B, C, D]
Was the element removed? true
LinkedList after removing last occurrence of 'Z': [A, B, C, D]
Was the element removed? false
```

### 11. public E set(int index, E ele):-

- This method replaces the element at the specified position in this list with the specified element.
- E.g.

```
import java.util.LinkedList;
class SetExample
{
    public static void main(String[] args)
    {
            LinkedList<String> list = new LinkedList<>();
            list.add("A");
            list.add("B");
            list.add("C");
            list.add("D");
            System.out.println("Original LinkedList : "+list);
            String oldElement = list.set(2,"X");
            System.out.println("LinkedList after set operation : "+list);
            System.out.println("Replaced element : "+oldElement);
            oldElement = list.set(0,"Start");
            System.out.println("LinkedList after set operation at index 0 : "+list);
            System.out.println("Replaced element : "+oldElement);
    }
}
```

**Output:-**

```
Original LinkedList : [A, B, C, D]
LinkedList after set operation : [A, B, X, D]
Replaced element : C
LinkedList after set operation at index 0 : [Start, B, X, D]
Replaced element : A
```

**-: Implementation of User created LinkedList :-**

```
import java.util.*;
@SuppressWarnings("unchecked")
class SingallyLinkedList<E>
{
	int indx = 0;
	Node<E> head;
	Node<E> tail;
	@SuppressWarnings("hiding")
	// inner class
		static class Node<E>
		{
			E ele;
			Node<E> next;
			Node(E ele)
			{
				this.ele = ele;
			}
		}
	SingallyLinkedList()
	{
		super();
	}
	SingallyLinkedList(Collection<E> coll)
	{
		for (E item : coll)
		{
			addLast(item);
		}
	}
	@Override
	public String toString()
	{
		if(head == null)
		{
			return "[]";
		}
		String op ="[";
		Node<E> currentNode = head;
		while(currentNode.next!=null)
		{
			op += currentNode.ele+", ";
			currentNode = currentNode.next;
		}
		op += currentNode.ele+"]";
		return op;
	}
	public void addLast(E ele)
	{
		Node<E> newNode = new Node<E>(ele);
		if(head==null)
		{
			head = tail = newNode;
		}
		else
		{
			Node<E> currentNode = head;
			while(currentNode.next != null)
			{
				currentNode = currentNode.next;
```

```
                }
                currentNode.next = newNode;
        }
        indx++;
}
public void addFirst(E ele)
{
        Node<E> newNode = new Node<E>(ele);
        if(head==null)
        {
                head = tail = newNode;
        }
        else
        {
                newNode.next = head;
                head = newNode;
        }
        indx++;
}
public int size()
{
        return indx;
}
public E getFirst()
{
        if(head==null)
                throw new NoSuchElementException("User Linked List is empty");
        return head.ele;
}
public E getLast()
{
        if(tail==null)
                throw new NoSuchElementException("User Linked List is empty");
        return tail.ele;
}
public E removeFirst()
{
        if(head==null)
        {
                throw new NoSuchElementException("User the linked list is empty");
        }
        E temp = head.ele;
        head = head.next;
        if(head==null)
        {
                tail = null;
        }
        indx--;
        return temp;
}
public E removeLast()
{
        if(head == null)
        {
                throw new NoSuchElementException("User the linked list is empty");
        }
        if(head==tail)
        {
                return removeFirst();
        }
```

```
		Node<E> currentNode = head;
		for(int i=0;i<size()-2;i++)
		{
			currentNode = currentNode.next;
		}
		currentNode.next = null;
		E temp = tail.ele;
		tail = currentNode;
		indx--;
		return temp;
	}
	public boolean removeFirstOccurrence(E ele)
	{
		Node<E> currentNode = head;
		boolean flag = false;
		for(int i=0;i<size();i++)
		{
			if(ele.equals(currentNode.ele))
			{
				remove(i);
				flag = true;
				break;
			}
			currentNode = currentNode.next;
		}
		return flag;
	}
	public boolean removeLastOccurrence(E ele)
	{
		if(head==null)
		{
			return false;
		}
		Node<E> currentNode = head;
		boolean flag = false;
		int ind = 0;
		for(int i=0;i<size();i++)
		{
			if(ele.equals(currentNode.ele))
			{
				flag = true;
				ind = i;
			}
			currentNode = currentNode.next;
		}
		if(flag)
		{
			remove(ind);
		}
		return flag;
	}

	public E remove(int indx)
	{
		if(indx<0 || indx>=size())
		{
			throw new IndexOutOfBoundsException("User wrong index"+indx);
		}
		if(indx==0)
		{
```

```
                return removeFirst();
            }
            Node<E> currentNode1 = head;
            Node<E> currentNode2 = null;
            for(int i=0;i<indx-1;i++)
            {
                currentNode1 = currentNode1.next;
            }
            currentNode2 = currentNode1.next;
            currentNode1.next = currentNode2.next;
            if(currentNode2 == tail)
            {
                tail = currentNode1;
            }
            this.indx--;
            return currentNode2.ele;
        }
        public void add(int indx, E ele)
        {
            if(indx<0 || indx>=size())
            {
                throw new IndexOutOfBoundsException("User wrong index"+indx);
            }
            if(indx==0)
            {
                addFirst(ele);
                return;
            }
            Node<E> newNode = new Node<E>(ele);
            Node<E> currentNode = head;
            for(int i = 0;i<indx-1;i++)
            {
                currentNode = currentNode.next;
            }
            newNode.next = currentNode.next;
            currentNode.next = newNode;
            this.indx++;
        }
        public void addAll(Collection<E> coll)
        {
            for (E ele : coll)
            {
                addLast(ele);
            }
        }
}
```

1. **public String toString():-**
   - The toString() is used to display the LinkedList data elements.
   - The elements of LinkedList are enclosed within square brackets and separated by comma.
   - First it checks that the LinkedList contains elements or not based on the logic of **if(head==null)** ,
     If it is true then it returns empty square brackets.
   - toString method iterates a loop based on the current node has reference another node till to the last node of the LinkedList and concatenate the elements of LinkedList from the currentNode reference.
   - Lastly it concatenates the last node element into String and with the closed square bracket and returns the String.

2. **public void addLast(E ele):-**
   - The addLast() is used to add an element to the LinkedList at the last.
   - Firstly it creates a new node and assigned to the tail node.
   - If head node is null the newNode is assigned to the head node.
   - If not new current node is created based on head node and iterate the nodes till the last node.
   - The last node gets initialized with the new node and new node becomes the last node.
   - When the element is added to the LinkedList the counter variable is incremented by one.

3. **public void addFirst(E ele):-**
   - The addFirst() is used to add an element to the LinkedLIst at the first.
   - Firstly it creates a new node.
   - Then it checks that the head is already null or not, if yes new node is assigned to the head node.
   - If not, new node is assigned to the head node and previous head node reference is assigned to the new head node.
   - When the element is added to the LinkedList the counter variable is incremented by one.

4. **public int size():-**
   - The size() is used to get the number of elements present in the LinkedList.
   - It returns an integer value based on the index i.e. the index track the elements insertion and remove operation.
   - Index variable gets incremented when element is added, and gets decremented when element is removed.

5. **public E getFirst():-**
   - The getFirst() is used to get the first element present in LinkedList.
   - It returns the element present at the head node.

6. **public E getLast():-**
   - The getLast() is used to get the last element present in LinkedList.
   - It returns the element present at the tail node.

7. **public E removeFirst():-**
   - The removeFirst() is used to remove the element from LinkedList which is present in the head node.
   - Firstly it checks that LinkedList is empty or not if it is empty throws an exception that NoSuchElementException.
   - If not, the element which is present at the head node is stored in temporary variable.
   - Then head node is initialized to the next node,the temp variable is returned.
   - Because the method returns the removed variable.

8. **public E removeLast():-**
   - The removeLast() is used to remove the element from LinkedList which is present in the tail node.
   - Firstly it checks that LinkedList is empty or not if it is empty throws an exception that NoSuchElementException.
   - If not, head is assigned to current node and it is iterated till to the second last node.
   - The tail node element stored in the temporary variable for return purpose.
   - Then the second last node is assigned with null reference and initialized to tail node.
   - Lastly temp variable is returned.

9. **public E remove(int index):-**
   - The remove() is used to remove a node from the LinkedList based on the index.
   - Firsly it checks for the index is valid or not based on condition (index<0 || index>=size()), if the condition is true then IndexOutOfBoundsException is throw.
   - If not it checks for the is second condition that the index is equal to zero if yes then removeFirst() is called if not it skips the if block.
   - Then new current node 1 is initialized by head node.
   - And the current node is iterated till to the specified index-1 node.
   - Then current node 2 is created and current node one referenced is initialied to current node 2 and then current node one referenced gets the current node 2 reference.
   - Then it checks that currnet node 2 is tail or not if yes current node one is assigned to the tail node.
   - Counter variable of LinkedList is decremented by one.
   - And current node 2 element is returned.

10. **public boolean removeFirstOccurance(E ele):-**
    - The removeFirstOccurance() is used to remove the first occurance of an specified element from LinkedList.
    - Firstly it creates current node initialied by head node, and boolean flag initialized to false for return.
    - Iterate the loop on LinkedList based on the size of the LinkedList and checks each element that it is equals to the specified element using equals method.
    - If it is matched then the remove() method is called for removing the matched element index and flag initialized to true and break the loop.
    - Lastly returns the flag.

11. **public boolean removeLastOccurance(E ele):-**
    - The removeLastOccurance() is used to remove the last occurance of an specified element from the LinkedList.
    - Firstly it creates current node initialized by head node, and boolean flag initialized to false for return.
    - An temporary index variable is created initialized to zero for tracking the index of the element.
    - Iterate the loop on LinkedList elements and check the specified element present in the list or not if present the index variable is initialized to iteration number i.e. **i** and flag initialized to true**.**
    - It itrates through all the elements of the LinkedList and checks where is the last occurance of the specified element because the temporary variable gets reinitialized to iterative number i.e. **i** whenever the specified element is matched.
    - After the loop terminated it checks that flag is true then remove method is called and the temporary index variable is passed for removing the element.
    - Lastly flag is returned.

12. **public void addAll(Collection<E> coll):-**
    - The addAll() is used to add the elements specified in the collection at the last in the LinkedList.
    - for each loop is used to iterate Collection elements and addLast() is called and Collection element is passed to the addLast() as an argument.
    - All the elements specified in the collection are stored in LinkedList.

★ **Difference between ArrayList and LinkedList.**

| ArrayList | LinkedList |
|---|---|
| i. ArrayList internally uses a dynamic array to store the elements. | i. LinkedList internally uses a doubly linked list to store the elements. |
| ii. Manipulation in ArrayList is slow because it performs shifting operation. | ii. It is best used for removing and insertion of element in LinkedList. |
| iii. An ArrayList class can act as a list only because it implements List only. | iii. LinkedList class can act as a list and queue both because it implements List and Deque interfaces. |
| iv. ArrayList is better for storing and accessing data. | iv. LinkedList is better for manipulating data. |
| v. The memory location for the elements of an ArrayList is contiguous. | v. The location for the elements of a LinkedList is non contiguous. |
| vi. Generally, when an ArrayList is initialized a default capacity of 10 is assigned to the ArrayList. | vi. There is no case of default capacity in LinkedList. |
| vii. To be precise an ArrayList is a resizable array. | vii. LinkedList implements the doubly linked list of the list interface. |
| viii. ArrayList implements RandomAccess interface. | viii. LinkedList doesn't implements RandomAccess interface. |

★ **Difference between ArrayList and Vector.**

| ArrayList | Vector |
|---|---|
| i. ArrayList introduced in JDK 1.2 version. | i. Vector introduced in JDK 1.0 version. |
| ii. ArrayList is not a Legacy class. | ii. Vector is Legacy class. |
| iii. ArrayList is not synchronized. | iii. Vector is synchronized. |
| iv. ArrayList is faster compared to Vector. | iv. Vector is slower compared to ArrayList. |
| v. Multiple threads can work at a time. | v. Only one thread can work at a time. |
| vi. ArrayList is not thread safe. | vi. Vector is thread safe. |
| vii. When ArrayList is full and we insert a new element it grows based on the formula, **((initialCapacity*3)/2)+1** | vii. When Vector capacity is full and we insert a new element it grows based on the formula, **(initialCapacity*2)** |

# 3. Vector:-

- Vector is implemented class of List interface.
- It is present inside java.util package.
- It is introduced in JDK version 1.0
- It internally implements growable array data structure.
- It preserve insertion order.
- It allows you to store duplicate values as well as multiple null values.
- It is thread safe use for multithreading.
- It provide the implementation for the RandomAccess interface because of that it is faster for searching.
- It also provide implementation for other interfaces such as, Collection, Interable, Cloneable, Serializable.
- Vector is synchronized because of that we perform multithreading.
- Vector class is also known as Legacy class.
- It can store both heterogeneous and homogeneous type of data.
- Vector have a built-in capacity of 10.
- When Vector capacity is full and we insert a new element in it, it is doubled (it will be 20).
- Vector increase its capacity by formula **( initialCapacity*2)**.

### ★ Constructors of Vector class:-

i. public Vector()
ii. public Vector(int initialCapacity)
iii. public Vector(int initialCapacity, int capacityIncrease)
iv. public Vector(Collection c)

### i. public Vector():-

- It constructs an empty Vector with no elements in it and with capacity of 10.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Vector<Integer> v = new Vector<>();
            for(int i=1;i<=10;i++)
                    v.add(i);
            System.out.println(v.capacity());
            System.out.println(v);
    }
}
```

**Output:-**

```
10
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

### ii. public Vector(int initialCapcity):-

- It constructs an empty Vector with specified initial capacity.
- E.g.

```
import java.util.*;
class InitialCapacityConstructor
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Vector<Integer> v = new  Vector<>(20);
            for(int i=1;i<=20;i++)
                    v.add(i);
            System.out.println(v.capacity());
            System.out.println(v);
    }
}
```

**Output:-**

```
20
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

### iii. public Vector(int initialCapacity, int capacityIncrease):-

- It constructs an empty Vector with specified initial capacity and the increment capacity.

**Parameters:-**

i. initialCapacity:-
   - initialCapacity of Vector.

ii. capacityIncrease:-
   - The amount by which capacity increase when Vector overflows.

- E.g.

```
import java.util.*;
class CapacityIncreaseArgumentConstructor
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Vector<Integer> v = new Vector<>(10,20);
            System.out.println("Capacity of vector : "+v.capacity());
            for(int i=1;i<=11;i++)
                    v.add(i);
            System.out.println("Capacity of vector : "+v.capacity());
            System.out.println(v);
    }
}
```

**Output:-**

```
Capacity of vector : 10
Capacity of vector : 30
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

### iv. public Vector(Collection c):-

- It constructs a Vector containing elements of the specified collection.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructor
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Vector<Integer> v = new Vector<>();
            for(int i=1;i<=10;i++)
            {
                    v.add(i);
            }
            System.out.println(v);
            ArrayList<Integer> al = new ArrayList<>(v);
            System.out.println(al);
            LinkedList<Integer> ll = new LinkedList<>();
            for(int i=1;i<=10;i++)
                    ll.add(i);
            PriorityQueue p = new PriorityQueue();
            for(int i=1000;i<=10000;i+=1000)
                    p.add(i);

            Vector<Integer> v1 = new Vector<>(ll);
            Vector<Integer> v2 = new Vector<>(ll);
            Vector<Integer> v3 = new Vector<>(p);
            System.out.println(v1);
            System.out.println(v2);
            System.out.println(v3);

    }
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]
```

## ★ Methods of Vector class:-

i. public synchronized int capacity();
ii. public synchronized E elementAt(int index);
iii. public synchronized E firstElemet();
iv. public synchronized E lastElement();
v. public synchronized void setElementAt(E ele,int index);
vi. public synchronized void removeElementAt(E ele,int index);
vii. public synchronized void insertElementAt(E ele,int index);
viii. public Enumeration elements();
ix. public synchronized void trimToSize();
x. public synchronized void ensureCapacity(int newCapacity);
xi. public synchronized void addElement(E ele);
xii. public synchronized boolean removeElement(Object obj);

**Example on Methods:-**

```
import java.util.*;
class MethodExample
{
  @SuppressWarnings("unchecked")
  public static void main(String[] args)
  {
            Vector<Integer> v = new Vector<>();
            for(int i=1;i<=100;i+=10)
                        v.add(i);
            System.out.println(v.capacity());
            System.out.println(v.elementAt(2));
            System.out.println(v.firstElement());
            System.out.println(v.lastElement());
            v.setElementAt(20,0);
            System.out.println(v);
            v.removeElementAt(3);
            System.out.println(v);
            v.insertElementAt(900,7);
            System.out.println(v);
  }
}
```

**Output:-**

```
10
21
1
91
[20, 11, 21, 31, 41, 51, 61, 71, 81, 91]
[20, 11, 21, 41, 51, 61, 71, 81, 91]
[20, 11, 21, 41, 51, 61, 71, 900, 81, 91]
```

### i. public synchronized void addElement(E ele):-

- Adds the specified component to the end of the this Vector, increasing its size by one.

### ii. public synchronized boolean removeElement(Object obj):-

- Removes the first (lowest-indexed) occurrence of the argument from this Vector.

**Below addElement() and removeElement() example:-**

```
import java.util.Vector;
class AddAndRemoveElementExample{
    public static void main(String[] args){
            Vector<String> vector = new Vector<>();
            vector.addElement("A");
            vector.addElement("B");
            vector.addElement("C");
            vector.addElement("B");
            System.out.println("Original Vector : "+vector);
            boolean isRemoved = vector.removeElement("B");
            System.out.println("Vector after removeElement('B') : "+vector);
            System.out.println("Was 'B' removed ? "+isRemoved);
            isRemoved = vector.removeElement("Z");
            System.out.println("Vector after removeElement('Z') : "+vector);
            System.out.println("Was 'Z' removed ? "+isRemoved);
    }
}
```

**Output:-**

```
Original Vector : [A, B, C, B]
Vector after removeElement('B') : [A, C, B]
Was 'B' removed ? true
Vector after removeElement('Z') : [A, C, B]
Was 'Z' removed ? false
```

### iii. public synchronized E elementAt(int index):-

- Returns the component at the specified index.
- E.g.

```
import java.util.Vector;
class ElementAtExample{
    public static void main(String[] args) {
            Vector<String> vector = new Vector<>();
            vector.add("A");
            vector.add("B");
            vector.add("C");
            vector.add("D");
            System.out.println("Original Vector : "+vector);
            String element = vector.elementAt(2);
            System.out.println("Element at index 2 : "+element);
            try {
                    String invalidElement = vector.elementAt(5);
            }
            catch(ArrayIndexOutOfBoundsException e){
                    System.out.println("Error : index out of bounds.");
            }
    }
}
```

**Output:-**

```
Original Vector : [A, B, C, D]
Element at index 2 : C
Error : index out of bounds
```

### iv. public synchronized void ensureCapacity(int capacity):-

- Increases the capacity of vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

### v. public public synchronized int capacity():-

- Returns the current capacity of this Vector.

**Below the example of ensureCapacity() and capacity():-**

```
import java.util.Vector;
class CapacityAndEnsureCapcity
{
    public static void main(String[] args)
    {
            Vector<Integer> vector = new Vector<>(2);
            System.out.println("Initial capacity : "+vector.capacity());
            vector.add(1);
            vector.add(2);
            vector.add(3);
            System.out.println("Capacity after adding elements : "+vector.capacity());
            vector.ensureCapacity(10);
            System.out.println("Capacity after ensureCapacity() : "+vector.capacity());
            vector.add(4);
            vector.add(5);
            System.out.println("Capacity after adding more elements : "+vector.capacity());
    }
}
```

### vi. public synchronized E firstElement():-

- Returns the first component (the item at index 0) of this Vector.

### vii. public synchronized E lastElement():-

- Returns the last component of this vector.

**Below the example of firstElement() and lastElement():-**

```
import java.util.Vector;
class FirstAndLastElementExample
{
    public static void main(String[] args)
    {
            Vector<String> vector = new Vector<>();
            vector.add("A");
            vector.add("B");
            vector.add("C");
            vector.add("D");
            System.out.println("Original Vector : "+vector);
            String firstElement = vector.firstElement();
            System.out.println("First element : "+firstElement);
            String lastElement = vector.lastElement();
            System.out.println("Last element : "+lastElement);
            vector.clear();
            try
            {
                    firstElement = vector.firstElement();
            }
            catch(Exception e)
            {
                    System.out.println("Error : Vector is empty, cannot retrieve first element");
            }
            try
            {
                    lastElement = vector.lastElement();
            }
```

```
            catch(Exception e)
            {
                System.out.println("Error : Vector is empty, cannot retrieve last element");
            }
        }
    }
```

**Output:-**

```
Original Vector : [A, B, C, D]
First element : A
Last element : D
Error : Vector is empty, cannot retrieve first element
Error : Vector is empty, cannot retrieve last element
```

### viii. public synchronized void insertElementAt(E element, int index):-

- Inserts the specified object as a component in this Vector at the specified index.

### ix. public synchronized void setElementAt(E element, int index):-

- Sets the component at the specified index of this Vector to be the specified index.

**Below example of insertElementAt() and setElementAt():-**

```
import java.util.Vector;
class InsertElementAtAndSetElementAtExample
{
    public static void main(String[] args)
    {
        Vector<String> vector = new Vector<>();
        vector.add("A");
        vector.add("B");
        vector.add("C");
        vector.add("D");
        System.out.println("Originial Vector : "+vector);
        vector.insertElementAt("X",2);
        System.out.println("Vector after insertElementAt('X',2) : "+vector);
        vector.setElementAt("Y",1);
        System.out.println("Vector after setElementAt('Y',1) : "+vector);
        try
        {
            vector.setElementAt("Z",10);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Error : Index out of bounds for the setElementAt().");
        }

    }
}
```

**Output:-**

```
Originial Vector : [A, B, C, D]
Vector after insertElementAt('X',2) : [A, B, X, C, D]
Vector after setElementAt('Y',1) : [A, Y, X, C, D]
Error : Index out of bounds for the setElementAt().
```

# 4. Stack:-

- Java collection framework contains a Stack class that has stack data structure as an implementation.
- Stack is a subclass of Vector which was introduced in JDK 1.0 version and derived in java.util package.
- As it was introduced in 1.0 version it is one of the Legacy class.
- Stack follows a basic principle of LIFO and FILO.
- Stack can be implemented by two ways-
  i. ArrayList
  ii. LinkedList
- As it is an implementation class of List interface we can store multiple null elements and as well as duplicate elements.
- It maintains or preserve the insertion order.
- Initial capapcity of Stack is 10 and it gets double when Stack capacity is full.

### ★ Constructor of Stack:-

- It construct an empty Stack with initial capacity of 10.
- E.g.

```
import java.util.Stack;
class StackConstructorExample
{
  @SuppressWarnings("unchecked")
  public static void main(String[] args)
  {
        Stack obj = new Stack();
        obj.push(10);
        obj.push(20);
        obj.push(30);
        System.out.println(obj);
        Stack obj1 = new Stack();
        for(int i=10;i<=100;i+=10)
        {
              obj1.push(i);
        }
        System.out.println(obj1);
  }
}
```

**Output:-**

```
[10, 20, 30]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

### ★ Methods of Stack:-

i. public boolean empty();
ii. public synchronized E peek();
iii. public synchronized E pop();
iv. public E push(E ele);
v. public synchronized int search(Object obj);

1. **public boolean empty():-**
   - Checks if this Stack is empty and returns true if no element is 0 or else returns false.
   - E.g.

```
import java.util.*;
class EmptyExample
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            Stack<String> stack = new Stack<>();
            System.out.println("Is the Stack empty ? "+stack.empty());
            stack.push("A");
            stack.push("B");
            stack.push("C");
            System.out.println("Is the Stack empty after pushing elements ? "+stack.empty());
            stack.pop();
            System.out.println("Is the Stack empty after popping elements ? "+stack.empty());
    }
}
```

**Output:-**

```
Is the Stack empty ? true
Is the Stack empty after pushing elements ? false
Is the Stack empty after popping elements ? false
```

2. **public synchronized E peek():-**
   - Returns the object from the top of this Stack without removing it from the Stack.
3. **public synchronized E pop():-**
   - Removes the object from the top of this Stack and returns that object.

**Below the example of peek() and pop():-**

```
import java.util.*;
class PopAndPeekExample
{
    public static void main(String[] args)
    {
            Stack<String> stack = new Stack<>();
            stack.push("A");
            stack.push("B");
            stack.push("C");
            stack.push("D");
            System.out.println("Original Stack : "+stack);
            String topElement = stack.peek();
            System.out.println("Top element using peek : "+topElement);
            String poppedElement = stack.pop();
            System.out.println("Element pop : "+poppedElement);
            System.out.println("Stack after pop : "+stack);
            topElement = stack.peek();
            System.out.println("Top element after pop using peek() : "+topElement);
            while(!stack.isEmpty())
            {
                    System.out.println("Element popped : "+stack.pop());
            }
            try
            {
                    stack.peek();
            }
            catch(Exception e)
            {
                    System.out.println("Error : Stack is empty.");
```

```
            }
            try
            {
                stack.pop();
            }
            catch(Exception e)
            {
                System.out.println("Error : Stack is empty.");
            }
        }
    }
```

**Output:-**

```
Original Stack : [A, B, C, D]
Top element using peek : D
Element pop : D
Stack after pop : [A, B, C]
Top element after pop using peek() : C
Element popped : C
Element popped : B
Element popped : A
Error : Stack is empty.
Error : Stack is empty.
```

4. **public E push(E item):-**
   - Pushes an item onto the top of this Stack.

5. **public synchronized int search(Object obj):-**
   - Returns the 1-based position where an object is on this Stack.
   - Its consider the last element as the first element are returns its position from top to bottom order.

**Below example of push() and search():-**

```
import java.util.*;
class PushAndSearchExample
{
    public static void main(String[] args)
    {
        Stack<String> stack = new Stack<>();
        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        System.out.println("Stack after push operations : "+stack);
        int position = stack.search("B");
        if(position!=1)
        {
            System.out.println("Element 'B' found at position (from top) : "+position);
        }
        else
        {
            System.out.println("Element 'B' not found in the stack. ");
        }
        position = stack.search("Z");
        if(position!=1)
        {
            System.out.println("Element 'Z' found at position (from top) : "+position);
        }
        else
        {
```

```
                System.out.println("Element 'Z' not found in the stack. ");
            }
            stack.pop();
            stack.pop();
            System.out.println("Stack after popping two elements : "+stack);
            position = stack.search("B");
            if(position!=1)
            {
                System.out.println("Element 'B' found at position (from top) : "+position);
            }
            else
            {
                System.out.println("Element 'B' not found in the stack. ");
            }
    }
}
```

**Output:-**

```
Stack after push operations : [A, B, C, D]
Element 'B' found at position (from top) : 3
Element 'Z' found at position (from top) : -1
Stack after popping two elements : [A, B]
Element 'B' not found in the stack.
```

## -: Implementation of User created Stack :-

```
import java.util.*;
@SuppressWarnings("unchecked")
class UserStack<E>
{

	public static final int DEFAULT_CAPACITY = 10;
	E[] arr;
	int indx = 0;
	UserStack()
	{
		arr =(E[]) new Object[DEFAULT_CAPACITY];
	}

	@Override
	public String toString()
	{
		if(indx==0)
		{
			return "[]";
		}
		String op = "[";
		for(int i=0;i<size()-1;i++)
		{
			op += arr[i]+", ";
		}
		op += arr[size()-1]+"]";
		return op;
	}

	public int size()
	{
		return indx;
	}

	public E push(E ele)
	{
		if(arr.length <=size())
		{
			int newCap = arr.length*2;
			E[] newArr = (E[]) new Object[newCap];
			for(int i=0;i<size();i++)
			{
				newArr[i] = arr[i];
			}
			arr = newArr;
		}
		arr[indx++] = ele;
		return ele;
	}

	public int capacity()
	{
		return arr.length;
	}

	public E pop()
	{
		E temp = arr[size()-1];
```

```
                arr[size()-1] = null;
                indx--;
                return temp;
        }

        public int search(E ele)
        {
                for(int i=size()-1,offSet = 1;i>=0;i--,offSet++)
                {
                        if(ele.equals(arr[i]))
                                return offSet;
                }
                return -1;
        }

        public boolean empty()
        {
                if(size() == 0)
                        return true;
                return false;
        }
}
```

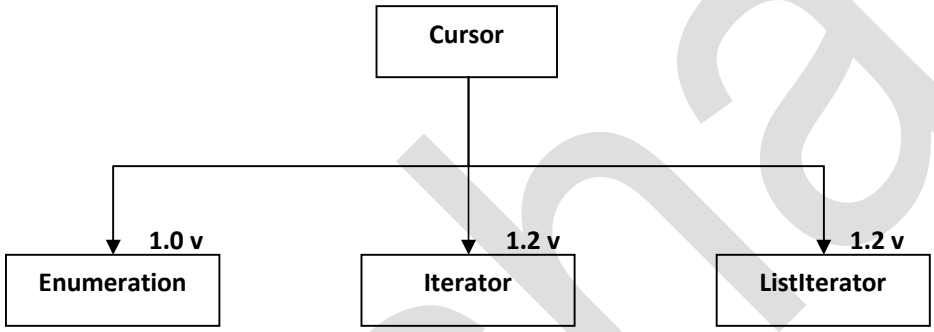
# ★ Cursor in Java :-

- A java cursor is an iterator which is used to iterate (traverse) over a collection, stream of objects one by one.

★ **Advantages of using cursors :-**

i. We can use it to iterate through elements of an collection or data sets without worry about the data structure.
ii. Cursors are designed to work with various data structures, Lists, Sets and Maps.
iii. It allows us to safely removes the elements from a collection, preventing ConcurrentModificationException.
iv. Cursor also allows us to traverse the collection in both forward and backword direction.

★ **Types of Cursors :-**




# 1. Enumeration:-

- Enumeration is Legacy interface in java.
- It was introduced in JDK 1.0 version and derived in java.util package.
- An obect might implements Enumeration interfaces generates series of elements one at a time.
- As it's a Legacy interface its only applicable for Vector and Stack.

**Disadvantages:-**

i. It cannot perform any operation other than retrieving of elements.
ii. It traverses the elements only in one direction.

★ **Methods of Enumeration:-**

**i. public abstract boolean hasMoreElements():-**
- This method checks if this Enumeration contains more elements in it and returns true if present or else returns false.

**ii. public abstract E nextElement():-**
- This method returns the next elements of this Enumeration and moves the cursor one element ahead.

★ **Example of Enumeration cursor:-**

```
import java.util.*;
class EnumerationCursor
{
    public static void main(String[] args)
    {
            Vector<Integer> list = new Vector<>();
            for(int i=10;i<=100;i+=10)
                    list.add(i);
            System.out.println(list);
            Enumeration<Integer> en = list.elements();
            while(en.hasMoreElements())
            {
                    Integer ele = en.nextElement();
                    System.out.println(ele);
            }
    }
}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
10
20
30
40
50
60
70
80
90
100
```

**Note:-**

We cannot remove elements using Enumeration to overcome this problem we have Iterator.

# 2. Iterator:-

- The functionality of this interface is duplicated by Enumeration.
- The difference is, Iterator takes the place of Enumeration in java's collection framework.
- Iterator differs from Enumeration in following ways,
  i. Iterator allows us to remove elements from underlying collection while traversing with a well defined implementation.
  ii. It has improved method names.
- Iterator is an universal cursor which can operate on any classes of collection framework.
- The iterator() is defined in Iterable interface (root of collection).
- There are two child classes of Iterator,
  i. ListIterator
  ii. EventIterator

★ **Methods of Iterator:-**

i. **public abstract boolean hasNext():-**
   - This method returns true if the iterator has more elements in it or else false.

ii. **public abstract E next():-**
   - This method returns the next element from this iterator.

iii. **public default void remove():-**
   - It remove the element from the underlying collection.
   - The method can be called only once per call to the next().
   - If we try to call it more than once the method throws exceptions such as UnsupportedOperationException, IllegalStateException.

★ **Example of Iterator cursor:-**

```
import java.util.*;
class IteratorCursorExample{
  public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for(int i=1;i<=10;i++)
              list.add(i);
        System.out.println(list);
        Iterator<Integer> it = list.iterator();
        while(it.hasNext())
              if(!(isPrime(it.next())))
                    it.remove();
        System.out.println(list);
  }
  public static boolean isPrime(int num){
        if(num<2)
              return false;
        for(int i=2;i<num;i++)
              if(num%i==0)
                    return false;
        return true;
  }
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 3, 5, 7]
```

**Note:-**

We can only perform remove operation and can traverse in one direction using Iterator, so to overcome this we have ListIterator.

# 3. ListIterator:-

- ListIterator is the child interface of Iterator.
- ListIterator is not a universal cursor as it only operates on List implementation classes.
- This Iterator allows the programmer to traverse the list in both directions i.e forward and backword.
- With the help of this cursor we can perform several operations such as insertion of an element, removing of an element, updating an element and fetch an element.
- In simple words we can perform CRUD (Create Read Update Delete) operations.

## ★ Methods of ListIterator:-

1. **public abstract boolean hasNext():-**
   - This method returns true if this Iterator has more elements while traversing this list in forward direction or else returns false.

2. **public abstract E next():-**
   - It returns the next element from this list and moves the cursors position ahead.

3. **public abstract boolean hasPrevious():-**
   - This method returns true if this Iterator has more elements while traversing this list in backword direction or else returns false.

4. **public abstract E previous():-**
   - This method returns the previous element from this list and moves the cursor position backword.

5. **public abstract void remove():-**
   - This method removes the last element that was returned by the next() or previous() (based on the operation).
   - This call to remove() can be made only if remove() or add() have been called after the last call to next() or previous().

6. **public abstract void set(E ele):-**
   - It replaces the element returned by next() or previous() with the specified element.

7. **public abstract void add(E ele):-**
   - It inserts the specified element into this list.

**Example:-**

```
import java.util.*;
class ListIteratorExample
{
  public static void main(String[] args)
  {
        ArrayList<Integer> list = new ArrayList<>();
        for(int i=10;i<=100;i+=10)
              list.add(i);
        System.out.println(list);
        ListIterator<Integer> li = list.listIterator();
        while(li.hasPrevious())                    //this while will be skipped
        {
              System.out.println(li.previous());
        }
        while(li.hasNext())
        {
              Integer currEle = li.next();
              System.out.println(currEle);
        }
```

```
            while(li.hasPrevious())
            {
                  Integer currEle = li.previous();
                  System.out.println(currEle);
            }
  }
}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
10
20
30
40
50
60
70
80
90
100
100
90
80
70
60
50
40
30
20
10
```

**Example of ListIterator:-**

```
import java.util.*;
class Mobile
{
	long imeiNo;
	String brand;
	String model;
	String processor;
	double price;
	public Mobile(long imeiNo,String brand,String model,String processor,double price)
	{
		this.imeiNo = imeiNo;
		this.brand = brand;
		this.model = model;
		this.processor = processor;
		this.price = price;
	}
	@Override
	public String toString()
	{
		return "Mobile [imeiNo="+imeiNo+", brand="+brand+", model="+model+
		                                                    ",                processor="+processor+",
		                                        price="+price+"]";
	}
}
class ListIteratorExample2
{
	public static void main(String[] args)
	{
		ArrayList<Mobile> list = new ArrayList<>();
		list.add(new Mobile(1234567l,"SAMSUNG","S24","Qualcomm",150000));
		list.add(new Mobile(2234567l,"Samsung","S21","Exynos",100000));
		list.add(new Mobile(3234567l,"OnePlus","Nord","SnapDragon",20000));
		list.add(new Mobile(4234567l,"APPLE","16pro","A17 bionic",150000));
		list.add(new Mobile(5234567l,"SAMSUNG","16pro","Qualcomm",80000));
		list.add(new Mobile(6234567l,"Google","pixel9","Silicon",42000));
		list.add(new Mobile(7234567l,"RED MAGIC","9pro","Qualcomm",200000));
		list.add(new Mobile(7234567l,"NOKIA","3310","Silicon",2500));
		ListIterator<Mobile> li = list.listIterator();
		//fetch and filter
		while(li.hasNext())
		{
			Mobile mobile = li.next();
			if(mobile.brand.equalsIgnoreCase("SAMSUNG"))
				System.out.println(mobile);
		}
		//update
		while(li.hasPrevious())
		{
			Mobile mobile = li.previous();

	if((mobile.brand.equalsIgnoreCase("APPLE"))&&(mobile.model.equalsIgnoreCase("16pro")))
				{
					System.out.println("Old price : "+mobile.price+" rs");
					mobile.price = 120000;
					li.set(mobile);
					System.out.println("New price : "+mobile.price+" rs");
				}
		}
		//show
```

```
		while(li.hasNext())
		{
			System.out.println(li.next());
		}
		//fetch, filter and remove
		while(li.hasPrevious())
		{
			if(li.previous().price<50000)
				li.remove();
		}
		System.out.println();
		//fetch or view
		while(li.hasNext())
		{
			System.out.println(li.next());
		}
	}
}
```

**Output:-**

```
Mobile [imeiNo=1234567, brand=SAMSUNG, model=S24, processor=Qualcomm, price=150000.0]
Mobile [imeiNo=2234567, brand=Samsung, model=S21, processor=Exynos, price=100000.0]
Mobile [imeiNo=5234567, brand=SAMSUNG, model=16pro, processor=Qualcomm, price=80000.0]
Old price : 150000.0 rs
New price : 120000.0 rs
Mobile [imeiNo=1234567, brand=SAMSUNG, model=S24, processor=Qualcomm, price=150000.0]
Mobile [imeiNo=2234567, brand=Samsung, model=S21, processor=Exynos, price=100000.0]
Mobile [imeiNo=3234567, brand=OnePlus, model=Nord, processor=SnapDragon, price=20000.0]
Mobile [imeiNo=4234567, brand=APPLE, model=16pro, processor=A17 bionic, price=120000.0]
Mobile [imeiNo=5234567, brand=SAMSUNG, model=16pro, processor=Qualcomm, price=80000.0]
Mobile [imeiNo=6234567, brand=Google, model=pixel9, processor=Silicon, price=42000.0]
Mobile [imeiNo=7234567, brand=RED MAGIC, model=9pro, processor=Qualcomm, price=200000.0]
Mobile [imeiNo=7234567, brand=NOKIA, model=3310, processor=Silicon, price=2500.0]

Mobile [imeiNo=1234567, brand=SAMSUNG, model=S24, processor=Qualcomm, price=150000.0]
Mobile [imeiNo=2234567, brand=Samsung, model=S21, processor=Exynos, price=100000.0]
Mobile [imeiNo=4234567, brand=APPLE, model=16pro, processor=A17 bionic, price=120000.0]
Mobile [imeiNo=5234567, brand=SAMSUNG, model=16pro, processor=Qualcomm, price=80000.0]
Mobile [imeiNo=7234567, brand=RED MAGIC, model=9pro, processor=Qualcomm, price=200000.0]
```

# ★ Collections class:-

- Collections class in java is part of java.util package.
- It provides static utility methods for working with collection framework.
- The methods of this class all throw a NullPointerException if the Collections or objects provided to them are null.

### ★ Methods of Collections class:-

i. public static int frequency(Collection c, Object frequencyObj)
ii. public static Comparator<T> reverseOrder()
iii. public static T max(Collection c)
iv. public static T min(Collection c)
v. public static void rotate(List list,int noOfRotation)
vi. public static void fill(List list, T type)
vii. public static void shuffle(List list)
viii. public static void swap(List list, int indexEle1, int indexEle2)
ix. public static void sort(List list, Comparator com)
x. public static Comparabale<T> void sort(List list)

### 1. public static Comparable <T> void sort(List list):-

- This method sorts the specified list into ascending order according to the natural ordering of elements.
- All the elements in this list must implement the Comparable interface.
- e.g.

```
import java.util.*;
class ComparableExample
{
    public static void main(String[] args)
    {
            List<Integer> list = Arrays.asList(9,5,6,3,4,2,1,8,7);
            System.out.println(list);
            Collections.sort(list);
            System.out.println(list);
    }
}
```

**Output:-**

```
[9, 5, 6, 3, 4, 2, 1, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

**Note:-**

- Collections.sort() is only applicable for the List implementation classes or List type objects and the data stored in this list i.e. object must implement Comparable interface.
- We cannot sort custom objects (User defined non-primitive datatype).
- If we want to sort custom objects the class must implements Comparable interface and we need to override compareTo().

### i. public staic int frequency(Collection c, Object obj) :-

- This method returns the number of elements in the specified collection equal to the specified object.
- E.g.

**Finding highest frequency of an element**

```
import java.util.*;
class FrequencyMethodExample1
{
    public static void main(String[] args)
    {
            int[] arr = {5,4,6,2,5,6,2,7,3,7,3,8,9,3};
            System.out.println(Arrays.toString(arr));
            ArrayList<Integer> list = new ArrayList<Integer>();
            for (int i : arr)
            {
                    list.add(i);
            }
            System.out.println(list);
            //Using TreeSet for storing only distinct element
            TreeSet<Integer> set = new TreeSet<Integer>(list);
            Integer ele = 0;
            int max = 0;
            for (Integer integer : set)
            {
                    Integer freCurr = Collections.frequency(list,integer);
                    if(max<freCurr)
                    {
                            max = freCurr;
                            ele = integer;
                    }
            }
            System.out.println(ele +" : "+max);
    }
}
```

**Output:-**

```
[5, 4, 6, 2, 5, 6, 2, 7, 3, 7, 3, 8, 9, 3]
[5, 4, 6, 2, 5, 6, 2, 7, 3, 7, 3, 8, 9, 3]
3 : 3
```

★ **WAJP to find frequency of elements from an array using stream API.**

```
import java.util.*;
class FrequencyFindStream
{
    public static void main(String[] args)
    {
            int[] arr = {5,4,6,2,5,6,2,7,3,7,3,8,9,3};
            System.out.println(Arrays.toString(arr));
            int[] distArr = Arrays.stream(arr).distinct().toArray();
            System.out.println(Arrays.toString(distArr));
            for(int i : distArr)
            {
                    int cnt = 0;
                    for(int j : arr)
                    {
                            if(i==j)
                                    cnt++;
                    }
                    System.out.println(i+" : "+cnt);
            }
    }
}
```

**Output:-**

```
[5, 4, 6, 2, 5, 6, 2, 7, 3, 7, 3, 8, 9, 3]
[5, 4, 6, 2, 7, 3, 8, 9]
5 : 2
4 : 1
6 : 2
2 : 2
7 : 2
3 : 3
8 : 1
9 : 1
```

### ii. public static void fill(List list, T obj) :-

- This method replaces all the elements of this specified list with the specified element.
- E.g

```
import java.util.*;
class FillMethodExample1
{
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
    {
            List list = new ArrayList();
            list.add(10);
            list.add("Hello");
            list.add(true);
            list.add('A');
            System.out.println(list);
            Collections.fill(list,"Akshay");
            System.out.println(list);
    }
}
```

**Output:-**

```
[10, Hello, true, A]
[Akshay, Akshay, Akshay, Akshay]
```

### iii. public static T max(Collection c):-

- This method returns the maximum element of the given Collection according to the natural ordering of its element.
- E.g.

```
import java.util.*;
class MaxMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<Integer>();
            for(int i=1;i<=10;i++)
            {
                    list.add(randomNumber());
            }
            System.out.println(list);
            System.out.println(Collections.max(list));
    }
    public static Integer randomNumber()
    {
            for (; ; )
            {
                    Integer num = (int)(Math.random()*100);
                    if(num>9)
                    {
                            return num;
                    }
            }
    }
}
```

**Output:-**

```
[32, 96, 95, 78, 24, 51, 43, 74, 75, 88]
96
```

### iv. public static T min(Collection c):-

- This method returns the minimum element from this given Collection to the natural ordering of its element.
- E.g.

```
import java.util.*;
class MinMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<Integer>();
            for(int i=1;i<=10;i++)
            {
                    list.add(randomNumber());
            }
            System.out.println(list);
            System.out.println(Collections.min(list));
    }
    public static Integer randomNumber()
    {
            for (; ; )
            {
                    Integer num = (int)(Math.random()*100);
                    if(num>9)
                    {
                            return num;
                    }
            }
    }
}
```

**Output:-**

```
[12, 39, 32, 80, 27, 22, 84, 74, 71, 75]
12
```

### v. public static void reverse(List list):-

- It reverses the order of elements in the specified list.
- E.g.

```
import java.util.*;
class ReverseMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<Integer>();
            for(int i=1;i<=10;i++)
            {
                    list.add(i);
            }
            System.out.println(list);
            Collections.reverse(list);
            System.out.println(list);
    }
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

### vi. public static void rotate(List list, int numOfRotation):-

- This method rotates the elements in the specified list by specified number of rotations.
- Positive integer will rotate it clockwise.
- Negative integer will rotate it anti-clockwise.
- E.g.

```
import java.util.*;
class RotateMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<Integer>();
            for(int i=1;i<=10;i++)
            {
                    list.add(i);
            }
            System.out.println(list);
            Collections.rotate(list,2);
            System.out.println(list);
            Collections.rotate(list,-1);
            System.out.println(list);
    }
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[9, 10, 1, 2, 3, 4, 5, 6, 7, 8]
[10, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### vii. public static void shuffle(List list):-

- It randomly permutes the list using the specified source of random list.
- E.g.

```
import java.util.*;
class ShuffleMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<Integer>();
            for(int i=1;i<=10;i++)
            {
                    list.add(i);
            }
            System.out.println(list);
            Collections.shuffle(list);
            System.out.println(list);
    }
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[6, 9, 2, 3, 7, 1, 8, 5, 4, 10]
```

### viii. public static void swap(List list, int i, int j):-

- This method swaps the elements at the specified position from this list.
- E.g.

```
import java.util.*;
class SwapMethodExample
{
    public static void main(String[] args)
    {
            ArrayList<Integer> list = new ArrayList<Integer>();
            for(int i=1;i<=10;i++)
            {
                    list.add(i);
            }
            System.out.println(list);
            Collections.swap(list,0,5);
            System.out.println(list);
    }
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[6, 2, 3, 4, 5, 1, 7, 8, 9, 10]
```

## ★ Comparable interface:-

- Comparable interface was introduced in JDK 1.2 version and derived in java.lang package.
- Comparable interface is used for natural ordering of elements (i.e ascending order).
- Collections.sort() internally uses Comparable interface for sorting.
- Comparable is a functional interface i.e. it contains only a single abstract method which is compareTo().
- **Steps to use Comparable interface:-**
  i. A class must implements Comparable interface.
  ii. We need to override the compareTo().
- Comparable can perform single sorting sequence at a time i.e. we can sort the elements based on the single data members.

## ★ public abstract int compareTo(T obj):-

- This method is used to compare the current object (this) with the specified object and returns soma value based on the condition.
- This method returns,
  i. **Positive integer,** if the current object data member is greater than the specified object data member.
  ii. **Negative integer,** if the current object data member is less than the specified object data member.
  iii. **Zero,** if the current object data member is same as specified object data member.
- **e.g.**

```
import java.util.*;
class Employee implements Comparable<Employee>
{
    int empid;
    String ename;
    double salary;
    Employee(int empid,String ename, double salary)
    {
            super();
            this.empid = empid;
            this.ename = ename;
            this.salary = salary;
    }
    @Override
    public String toString()
    {
            return "empid = "+empid+", ename = "+ename+", salary = "+salary;
    }

    @Override
    public int compareTo(Employee emp2)
    {
            if(this.salary<emp2.salary)
                    return -1;
            else if(this.salary>emp2.salary)
                    return 1;
            else
                    return 0;
    }
}
class Example1
{
    public static void main(String[] args)
    {
            ArrayList<Employee> list = new ArrayList<Employee>();
            list.add(new Employee(4,"RAMESH",30000));
            list.add(new Employee(3,"SURESH",40000));
            list.add(new Employee(1,"MUKESH",45000));
```

```
            list.add(new Employee(7,"GANESH",47000));
            list.add(new Employee(5,"MAHESH",42000));
            list.add(new Employee(2,"RAJESH",35000));
            list.add(new Employee(6,"RAMESH",57000));

            Collections.sort(list);
            ListIterator li = list.listIterator();
            while(li.hasNext())
            {
                  Employee emp = (Employee) li.next();
                  System.out.println(emp);
            }
      }
}
```

**Output:-**

```
empid = 4, ename = RAMESH, salary = 30000.0
empid = 2, ename = RAJESH, salary = 35000.0
empid = 3, ename = SURESH, salary = 40000.0
empid = 5, ename = MAHESH, salary = 42000.0
empid = 1, ename = MUKESH, salary = 45000.0
empid = 7, ename = GANESH, salary = 47000.0
empid = 6, ename = RAMESH, salary = 57000.0
```

## ★ Comparator interface:-

- A Comparator interface in java is used to order the elements based on user defined implementation.
- It is oftenly used when we need to sort objects in way i.e. different from the natural ordering (Comparable).
- Comparator is introduced in JDK 1.2 version and derived in java.util package.
- We can use it for multiple sorting of data members at a time.
- **Steps to use Comparator interface :-**
  i. Define a class for sorting, implement it with Comparator interface.
  ii. Override the compare().
  iii. Invoke Collections.sort() and pass first argument as list and second argument as instance of Comparator type.

## ★ public abstract int compare(T obj1,T obj2) :-

- compare() compares both the specified argument and returns a value based on a condition.
- It returns,
  i. **Positive Integer,** if object one data members is greater than the object two's data member.
  ii. **Negative Integer,** if object one data members is smaller than the object two's data member.
  iii. **Zero,** if both objects data members are same.
- **e.g.**

```
import java.util.*;
class Student
{
    int sid;
    String sname;
    long contact;
    String graduation;
    String branch;
    int yop;
    double cgpa;
    public Student(int sid,String sname,long contact,String graduation,
                                                    String branch,int yop,double cgpa)
    {
            super();
            this.sid = sid;
            this.sname = sname;
            this.contact = contact;
            this.graduation = graduation;
            this.branch = branch;
            this.yop = yop;
            this.cgpa = cgpa;
    }
    @Override
    public String toString()
    {
            return "sid = "+sid+", sname = "+sname+", contact = "+contact+
                    ", graduation = "+graduation+", branch = "+branch+", yop = "
                    +yop+", cgpa = "+cgpa;
    }
}
// sort by id ascending
class SortById implements Comparator<Student>
{
    @Override
    public int compare(Student s1,Student s2)
    {
            if(s1.sid>s2.sid)
                    return 1;
            else if(s1.sid<s2.sid)
                    return -1;
```

```
            else
                return 0;
        }
    }
    class Example2Comparator
    {
        public static void main(String[] args)
        {
            ArrayList<Student> list = new ArrayList<Student>();
            list.add(new Student(3,"RAMESH",9876543210l,"BE","CIVIL",2024,9.7));
            list.add(new Student(4,"SURESH",8876543210l,"BE","MECHANICAL",2024,8.7));
            list.add(new Student(2,"GANESH",7876543210l,"BE","ELECTRICAL",2023,7.9));
            list.add(new Student(6,"RAJESH",6876543210l,"BE","MECHANICAL",2024,9.1));
            list.add(new Student(5,"ANIMESH",5876543210l,"BE","CIVIL",2022,9.3));
            list.add(new Student(1,"MUKESH",4876543210l,"BE","ELECTRONICS",2023,8.7));
            list.add(new Student(7,"KAMLESH",3876543210l,"BE","ELECTRICAL",2023,6.7));
            Collections.sort(list,new SortById());
            ListIterator<Student> li = list.listIterator();
            while(li.hasNext())
            {
                System.out.println(li.next());
            }
        }
    }
```

**Output:-**

sid = 1, sname = MUKESH, contact = 4876543210, graduation = BE, branch = ELECTRONICS, yop = 2023, cgpa = 8.7
sid = 2, sname = GANESH, contact = 7876543210, graduation = BE, branch = ELECTRICAL, yop = 2023, cgpa = 7.9
sid = 3, sname = RAMESH, contact = 9876543210, graduation = BE, branch = CIVIL, yop = 2024, cgpa = 9.7
sid = 4, sname = SURESH, contact = 8876543210, graduation = BE, branch = MECHANICAL, yop = 2024, cgpa = 8.7
sid = 5, sname = ANIMESH, contact = 5876543210, graduation = BE, branch = CIVIL, yop = 2022, cgpa = 9.3
sid = 6, sname = RAJESH, contact = 6876543210, graduation = BE, branch = MECHANICAL, yop = 2024, cgpa = 9.1
sid = 7, sname = KAMLESH, contact = 3876543210, graduation = BE, branch = ELECTRICAL, yop = 2023, cgpa = 6.7

## ★ Difference between Comparable and Comparator

| Comparable | Comparator |
|---|---|
| i. It is used to sort the object with natural order (ascending). | i. It is used to sort the object based on custom order(ascending or descending). |
| ii. Derived in java.lang package. | ii. Derived in java.util package. |
| iii. Comparable is used for both built-in classes as well as user defined classes. | iii. Comparator is used for user defined classes. |
| iv. Comparable can perform single sorting of attribute at a time. | iv. Comparator can perform multiple sorting of attribute at a time. |
| v. Comparable contains an abstract method i.e. compareTo(). | v. Comparator contains two abstract methods i.e. compare() and equals(). |

**Assignment:-**

Comparator sorting program for student object

i. Sort by id (ascending)
ii. Sort by name (ascending)
iii. Sort by branch (descending)
iv. Sort by yop (descending)
v. Sort by cgpa (descending)

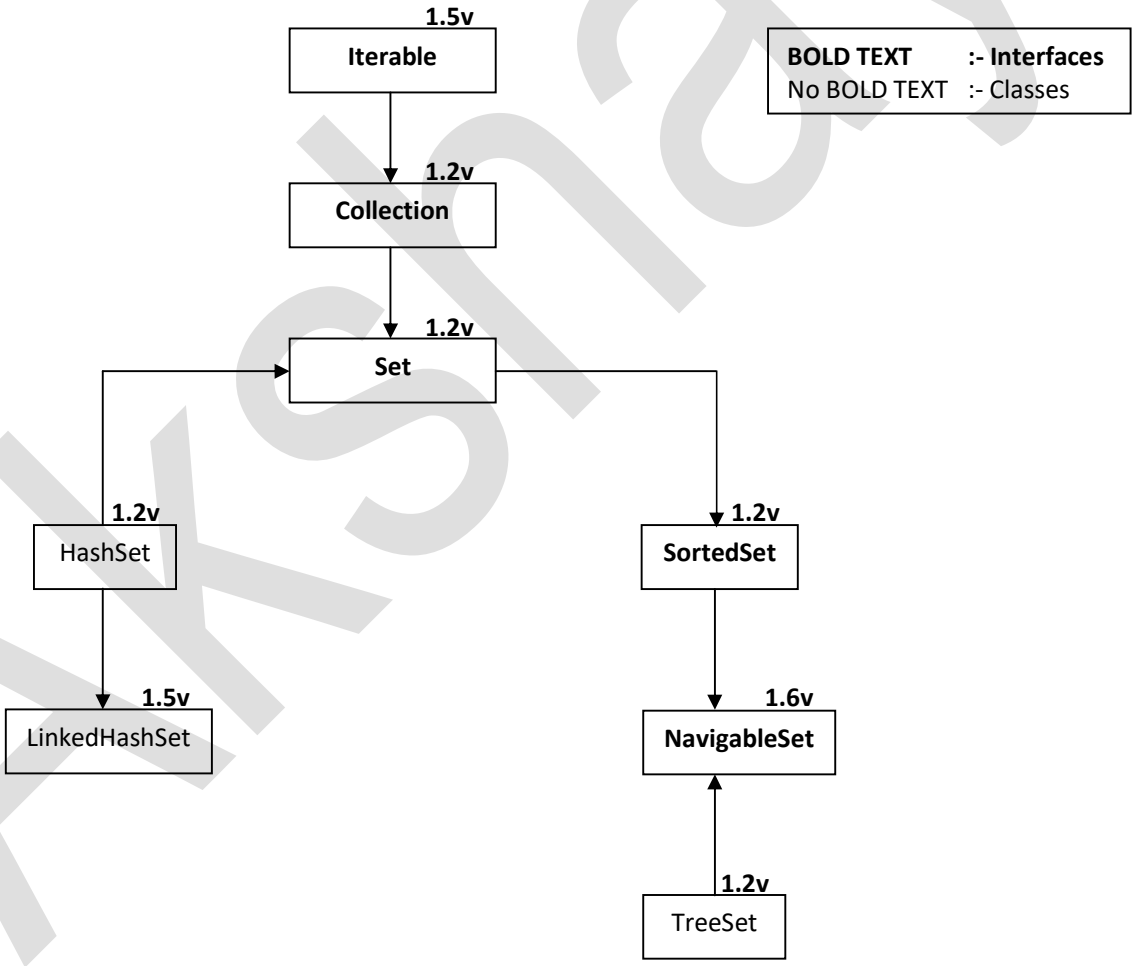
```
import java.util.*;
class Student
{
  int sid;
  String sname;
  long contact;
  String graduation;
  String branch;
  int yop;
  double cgpa;
  public Student(int sid,String sname,long contact,String graduation,String branch,
                                                                    int yop,double cgpa)
  {
        super();
        this.sid = sid;
        this.sname = sname;
        this.contact = contact;
        this.graduation = graduation;
        this.branch = branch;
        this.yop = yop;
        this.cgpa = cgpa;
  }

  @Override
  public String toString()
  {
        return "sid = "+sid+", sname = "+sname+", contact = "+contact+
              ", graduation = "+graduation+", branch = "+branch+", yop = "+yop+", cgpa = "+cgpa;
  }

}
//sorting student collection by id in ascending
class SortById implements Comparator<Student>
{
  @Override
  public int compare(Student s1,Student s2)
  {
        if(s1.sid>s2.sid)
              return 1;
        else if(s1.sid<s2.sid)
              return -1;
        else
              return 0;
        //another logic for sorting
        // return s1.sid-s2.sid;
  }
}
//sorting student collection by name in ascending
class SortByName implements Comparator<Student>
{
  @Override
  public int compare(Student s1,Student s2)
```

```
  {
        if(s1.sname.compareTo(s2.sname)>0)
              return 1;
        else if(s1.sname.compareTo(s2.sname)<0)
              return -1;
        else
              return 0;
        //another logic for sorting
        // return s1.sname.compareTo(s2.sname);
  }
}

//sorting student collection by branch in descending
class SortByBranch implements Comparator<Student>
{
  @Override
  public int compare(Student s1,Student s2)
  {
        if(s1.branch.compareTo(s2.branch)>0)
              return -1;
        else if(s1.branch.compareTo(s2.branch)<0)
              return 1;
        else
              return 0;
        //another logic for sorting
        // return s2.branch.compareTo(s1.branch);
  }
}

//sorting student collection by yop in descending order
class SortByYop implements Comparator<Student>
{
  @Override
  public int compare(Student s1,Student s2)
  {
        if(s1.yop>s2.yop)
              return -1;
        else if(s1.yop<s2.yop)
              return 0;
        else
              return 0;
        //another logic for sorting
        // return s2.yop-s1.yop;
  }
}

//sorting student collection by cgpa in descending
class SortByCgpa implements Comparator<Student>
{
  @Override
  public int compare(Student s1,Student s2)
  {
        if(s1.cgpa>s2.cgpa)
              return -1;
        else if(s1.cgpa>s2.cgpa)
              return 1;
        else
              return 0;
  }
}
```

```
class ComparatorExample
{
  public static void main(String[] args)
  {
        ArrayList<Student> list = new ArrayList<Student>();
        list.add(new Student(3,"RAMESH",9876543210l,"BE","CIVIL",2024,9.7));
        list.add(new Student(4,"SURESH",8876543210l,"BE","MECHANICAL",2024,8.7));
        list.add(new Student(2,"GANESH",7876543210l,"BE","ELECTRICAL",2023,7.9));
        list.add(new Student(6,"RAJESH",6876543210l,"BE","MECHANICAL",2024,9.1));
        list.add(new Student(5,"ANIMESH",5876543210l,"BE","CIVIL",2022,9.3));
        list.add(new Student(1,"MUKESH",4876543210l,"BE","ELECTRONICS",2023,8.7));
        list.add(new Student(7,"KAMLESH",3876543210l,"BE","ELECTRICAL",2023,6.7));

        //sort by id using custom class
        // Collections.sort(list,new SortById());

        //sort by id using lambda expression
        // Collections.sort(list,(s1,s2)->s1.sid-s2.sid);

        // sort by name using custom class
        // Collections.sort(list,new SortByName());

        //sort by name using lambda expression
        // Collections.sort(list,(s1,s2)->s1.sname.compareTo(s2.sname));

        //sort by branch using custom class
        // Collections.sort(list,new SortByBranch());

        //sort by branch using lambda expression
        // Collections.sort(list,(s1,s2)->s2.branch.compareTo(s1.branch));

        //sort by yop using custom class
        // Collections.sort(list,new SortByYop());

        //sort by yop using lambda expression
        // Collections.sort(list,(s1,s2)->s2.yop-s1.yop);

        //sort by cgpa using custom class
        // Collections.sort(list,new SortByCgpa());

        ListIterator<Student> li = list.listIterator();
        while(li.hasNext())
        {
              System.out.println(li.next());
        }
  }
}
```

# ★ Set interface:-

- Set is a sub interface of Collection interface.
- It is an unordered collection of objects (i.e. does not maintains the insertion order).
- We cannot store any duplicate elements in it.
- And also we cannot store multiple null values.
- This interface contains the methods inherited from Collection interface and adds a functionality that restricts the insertion of duplicate elements.
- It was introduced in JDK 1.2 version and derived in java.util package.
- There are two interfaces that extends Set interface,
  i. SortedSet
  ii. NavigableSet
- And it has some implementation classes such as,
  i. HashSet
  ii. LinkedHashSet
  iii. TreeSet




**Classification of Set interface**

## 1. HashSet:-

- HashSet is an implementation class of Set interface.
- It was introduced in JDK 1.2 version and derived in java.util package.
- HashSet is not synchronized.
- The internal implementation of HashSet is HashMap.
- So whenever we create a HashSet object, one HashMap object is associated with it, and it is created internally.
- The elements which we add into the HashSet is stored as a key inside the HashMap object.
- And the value associated with those keys is will be a constant **"PRESENT"**.
- We cannot store duplicate keys but we can store duplicate values.
- HashSet provides no gurantee of ordering the elements (it means does not maintains the insertion order).
- We cannot store duplicate elements and also one null value is allowed.
- HashSet offers constant time performance for basic operations like, **add, remove, contains(search) and size.**
- Every Hash implementation classes in java contains Default Load Factor i.e. 0.75 (HashSet, LinkedHashSet, HashMap, LinkedHashMap).
- HashSet have a default capacity of 16, which will be doubled once the object reaches its threshold value (load factor).
- HashSet has child class known as LinkedHashSet.
- HashSet also implements other several interfaces such as,
  i. Collection
  ii. Iterable
  iii. Serializable
  iv. Cloneable

★ **Constructors of HashSet:-**

i. public HashSet()
ii. public HashSet(Collection c)
iii. public HashSet(int initialCapacity)
iv. public HashSet(int initialCapacity, float loadFactor)

### i. public HashSet():-

- It creates an empty set with an initial capacity of 16 and default load factor 0.75.
- e.g.

```
import java.util.*;
class NoArgumentConstructor
{
    public static void main(String[] args)
    {
        HashSet<Integer> set = new HashSet<Integer>();
        System.out.println(set.add(10));
        System.out.println(set.add(20));
        System.out.println(set.add(40));
        System.out.println(set.add(30));
        System.out.println(set.add(10));
        System.out.println(set);
    }
}
```

**Output:-**

```
true
true
true
true
false
[20, 40, 10, 30]
```

### ii. public HashSet(Collection c):-

- It creates a set with a specified collection in it.
- The HashMap which is created will have a default load factor 0.75 and the capacity will be equals to the number of elements inside this specified collection.
- It throw NullPointerException if the specified collection is null.
- e.g.

```
import java.util.*;
class CollectionArgumentConstructor
{
	public static void main(String[] args)
	{
		ArrayList<Integer> list = new ArrayList<>();
		list.add(null);
		list.add(20);
		list.add(10);
		list.add(30);
		list.add(null);
		list.add(10);
		System.out.println(list);
		HashSet<Integer> set = new HashSet<>(list);
		System.out.println(set);
	}
}
```

**Output:-**

```
[null, 20, 10, 30, null, 10]
[null, 20, 10, 30]
```

### iii. public HashSet(int initialCapacity):-

- It creates an empty set with the specified initial capacity and a default load factor of 0.75.
- e.g.

```
import java.util.*;
class InitialCapacityArgumentConstructor
{
	public static void main(String[] args)
	{
		HashSet<Integer> set = new HashSet<Integer>(2);
		set.add(10);
		set.add(90);
		set.add(50);
		set.add(70);
		set.add(80);
		set.add(40);
		System.out.println(set);
	}
}
```

**Output:-**

```
[80, 40, 10, 90, 50, 70]
```

### iv. public HashSet(int initialCapacity, float loadFactor):-

- It creates an empty set with an specified initial capacity and default load factor.
- E.g.

```
import java.util.*;
class LoadFactorConstructor
{
	public static void main(String[] args)
	{
		HashSet<Integer> set = new HashSet<Integer>(10,0.6f);
		set.add(10);
		set.add(20);
		set.add(30);
		set.add(40);
		set.add(50);    //10
		set.add(60);    //10 -> 20
		System.out.println(set);
	}
}
```

**Output:-**

[50, 20, 40, 10, 60, 30]

## 2. LinkedHashSet :-

- LinkedHashSet is a child class of HashSet.
- It was introduced in JDK 1.4 version and derived in java.util package.
- LinkedHashSet is an ordered collection of elements i.e. it maintains the insertion order.
- We cant store duplicate elements in it.
- Only one null insertion is allowed.
- LinkedHashSet is a combination of LinkedList and HashSet.
- The internal implementation of LinkedHashSet is doubly linked list and hashmap.
- LinkedHashSet is not synchronized.
- Its initial capacity is 16 with the default load factor of 0.75.

★ **Constructors of LinkedHashSet:-**

i. public LinkedHashSet()
ii. public LinkedHashSet(int initialCapacity)
iii. public LinkedHashSet(Collection c)
iv. public LinkedHashSet(int initialCapacity, float loadFactor)

### i. public LinkedHashSet():-

- Creates a new empty LinkedHashSet with the default initial capacity of 16 and load factor of 0.75.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
	public static void main(String[] args)
	{
		LinkedHashSet<Integer> a = new LinkedHashSet<Integer>();
		for(int i=1;i<=20;i++)
		{
			a.add(i);
		}
		System.out.println(a);
	}
}
```

**Output:-**
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

### ii. public LinkedHashSet(int initialCapacity):-

- It constructs an empty LinkedHashSet with the specified initial capacity and load factor of 0.75.
- E.g.

```
import java.util.*;
class InitialCapacityArgumentConstructor
{
	public static void main(String[] args)
	{
		LinkedHashSet<Integer> a = new LinkedHashSet<Integer>(10);
		for(int i=1;i<=10;i++)
		{
			a.add(i);
		}
		System.out.println(a);
	}
}
```

**Output:-**
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

### iii. public LinkedHashSet(Collection c):-

- It constructs a new LinkedHashSet with the same elements as specified in the Collection.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructor
{
	public static void main(String[] args)
	{
		ArrayList<Integer> list = new ArrayList<Integer>();
		for(int i=10;i<=100;i+=10)
		{
			list.add(i);
		}
		System.out.println(list);
		LinkedHashSet<Integer> set = new LinkedHashSet<Integer>(list);
		System.out.println(set);
	}
}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

### iv. public LinkedHashSet(int initialCapacity, float loadFactor):-

- It constructs an empty LinkedHashSet with the specified initial capacity and specified load factor.
- E.g.

```
import java.util.*;
class CapacityAndLoadFactorArgumentConstructor
{
	public static void main(String[] args)
	{
		LinkedHashSet<Integer> set = new LinkedHashSet<Integer>(10,0.5f);
		for(int i=10;i<=100;i+=10)
		{
			set.add(i);
		}
		System.out.println(set);
	}
}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

## 3. TreeSet:-

- TreeSet is an implementation class of NavigableSet interface.
- TreeSet was introduced in JDK 1.2 version and derived in java.util package.
- TreeSet also implements several other interfaces,
  i. SortedSet
  ii. Set
  iii. Collection
  iv. Iterable
  v. Serializable
  vi. Cloneable
- The TreeSet internally uses a TreeMap for storing its elements and the TreeMap's internal implementation is **"Red Black Self Balanced Binary Tree"**.
- As its an implementation class of Set interface it inherits a functionality from it and does not stores duplicate elements.
- No null values is allowed in TreeSet.
- We cannot store heterogeneous data elements as it performs sorting.
- The elements are ordered using the natural ordering (ascending), or by using a Comparator we can provide our customise ordering of its elements.
- The initial capacity of TreeSet is 16.
- TreeSet provides a TimeComplexity of O(logn) for the basic operations of add, remove and contains.
- O(logn) represents logarithmic time complexity.
- NavigableSet interface provides the methods to navigate such as,
  higher(), lower(), ceiling(), floor().

★ **Constructors of TreeSet:-**

i. public TreeSet()
ii. public TreeSet(Comparator c)
iii. public TreeSet(Collection c)

### i. public TreeSet():-

- It creates a new empty TreeSet, sorted according to the natural ordering of its elements.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
    public static void main(String[] args)
    {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(5);
        set.add(10);
        set.add(2);
        set.add(3);
        set.add(7);
        set.add(1);
        set.add(6);
        System.out.println(set);
    }
}
```

**Output:-**

[1, 2, 3, 5, 6, 7, 10]

### ii. public TreeSet(Collection c):-

- It creates a new TreeSet containing the elements from the specified collection sorted accoding to the natural ordering of elements.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructor
{
	public static void main(String[] args)
	{
		ArrayList<Integer> list = new ArrayList<>();
		for(int i=1;i<=10;i++)
		{
			list.add((int)(Math.random()*9));
		}
		System.out.println(list);
		TreeSet<Integer> set = new TreeSet<Integer>(list);
		System.out.println(set);
	}
}
```

### iii. public TreeSet(Comparator c):-

- It creates a new empty TreeSet, sorted according to the specified Comparator.
- E.g.

```
import java.util.*;
class Marker
{
	String brand;
	String color;
	double price;
	public Marker(String brand, String color, double price)
	{
		super();
		this.brand = brand;
		this.color = color;
		this.price = price;
	}
	@Override
	public String toString()
	{
		return "Marker:- Brand = "+brand+", Color = "+color+", Price = "+price;
	}
}
class SortByBrand implements Comparator<Marker>
{
	@Override
	public int compare(Marker m1, Marker m2)
	{
		//for ascending order
		return m1.brand.compareTo(m2.brand);
		//for descending order
		// return m2.brand.compareTo(m1.brand);
	}
}
class ComparatorArgumentConstructor
{
	public static void main(String[] args)
	{
		//sort by price ascending
		Comparator<Marker> SortByPriceAsc = (o1,o2)->(int)(o1.price-o2.price);
		//sort by price descending
```

```
                Comparator<Marker> SortByPriceDesc = (o1,o2)->(int)(o2.price-o1.price);
                //Sort by color asc
        Comparator<Marker> SortByColorAsc = (o1,o2)->o1.color.compareTo(o2.color);
                //Sort by color desc
        Comparator<Marker> SortByColorDesc = (o1,o2)->o2.color.compareTo(o1.color);
                TreeSet<Marker> set = new TreeSet<>(SortByColorDesc);
                set.add(new Marker("CAMLIN","RED",25));
                set.add(new Marker("CAMLIN","BLUE",26));
                set.add(new Marker("CAMLIN","GREEN",22));
                set.add(new Marker("CAMLIN","BLACK",27));
                Iterator<Marker> it = set.iterator();
                while(it.hasNext())
                {
                        System.out.println(it.next());
                }
        }
}
```

**Output:-**

```
Marker:- Brand = CAMLIN, Color = RED, Price = 25.0
Marker:- Brand = CAMLIN, Color = GREEN, Price = 22.0
Marker:- Brand = CAMLIN, Color = BLUE, Price = 26.0
Marker:- Brand = CAMLIN, Color = BLACK, Price = 27.0
```

## ★ Methods of TreeSet:-

i. public E first();
ii. public E last();
iii. public E floor(E ele);
iv. public E ceiling(E ele);
v. public E higher(E ele);
vi. public E pollFirst();
vii. public E pollLast();
viii. public E addFirst(E ele);
ix. public E addLast(E ele);
x. public SortedSet subSet(E ele, E ele);
xi. public SortedSet headSet(E ele);
xii. public SortedSet tailSet(E ele);

**i. public E first():-**

- This method returns the first (lowest) element currently in this Set.
- It throws, NoSuchElementException if the Set is empty.

**ii. public E last():-**

- This method returns the last (highest) element currently in this Set.
- It throws, NoSuchElementException if the Set is empty.

**iii. public E pollFirst():-**

- This method retrives and removes the first element from this Set.
- It returns null if the Set is empty.

**iv. public E pollLast():-**

- This method retrives and removes the last element from this Set.
- It returns null if the Set is empty.

**v. public E higher(E ele):-**

- This method returns the least element from this Set strictly greater than the given element.
- It returns null is there if no such element.

**vi. public E lower(E ele):-**

- This method returns the greatest element from this Set strictly lower than the given element.
- It returns null if there is no such element.

**Example of above six methods :-**

```
import java.util.*;
class MethodsExample
{
	public static void main(String[] args)
	{
		List<Integer> list = Arrays.asList(10,20,30,40,50,60,70,80,90);
		TreeSet<Integer> set = new TreeSet<>(list);
		System.out.println(set.first());
		System.out.println(set.last());
		System.out.println(set.pollFirst());
		System.out.println(set.pollLast());
		System.out.println(set.higher(40));
		System.out.println(set.lower(30));
		System.out.println(set);
	}
}
```

**Output:-**

```
10
90
10
90
50
20
[20, 30, 40, 50, 60, 70, 80]
```

**vii. public E floor(E ele):-**

- This method returns the greatest element from the Set which is less than or equals to the given element.
- It returns null if no such element is present.

**viii. public E ceiling(E ele):-**

- This method returns the lowest element from the Set which is greater than or equals to the given element.
- It returns null if no such element is present.

**Example for above floor and ceiling methods:-**

```
import java.util.*;
class FloorAndCeilingMethodExample
{
	public static void main(String[] args)
	{
		TreeSet<Integer> set = new TreeSet<Integer>();
		for(int i=10;i<=100;i+=10)
			set.add(i);
		System.out.println(set);
		System.out.println(set.ceiling(52));	//60
		System.out.println(set.ceiling(55));	//60
		System.out.println(set.ceiling(57));	//60
		System.out.println(set.floor(52));	//50
		System.out.println(set.floor(55));	//50
		System.out.println(set.floor(57));	//50
	}
}
```

**ix. public SortedSet subset(E fromElement, E toElement):-**

- This method returns a view of a portion of this Set whose element ranges from fromElement till toElement.
- If fromElement and toElement are equal it returns an empty Set.
- The return type of this method is SortedSet.
- E.g.

```
package demo;

import java.util.SortedSet;
import java.util.TreeSet;

public class subSetMethodExample {

    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<Integer>();
        for(int i=10;i<=100;i+=10)
            set.add(i);
        System.out.println(set);
        TreeSet<Integer> subSet1 = (TreeSet<Integer>)set.subSet(30, 70);
        System.out.println(subSet1);
        SortedSet<Integer> subSet2 = set.subSet(10, 50);
        System.out.println(subSet2);
    }

}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[30, 40, 50, 60]
[10, 20, 30, 40]
```

**Note:-**

- If the first argument is greater than second argument this method throws IllegalArgumentException.
- If we provide an argument which is not present it will use its ceiling value.

**x. public SortedSet headset(E ele):-**

- This method returns a view of the portion from this Set whose elements are strictly less than the specified element.

**xi. public SortedSet tailSet(E ele):-**

- This method returns a view of the portion from this Set whose elements are greater than or equal to specified object.

**Example of headSet() and tailSet():-**

```
import java.util.*;
class headSetAndtailSetMethodExample
{
	public static void main(String[] args)
	{
		TreeSet<Integer> set = new TreeSet<Integer>();
		for(int i=10;i<=100;i+=10)
			set.add(i);
		System.out.println(set);
		System.out.println(set.headSet(54));
		System.out.println(set.headSet(50));
		System.out.println(set.headSet(5));
		System.out.println(set.headSet(15));
		System.out.println(set.headSet(50));
		System.out.println(set.tailSet(50));
		System.out.println(set.tailSet(100));
		System.out.println(set.tailSet(10));
		System.out.println(set.tailSet(15));
	}
}
```

**Output:-**

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[10, 20, 30, 40, 50]
[10, 20, 30, 40]
[]
[10]
[10, 20, 30, 40]
[50, 60, 70, 80, 90, 100]
[100]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[20, 30, 40, 50, 60, 70, 80, 90, 100]
```

## ★ Iterable interface:-

- The Iterable interface is a root of Collection interface.
- It was introduced in JDK 1.5 version and derived in java.lang package.
- This interface represents a collection of objects that can br iterated using an Iterator.
- It provides the ability to traverse the elements sequentially in a collection.
- The Iterable interface has a single abstract method in it is iterator().
- Iterable is an example of functional interface.
- It also contains a default method i.e. forEach().
- The interfaces which extends Iterable are,
  Collection, List, Set, Queue, SortedSet, NavigableSet, Deque

### ★ forEach():-

- This method performs the given action by iterating the elements of the Iterable until all elements is not processed.
- **E.g. 1**

```
import java.util.*;
class forEachMethodExample
{
	public static void main(String[] args)
	{
		TreeSet<Integer> set = new TreeSet<Integer>();
		for(int i=1;i<=10;i++)
			set.add(i);
		System.out.println(set);
		set.stream().filter(ele->ele%2==0).forEach(ele->System.out.println(ele+" "));
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
4
6
8
10
```

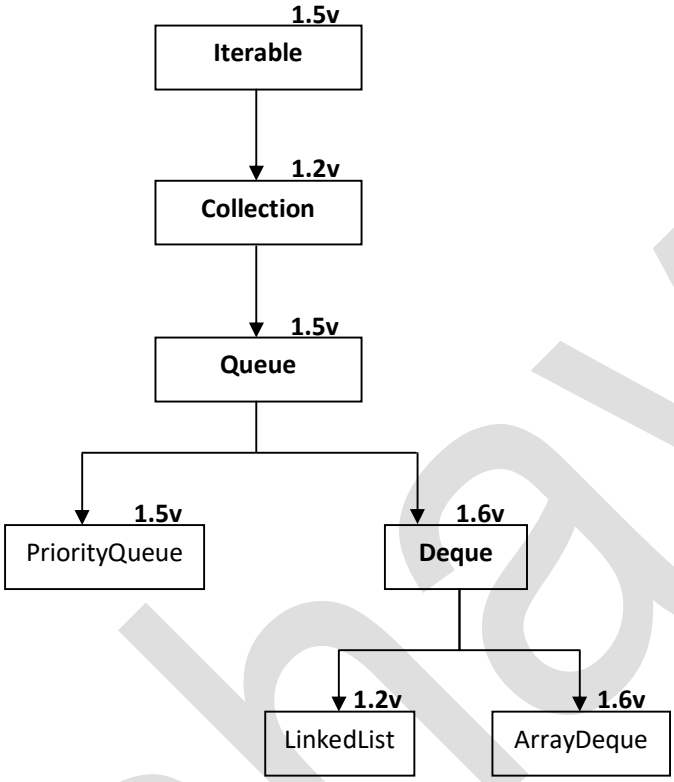
- **E.g. 2**

```
import java.util.*;
class forEachMethodExample2
{
	public static void main(String[] args)
	{
		ArrayList<Integer> list = new ArrayList<Integer>();
		for(int i=10;i<=100;i+=10)
			list.add(i);
		list.forEach(ele->System.out.print(ele+" "));
	}
}
```

**Output:-**

```
10 20 30 40 50 60 70 80 90 100
```

# ★ Queue interface :-




## Classification of Queue

- Queue is an sub interface of Collection interface.
- It is an ordered list of objects which is used to insert the elements at the end of the list and removing it from the start of this list.
- Queue itself is a datastructure which follows the simple principle of First In First Out (FIFO).
- It was introduced in JDK 1.5 version and derived in java.util package.
- Classes which implements Queue are PriorityQueue, LinkedList, ArrayDeque, PriorityBlockingQueue, LinkedBlockingQueue, etc.
- There are some sub interfaces of Queue such as Deque, BlockingQueue
- Queue can be implemented using LinkedList, BinaryHeap ( A binary heap is nothing but a complete binary tree).
- The initial capacity of Queue is 11.

★ **Methods of Queue:-**

i. public abstract boolean add(E ele);
ii. public abstract boolean offer(E ele);
iii. public abstract E remove();
iv. public abstract E poll();
v. public abstract E element();
vi. public abstract E peek();

i. **public abstract boolean add(E ele):-**
   - This method inserts the specified element into this Queue.
   - It returns true if element is been added or else returns false.

ii. **public abstract boolean offer(E ele):-**
   - This method inserts the specified element into this Queue.
   - It returns true if element is been added or else returns false.
   - Method throws ClassCastException if the specified element prevents it from being added to this Queue.
   - It throws NullPointerException if the specified element is null.
   - Also it throws IllegalArgumentException if the Queue prevents it from being added the element into the Queue.

iii. **public abstract E remove():-**
   - This method retrives and remove the head of this Queue.
   - This method is different from poll() as it throws an Exception if the Queue if empty.
   - It throws NoSuchElementException if the Queue is empty.

iv. **public abstract E poll():-**
   - This method retirves and remove the head of this Queue.
   - It returns null if the Queue is empty.

v. **public abstract E element():-**
   - This method retrives but doesn't remove the head of this Queue.
   - It throws NoSuchElementException if the Queue is empty.

vi. **public abstract E peek():-**
   - This method retrives but doesn't remove the head this Queue.
   - It returns null if the Queue if empty.

**Example on methods of Queue:-**

```
import java.util.*;
class QueueMethods
{
	public static void main(String[] args)
	{
		Queue<Integer> queue = new PriorityQueue<>();
		for(int i=10;i<=100;i+=10)
			queue.add(i);
		System.out.println(queue);
		System.out.println();
		System.out.println(queue.add(110));
		System.out.println(queue);
		System.out.println(queue.offer(110));
		System.out.println(queue);
		System.out.println();
		System.out.println(queue.remove());
		System.out.println(queue);
		System.out.println(queue.poll());
		System.out.println(queue);
		System.out.println();
		System.out.println(queue.element());
		System.out.println(queue);
		System.out.println(queue.peek());
		System.out.println(queue);
	}
}
```

Output:-

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

true
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
true
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 110]

10
[20, 40, 30, 80, 50, 60, 70, 110, 90, 100, 110]
20
[30, 40, 60, 80, 50, 110, 70, 110, 90, 100]

30
[30, 40, 60, 80, 50, 110, 70, 110, 90, 100]
30
[30, 40, 60, 80, 50, 110, 70, 110, 90, 100]
```

## ★ PriorityQueue:-

- PriorityQueue is an implementation class of Queue interface.
- It was introduced in JDK 1.5 version and derived in java.util package.
- When we should go for the PriorityQueue?
- The PriorityQueue is used when the object is being processed based on certain priority.
- It is known that Queue follows the FIFO algorithm, but sometimes the elements of the Queue needed to be processed according to the Priority that's when the PriorityQueue comes into the picture.
- The PriorityQueue is implemented using a priority heap, a priority heap is nothing but a binary heap ( A binary heap is nothing but complete binary tree).
- There are two types of binary heap, i. MaxHeap (Deseceinding ), ii. MinHeap ( Ascending )
- In MaxHeap the root element will be always the largest element and it stores the data in descending order.
- In MinHeap the root element will be always the smallest element and it stores the data in ascending order ( default implementation of PriorityQueue is MinHeap).
- It uses Comparable for the natural sorting of elements.
- If we want to customize the sorting we can use Comparator.

### ★ Constructors of PriorityQueue:-

i. public PriorityQueue();
ii. public PriorityQueue(int initialCapacity);
iii. public PriorityQueue(Collection c);
iv. public PriorityQueue(Comparator c);

### i. public PriorityQueue():-

- It constructs an empty PriorityQueue with the default initial capacity of 11, that order its elements according to natural ordering.
- E.g.

```
import java.util.*;
class NoArgumentConstructorExample
{
    public static void main(String[] args)
    {
        PriorityQueue<Integer> queue = new PriorityQueue<Integer>();
        for(int i=10;i<=110;i+=10)
            queue.add(i);
        System.out.println(queue);
    }
}
```

**Output:-**

[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]

### ii. public PriorityQueue(int initialCapacity):-

- It creates a PriorityQueue with the specified initial capacity that order its elements according to the natural ordering.
- Parameters:-
- initialCapacity:-
- The initial capacity for this priority queue.
  Throws:- IllegalArgumentException
  If intial capacity is less than 1.

- **Example of initialCapacity argument constructor:-**

```
import java.util.*;
class InitialCapacityArgumentConstructorExample
{
    public static void main(String[] args)
    {
        PriorityQueue<Integer> queue = new PriorityQueue<Integer>(10);
        for(int i=10;i<=100;i+=10)
            queue.add(i);
        System.out.println(queue);
    }
}
```

**Output:-**

[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

### iii. public PriorityQueue(Collection c):-

- It constructs a PriorityQueue containing the elements in a specified collection.
- The capacity of this PriorityQueue will be equals to the number of elements in specified collection.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructorExample
{
    public static void main(String[] args)
    {
        SortedSet<Integer> set = new TreeSet<Integer>();
        set.add(100);
        set.add(50);
        set.add(90);
        set.add(40);
        set.add(70);
        set.add(60);
        System.out.println(set);
        PriorityQueue<Integer> queue = new PriorityQueue<Integer>(set);
        System.out.println(queue);
    }
}
```

**Output:-**

[40, 50, 60, 70, 90, 100]
[40, 50, 60, 70, 90, 100]

### iv. public PriorityQueue(Comparator c):-

- It creates a PriorityQueue with default initial capacity of 11 that order its elements according to the specified Comparator ( custom ordering ).
- E.g.

```
import java.util.*;
class Marker{
    String brand;
    String color;
    double price;
    public Marker(String brand, String color, double price){
        super();
        this.brand = brand;
        this.color = color;
        this.price = price;
    }
    @Override
    public String toString(){
        return "Marker:- Brand = "+brand+", Color = "+color+", Price = "+price;
    }
}
class SortByColor implements Comparator<Marker>{
    @Override
    public int compare(Marker m1, Marker m2){
        //for ascending order
        return m1.color.compareTo(m2.color);
        //for descending order
        // return m2.brand.compareTo(m1.brand);
    }
}
class ComparatorArgumentConstructorExample{
    public static void main(String[] args) {
        //sort by price ascending
        Comparator<Marker> SortByPriceAsc = (o1,o2)->(int)(o1.price-o2.price);
        //sort by price descending
        Comparator<Marker> SortByPriceDesc = (o1,o2)->(int)(o2.price-o1.price);
         PriorityQueue<Marker> queue = new PriorityQueue<>(new SortByBrand());
        queue.add(new Marker("CAMLIN","RED",25));
        queue.add(new Marker("CAMLIN","BLUE",26));
        queue.add(new Marker("CAMLIN","GREEN",22));
        queue.add(new Marker("CAMLIN","BLACK",27));
        Iterator<Marker> it = queue.iterator();
        while(it.hasNext())
        {
            System.out.println(it.next());
        }
    }
}
```

**Output:-**

```
Marker:- Brand = CAMLIN, Color = BLACK, Price = 27.0
Marker:- Brand = CAMLIN, Color = BLUE, Price = 26.0
Marker:- Brand = CAMLIN, Color = GREEN, Price = 22.0
Marker:- Brand = CAMLIN, Color = RED, Price = 25.0
```

### v. public PriorityQueue(int initialCapacity, Comparator c):-

- It creates a PriorityQueue with the specified initial capacity that order its elements according to the specified Comparator.
- E.g.

```
import java.util.*;
class Marker{
    String brand;
    String color;
    double price;
    public Marker(String brand, String color, double price){
        super();
        this.brand = brand;
        this.color = color;
        this.price = price;
    }
    @Override
    public String toString(){
        return "Marker:- Brand = "+brand+", Color = "+color+", Price = "+price;
    }
}
class SortByColor implements Comparator<Marker>{
    @Override
    public int compare(Marker m1, Marker m2){
        //for ascending order
        return m1.color.compareTo(m2.color);
        //for descending order
        // return m2.brand.compareTo(m1.brand);
    }
}
class ComparatorAndInitialCapacityArgumentConstructorExample{
    public static void main(String[] args) {
        PriorityQueue<Marker> queue = new PriorityQueue<>(10,new
SortByColor());
        queue.add(new Marker("CAMLIN","RED",25));
        queue.add(new Marker("CAMLIN","BLUE",26));
        queue.add(new Marker("CAMLIN","GREEN",22));
        queue.add(new Marker("CAMLIN","BLACK",27));
        Iterator<Marker> it = queue.iterator();
        while(it.hasNext())
        {
            System.out.println(it.next());
        }
    }
}
```

**Output:-**

```
Marker:- Brand = CAMLIN, Color = BLACK, Price = 27.0
Marker:- Brand = CAMLIN, Color = BLUE, Price = 26.0
Marker:- Brand = CAMLIN, Color = GREEN, Price = 22.0
Marker:- Brand = CAMLIN, Color = RED, Price = 25.0
```

## ★ Deque:-

- Deque is an sub interface of Queue.
- It is introduced in JDK 1.6 version and derived in java.util package.
- The name Deque is short for "**Double Ended Queue**" and usually pronounced as Deck.
- It is a linear collection of elements that supports insertion and removal operation from both the ends.
- Java's Deque is a data structure that combines an ordinary Stack and Queue.
- So basically it follows both the principles of Stack and Queue i.e. LIFO and FIFO respectively.
- It has some implementation classes like ArrayDeque and LinkedList.

## ★ Methods of Deque:-

i. public abstract void addFirst(E ele);
ii. public abstract void addLast(E ele);
iii. public abstract boolean offerFirst(E ele);
iv. public abstract boolean offerLast(E ele);
v. public abstract E removeFirst();
vi. public abstract E removeLast();
vii. public abstract E pollFirst();
viii. public abstract E pollLast();
ix. public abstract E getFirst();
x. public abstract E getLast();
xi. public abstract E peekFirst();
xii. public abstract E peekLast();
xiii. public abstract boolean removeFirstOccurrence(Object obj);
xiv. public abstract boolean removeLastOccurrence(Object obj);

### i. public abstract void addFirst(E ele);

- This method inserts a specified element at the front of this Deque.

### ii. public abstract void addLast(E ele);

- This method inserts a specified element at the end of this Deque.

### iii. public abstract boolean offerFirst(E ele);

- This method inserts a specified element at the front of this Deque.

### iv. public abstract boolean offerLast(E ele);

- This method inserts a specified element at the end of this Deque.

### v. public abstract E removeFirst();

- This method removes and retrives the first element of this Deque.

### vi. public abstract E removeLast();

- This method removes and retrives the last element of this Deque.

### vii. public abstract E pollFirst();

- This method removes and retrives the first element of this Deque.
- It returns null if the Deque is empty.

### viii. public abstract E pollLast();

- This method removes and retrives the last element of this Deque.
- It returns null if the Deque is empty.

### ix. public abstract E getFirst();

- It retrives but doesn't remove the first element of this Deque.

### x. public abstract E getLast();

- It retrives but doesn't remove the last element of this Deque.

**xi. public abstract E peekFirst();**

- It retrives but doesn't remove the first element of this Deque.
- It returns null if the Deque is empty.

**xii. public abstract E peekLast();**

- It retrives but doesn't remove the first element of this Deque.
- It returns null if the Deque is empty.

**xiii. public abstract boolean removeFirstOccurrence(Object obj);**

- It removes the first occurrence of the specified element from this Deque.

**xiv. public abstract boolean removeLastOccurrence(Object obj);**

- It removes the last occurrence of the specified element from this Deque.

**Example on first eight methods:-**

```
import java.util.*;
class FirstEightMethodExample
{
  public static void main(String[] args)
  {
        Deque<Integer> dq = new ArrayDeque<Integer>();
        dq.addFirst(10);
        dq.addFirst(20);
        dq.addLast(40);
        dq.addLast(30);
        System.out.println(dq);
        System.out.println(dq.offerFirst(5));
        System.out.println(dq.offerLast(45));
        System.out.println(dq);
        System.out.println(dq.removeFirst());
        System.out.println(dq);
        System.out.println(dq.removeLast());
        System.out.println(dq);
        System.out.println(dq.pollFirst());
        System.out.println(dq);
        System.out.println(dq.pollFirst());
        System.out.println(dq);
  }
}
```

**Output:-**

```
[20, 10, 40, 30]
true
true
[5, 20, 10, 40, 30, 45]
5
[20, 10, 40, 30, 45]
45
[20, 10, 40, 30]
20
[10, 40, 30]
10
[40, 30]
```

**Example on last six methods:-**

```
import java.util.*;
class LastSixMethodExample
{
  public static void main(String[] args)
  {
        Deque<Integer> dq = new ArrayDeque<Integer>();
        for(int i=10;i<=100;i+=10)
              dq.add(i);
        dq.addFirst(10);
        dq.addLast(70);
        System.out.println(dq);
        System.out.println(dq.getFirst());
        System.out.println(dq.getLast());
        System.out.println(dq.peekFirst());
        System.out.println(dq.peekLast());
        System.out.println(dq.removeFirstOccurrence(10));
        System.out.println(dq);
        System.out.println(dq.removeLastOccurrence(70));
        System.out.println(dq);
  }
}
```

**Output:-**

```
[10, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 70]
10
70
10
70
true
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 70]
true
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

## ★ ArrayDeque:-

- ArrayDeque is an implementation class of Deque interface.
- It was introduced in JDK 1.6 version and derived in java.util package.
- Internal implementation of an ArrayDeque is growable array where we can perform operations from both the ends.
- ArrayDeque is not synchronized so its not thread safe.
- It is than Stack.
- ArrayDeque has an initial capacity of 16.

## ★ Constructors of ArrayDeque :-

i. public ArrayDeque()
ii. public ArrayDeque(int initialCapacity)
iii. public ArrayDeque(Collection c)

### i. public ArrayDeque() :-

- It constructs an empty ArrayDeque with an initial capacity of 16.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
	public static void main(String[] args)
	{
		ArrayDeque<Integer> dq = new ArrayDeque<Integer>();
		for(int i=1;i<=16;i++)
			dq.add(i);
		System.out.println(dq);
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

### ii. public ArrayDeque(int initialCapacity) :-

- It constructs an empty ArrayDeque with the specified initial capacity.
- E.g.

```
import java.util.*;
class InitialCapacityArgumentConstructor
{
	public static void main(String[] args)
	{
		ArrayDeque<Integer> dq = new ArrayDeque<Integer>(10);
		for(int i=1;i<=10;i++)
			dq.add(i);
		System.out.println(dq);
	}
}
```

**Output:-**

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

**iii. public ArrayDeque(Collection c) :-**

- It constructs a new ArrayDeque with the specified elements from the collection.
- E.g.

```
import java.util.*;
class CollectionArgumentConstructor
{
	public static void main(String[] args)
	{
		List<Integer> list = Arrays.asList(9,3,6,2,2,65,6,2,1,36);
		System.out.println(list);
		ArrayDeque<Integer> dq = new ArrayDeque<Integer>(list);
		System.out.println(dq);
	}
}
```

**Output:-**

```
[9, 3, 6, 2, 2, 65, 6, 2, 1, 36]
[9, 3, 6, 2, 2, 65, 6, 2, 1, 36]
```

★ **Difference between List and Set :-**

| List | Set |
|---|---|
| i. List is an ordered collection of elements. | i. Set is an unordered collection of elements. |
| ii. We can store duplicate elements in List. | ii. We cannot store duplicate elements in Set. |
| iii. We can store multiple null values in List. | iii. We cannot store multiple null values in Set. |
| iv. It has four implementation classes. | iv. It has three implementation classes. |

★ **Difference between Array and ArrayList :-**

| Array | ArrayList |
|---|---|
| i. Array is an index collection of fixed number of homogeneous elements. | i. It is an collection of group of individual objects. |
| ii. Arrays are fixed in size. | ii. ArrayList is not dynamic in size. |
| iii. We can store both primitive and non primitive data type in Arrays. | iii. We can store only non primitive datatype in ArrayList. |
| iv. Arrays are faster in performance. | iv. ArrayList is slower compare to Arrays. |
| v. Arrays consumes more memory. | v. ArrayList consumes less memory. |
| vi. It does not have built in methods for storing, removing or searching the elements. | vi. It provides several built in methods like add(), remove(), contains(), etc. |

★ **Difference between Queue and Deque :-**

| Queue | Deque |
|---|---|
| i. Queue is a sub interface of Collection interface. | i. Deque is a sub interface of Queue interface. |
| ii. Queue follows the FIFO data structure. | ii. Deque follows both FIFO and LIFO data structure. |
| iii. Queue is introduced in JDK 1.5 version. | iii. Deque is introduced in JDK 1.6 version. |
| iv. The insertion of elements will always from end of the list. | iv. The insertion of elements is done in both the ways i.e. from the front as well as end. |
| v. The removing of elements will always from the front of the list. | v. The removing of elements is done in both the ways i.e. from the front as well as end. |
| vi. We can access only front element in a queue. | vi. We can access each element in a deque. |

★ **Difference between Stack and Queue :-**

| Stack | Queue |
|---|---|
| i. A linear data structure that follows the Last In First Out (LIFO) principle. | i. A linear data structure that follows the First In First Out (FIFO) principle. |
| ii. Elements are added and removed from the same end (top). | ii. Elements added from the end (tail) and removed from the start (head). |
| iii. Stack has default capacity of 10. | iii. Queue has default capacity of 11. |
| iv. Stack introduced in JDK 1.0 version and it is a known as Legacy class. | iv. Queue introduced in JDK 1.5 version. |
| v. Stack is a sub class of Vector. | v. Queue is a sub interface of Collection. |
| vi. Stack implements RandomAccess interface. | vi. Queue doesn't extends RandomAccess interface. |

## ★ Difference between LinkedList and LinkedHashSet :-

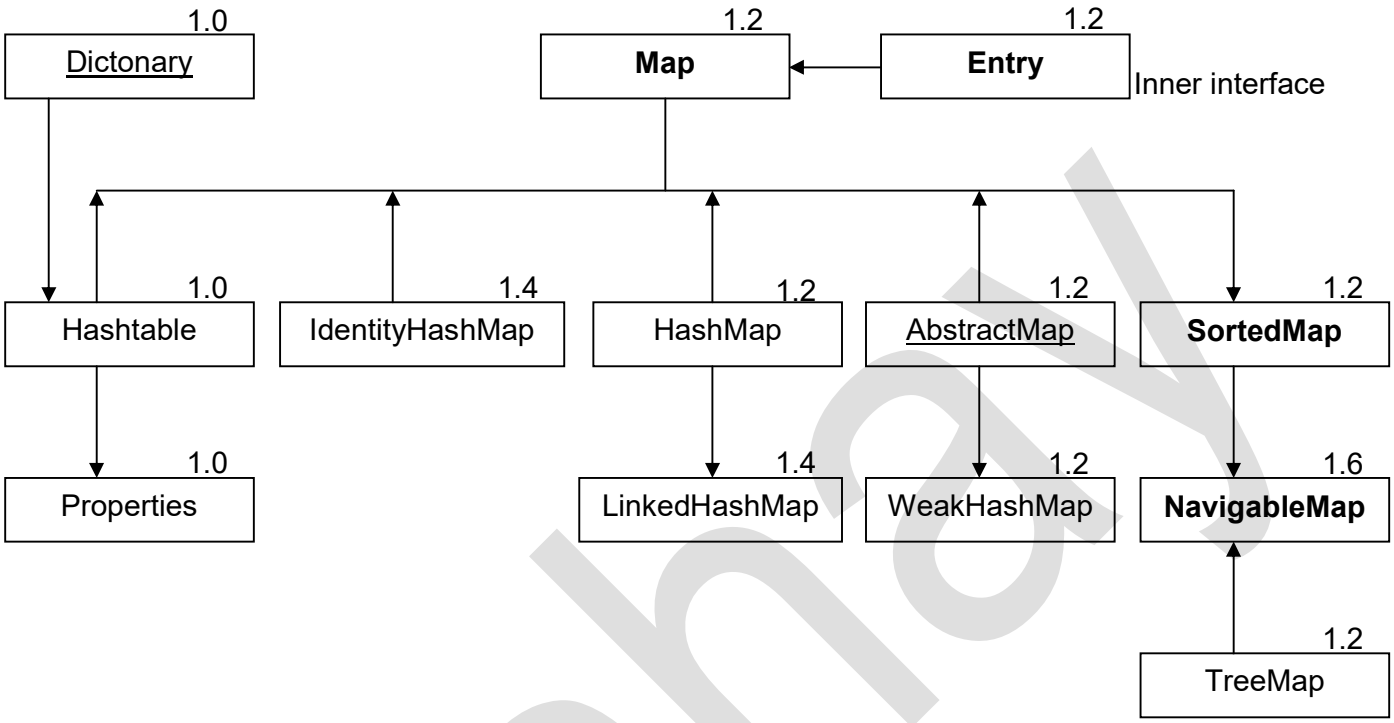
| LinkedList | LinkedHashSet |
|---|---|
| i. LinkedList is introduced in JDK 1.2 version. | i. LinkedHashSet is introduced in JDK 1.5 version. |
| ii. We can store duplicate elements in LinkedList. | ii. We cannot store duplicate elements in LinkedHashSet. |
| iii. We can store multiple null values in LinkedList. | iii. We can store only one null value in LinkedHashSet. |
| iv. Internal implementation of LinkedList is doubly linked list. | iv. Internal implementation of LinkedHashSet is doubly linked list and hash map. |
| v. LinkedList does not have capacity. | v. LinkedHashSet have default 16 capacity with the default load factor of 0.75. |
| vi. LinkedList is implementation class of List and Queue interface. | vi. LinkedHashSet is an sub class of HashSet. |

## ★ Difference between Collection and Map:-

| Collection | Map |
|---|---|
| i. A Collection is used to store and manage a group of individual objects (elements). | i. A Map is used to store key-value pairs. |
| ii. In Collection there is no concept of Key. | ii. In Map every value is paired with a Key. |
| iii. Collection does not have any inner interface. | iii. Map have a inner interface called as Entry. |
| iv. The Collection is not directly implemented by class. | iv. Map is directly implemented by class. |
| v. E.g.<br>List, Set, Queue | v. E.g.<br>HashMap, TreeMap |

# ★ Map interface :-

| | |
|---|---|
| **BOLD TEXT** | **:- Interfaces** |
| No BOLD TEXT | :- Classes |
| Underline TEXT | :- abstract class |




## Classification Of Map

★ **Introduction:-**

- Map in java is an interface which was introduced in JDK 1.2 and derived in java.
- It stores the data in the form of key and value pair.
- It is called as “Map” because it does the mapping between key and value.
- A Map cannot contain duplicate keys and one key can map atmost one value.
- The Map interface provides us three collection view which allows to map a Map contains to bew view as Set of keys, Collection of values ot Set of key and value.
- Each key and value which we store is known as “Entry” (Entry is an inner interface of Map).

★ **When we should use Map?**

→

- When we want to store group of individual object as a key and value pair representing them as a single entity is called as Map.

**Note:-**

- Map.Entry is an inner interface which is present inside Map interface.

## ★ Methods of Map interface:-

i. public abstract int size();
ii. public abstract boolean isEmpty();
iii. public abstract boolean containsKey(Object obj);
iv. public abstract boolean containsValue(Object obj);
v. public abstract V get(Object key);
vi. public abstract V put(K key, V value);
vii. public abstract V remove(Object key);
viii. public abstract void putAll(Map map);
ix. public abstract void clear();
x. public abstract Set <K> keySet();
xi. public abstract Collection <V> values();
xii. public default V replace(K key, V value);
xiii. public abstract Set <Map.Entry<K,V>> entrySet();

**i. public abstract int size();**

- It returns the number of Key Value mappings from this Map.

**ii. public abstract boolean isEmpty();**

- This method returns true if this Map contains no key value mapping in it or else returns false.

**iii. public abstract boolean containsKey(Object obj);**

- This method returns true if this Map contains a mapping for the specified key.

**iv. public abstract boolean containsValue(Object obj);**

- This method returns true if this Map contains one or more keys for the specified value.

**Example:-**

```
import java.util.*;
class MapExample1
{
  public static void main(String[] args)
  {
        Map<Integer,String> map = new LinkedHashMap<Integer,String>();
        map.put(1,"Ramesh");
        map.put(2,"Mahesh");
        map.put(3,"Suresh");
        map.put(4,"Mukesh");
        System.out.println(map);
        System.out.println(map.put(1,"Nitesh"));
        System.out.println(map);
        for(int i=0;i<map.size();i++)
        {
              String value = map.get(i+1);
              System.out.println((i+1)+" : "+value);
        }
  }
}
```

**Output:-**

```
{1=Ramesh, 2=Mahesh, 3=Suresh, 4=Mukesh}
Ramesh
{1=Nitesh, 2=Mahesh, 3=Suresh, 4=Mukesh}
1 : Nitesh
2 : Mahesh
3 : Suresh
4 : Mukesh
```

v. **public abstract V get(Object key);**
- This method returns the value to which the specified key is mapped, or returns null if this map contains no mapping for the specified key.

**Example:-**

```
import java.util.*;
class MapExample2{
  public static void main(String[] args) {
        Map<Integer,String> map1 = new LinkedHashMap<Integer,String>();
        map1.put(1,"Ramesh");
        map1.put(2,"Suresh");
        System.out.println(map1);
        Map<Integer,String> map2 = new LinkedHashMap<Integer,String>();
        map2.put(3,"Kamlesh");
        map2.put(4,"Mahesh");
        System.out.println(map2);
        map1.putAll(map2);
        Set<Map.Entry<Integer,String>> entries = map1.entrySet();
        for(Map.Entry<Integer,String> entry : entries){
              System.out.println(entry);
        }
  }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh}
{3=Kamlesh, 4=Mahesh}
1=Ramesh
2=Suresh
3=Kamlesh
4=Mahesh
```

vi. **public abstract V put(K key, V value);**
- This method associates the specified value with the specified key in this map.
- It checks whether the key is present inside the Map, if the key is present it will replace the value with the specified value and will return the old value or else it will add a new entry.

vii. **public abstract V remove(Object key);**
- Removes the entry for the specified key from this map if it is present.

viii. **public abstract void putAll(Map map);**
- It copies all of the mappings from the specified Map into this Map.

ix. **public abstract void clear();**
- This method removes all of the mappings from this Map.

x. **public abstract Set <K> keySet();**
- It returns the Set of view of the keys containing in this Map.

xi. **public abstract Collection <V> values();**
- It returns a Collection view of the values contained in this Map.

xii. **public default V replace(K key, V value);**
- This method replaces the entry for the specified key only if it is currently mapped to some value.

xiii. **public abstract Set <Map.Entry<K,V>> entrySet();**
- This method returns a view of Set of mappings (entries) contained in this Map.

**Example:-**

```
import java.util.*;
class MapExample3
{
  public static void main(String[] args)
  {
        Map<Integer,String> map1 = new LinkedHashMap<Integer,String>();
        map1.put(1,"Ramesh");
        map1.put(2,"Suresh");
        map1.put(3,"Mahesh");
        map1.put(4,"Nitesh");
        System.out.println(map1);
        System.out.println(map1.containsKey(6));
        System.out.println(map1.containsKey(1));
        System.out.println(map1.containsValue("Ramesh"));
        System.out.println(map1.containsValue("Ganesh"));

        Set<Integer> keys = map1.keySet();
        //Using foreach loop
        System.out.println("Printing keys and values using for each loop :");
        for(Integer key : keys)
        {
              System.out.println(key+" : "+map1.get(key));
        }
        //another way to print key and values using foreach method
        System.out.println("Printing keys and values using for each method :");
        map1.forEach((key,value)->System.out.println(key+" : "+value));

        Collection<String> values = map1.values();
        System.out.println("Printing values only");
        for(String value : values)
        {
              System.out.println(value);
        }
  }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mahesh, 4=Nitesh}
false
true
true
false
Printing keys and values using for each loop :
1 : Ramesh
2 : Suresh
3 : Mahesh
4 : Nitesh
Printing keys and values using for each method :
1 : Ramesh
2 : Suresh
3 : Mahesh
4 : Nitesh
Printing values only
Ramesh
Suresh
Mahesh
Nitesh
```

## ★ HashMap:-

- HashMap is an implementation class of Map interface.
- It was introduced in JDK 1.2 and derived in java.util package.
- HashMap stores the data in a form of Key and Value pair.
- It can have one null key insertion and can have multiple values as null.
- HashMap does not maintains the insertion order.
- The internal implementation of a HashMap is Hashtable.
- It have default capacity of 16 with a default load factor 0.75.

### ★ Constructors of HashMap:-

i. public HashMap()
ii. public HashMap(Map map)
iii. public HashMap(int initialCapacity)
iv. public HashMap(int initialCapacity, float loadFactor)

### i. public HashMap():-

- It is used to construct a HashMap with no entries in it, with the default capacity of 16 and default load factor of 0.75.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
    public static void main(String[] args)
    {
            HashMap<Integer, String> map = new HashMap<Integer,String>();
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
    }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
```

### ii. public HashMap(Map map):-

- It creates a new HashMap which contains the entries which are specified in this Map.
- E.g.

```
import java.util.*;
class MapArgumentConstructor
{
    public static void main(String[] args)
    {
            Map<Integer, String> map = new LinkedHashMap<Integer,String>();
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
            HashMap<Integer,String> map2 = new HashMap<Integer,String>(map);
            System.out.println(map2);
    }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
```

**iii. public HashMap(int initialCapacity):-**

- It constructs a new HashMap with no entries in it with the specified initial capacity and default load factor of 0.75.
- E.g.

```
import java.util.*;
class InitialCapacityArgumentConstructor
{
    public static void main(String[] args)
    {
            HashMap<Integer,String> map = new HashMap<Integer,String>(10);
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
    }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
```

**iv. public HashMap(int initialCapacity, float loadFactor):-**

- It constructs a new HashMap with no entries in it with the specified initial capacity and the specified load factor.
- E.g.

```
import java.util.*;
class LoadFactorArgumentConstructor
{
    public static void main(String[] args)
    {
            HashMap<Integer,String> map = new HashMap<Integer,String>(10,0.5f);
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
    }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
```

## ★ LinkedHashMap:-

- LinkedHashMap is a child class of HashMap.
- It stores the data in form of key and value pair.
- It was introduced in JDK 1.4 and derived in java.util package.
- The internal implementation of LinkedHashMap is Hashtable and LinkedList.
- It is an ordered collection of elements which maintains the insertion order.
- LinkedHashMap contains unique elements, we can store only one null key and store multiple null values.
- It is not synchronized because of which its not thread safe.
- The default initial capacity is 16 with the default load factor of 0.75.

### ★ Constructors of LinkedHashMap:-

i. public LinkedHashMap()
ii. public LinkedHashMap(Map map)
iii. public LinkedHashMap(int initialCapacity)
iv. public LinkedHashMap(int intitialCapacity, float loadFactor)

### i. public LinkedHashMap():-

- It is used to construct a new LinkedHashMap with no entries in it, with the default capacity of 16 and default load factor of 0.75.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
    public static void main(String[] args)
    {
            LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer,String>();
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
    }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
```

### ii. public LinkedHashMap(Map map):-

- It creates a new LinkedHashMap which contains the entries which are specified in this Map.
- E.g.

```
import java.util.*;
class MapArgumentConstructor
{
    public static void main(String[] args)
    {
            Map<Integer, String> map = new LinkedHashMap<Integer,String>();
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
            HashMap<Integer,String> map2 = new HashMap<Integer,String>(map);
            System.out.println(map2);
    }
}
```

**Output:-**

```
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}
```

**iii. public LinkedHashMap(int initialCapacity):-**

- It constructs a new LinkedHashMap with no entries in it with the specified initial capacity and default load factor of 0.75.
- E.g.

```
import java.util.*;
class InitialCapacityArgumentConstructor
{
    public static void main(String[] args)
    {
            LinkedHashMap<Integer,String> map = new LinkedHashMap<Integer,String>(10);
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
    }
}
```

**Output:-**

{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}

**iv. public LinkedHashMap(int initialCapacity, float loadFactor):-**

- It constructs a new LinkedHashMap with no entries in it with the specified initial capacity and the specified load factor.
- E.g.

```
import java.util.*;
class LoadFactorArgumentConstructor
{
    public static void main(String[] args)
    {
            LinkedHashMap<Integer,String> map=new LinkedHashMap<Integer,String>(10,0.5f);
            map.put(1,"Ramesh");
            map.put(2,"Suresh");
            map.put(3,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
    }
}
```

**Output:-**

{1=Ramesh, 2=Suresh, 3=Mukesh, 4=Ganesh}

## ★ TreeMap:-

- TreeMap is an implementation class of NavigableMap and also implements other interfaces such as SortedMap and Map.
- It was introduced in JDK 1.2 version and derived in java.util package.
- The internal implementation of TreeMap is "**Red Black Self Balanced Binary Tree**" which store the data / elements in natural order.
- If we want the customize sorting order of elements we can use Comparator.
- TreeMap doesn't allow null key insertion but however multiple null values can be inserted.
- TreeMap provides O(logn) Time Complexity.
- We cannot store heterogeneous data in it as it performs comparison for sorting of elements.
- Its not synchronized because of which its not thread safe.
- TreeMap have default initial capacity of 16.

### ★ Constructors of TreeMap:-

i. public TreeMap()
ii. public TreeMap(Map map)
iii. public TreeMap(Comparator c)

### i. public TreeMap():-

- It constructs a new empty TreeMap using the natural ordering of its key.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
    public static void main(String[] args)
    {
            TreeMap<Integer,String> map = new TreeMap<Integer,String>();
            map.put(4,"ABC");
            map.put(2,"ABC");
            map.put(5,"ABC");
            map.put(3,"ABC");
            map.put(7,"ABC");
            map.put(1,"ABC");
            map.put(6,"ABC");
            for(Map.Entry<Integer,String> entry : map.entrySet())
            {
                    System.out.println(entry);
            }
    }
}
```

**Output:-**

```
1=ABC
2=ABC
3=ABC
4=ABC
5=ABC
6=ABC
7=ABC
```

**ii. public TreeMap(Map map):-**

- It constructs a new TreeMap containing all the elements specified in the Map.
- E.g.

```
import java.util.*;
class MapArgumentConstructor
{
        public static void main(String[] args)
        {
                LinkedHashMap<Integer,String>map1=new LinkedHashMap<Integer,String>();
                 map1.put(4,"ABC");
                 map1.put(2,"ABC");
                 map1.put(5,"ABC");
                 map1.put(3,"ABC");
                 map1.put(7,"ABC");
                 map1.put(1,"ABC");
                 map1.put(6,"ABC");
                 for(Map.Entry<Integer,String> entry : map1.entrySet())
                 {
                         System.out.println(entry);
                 }
                 System.out.println();
                 TreeMap<Integer,String> map2 = new TreeMap<Integer,String>(map1);
                 for(Map.Entry<Integer,String> entry : map2.entrySet())
                 {
                         System.out.println(entry);
                 }
        }
}
```

**Output:-**

```
4=ABC
2=ABC
5=ABC
3=ABC
7=ABC
1=ABC
6=ABC

1=ABC
2=ABC
3=ABC
4=ABC
5=ABC
6=ABC
7=ABC
```

**iii. public TreeMap(Comparator c):-**

- It constructs an empty TreeMap which stores its element according to specified Comparator.
- E.g.

```
import java.util.*;
class sortDesc implements Comparator<String>
{
    @Override
    public int compare(String o1,String o2)
    {
            return o2.compareTo(o1);
    }
}
class ComparatorArgumentConstructor
{
    public static void main(String[] args)
    {
            TreeMap<String,Double> map = new TreeMap<String,Double>(new sortDesc());
            map.put("Marker",30.0);
            map.put("Duster",100.0);
            map.put("Board",1000.0);
            map.put("Pen",20.0);
            System.out.println();
            System.out.println("List of products : ");
            for(String key : map.keySet())
            {
                    System.out.println(key+" : "+map.get(key));
            }
    }
}
```

**Output:-**

```
List of products :
Pen : 20.0
Marker : 30.0
Duster : 100.0
Board : 1000.0
```

★ **Methods of TreeMap:-**

i. public K lowerKey(K key):-
- This method returns the smallest element then the specified key.

ii. public K higherKey(K key):-
- This method returns the least key strictly greater than the specified key.
- If there is no such key it returns null.

iii. public K floorKey(K key):-
- This method returns the greatest key less than or equal to the given key or else return null if there is no such key.

iv. public K ceilingKey(K key):-
- This method returns the least key i.e. greater than or equals to the given key or else returns null if there is no such key.

★ **WAJP to fing the frequency of elements and store it inside a TreeMap.**
→

```
import java.util.*;
class Example
{
  public static void main(String[] args)
  {
        ArrayList<Integer> list = new ArrayList<Integer>();
        for(int i=1;i<=20;i++)
              list.add((int)(Math.random()*10));
        System.out.println(list);
        HashSet<Integer> set = new HashSet<Integer>(list);
        TreeMap<Integer,Integer> map = new TreeMap<Integer,Integer>();
        for(Integer ele : set)
        {
              map.put(ele,Collections.frequency(list,ele));
        }
        for(Map.Entry<Integer,Integer> entry : map.entrySet())
        {
              System.out.println(entry);
        }
  }
}
```

**Output:-**

```
[1, 8, 3, 2, 7, 9, 0, 9, 8, 6, 7, 3, 2, 2, 0, 6, 4, 0, 1, 4]
0=3
1=2
2=3
3=2
4=2
6=2
7=2
8=2
9=2
```

## ★ Hashtable:-

- Hashtable is an implementation class of Map interface which stores the data in a form of key and value pair.
- It inherits the abstract class Dictionary and has a child class Properties.
- Hashtable was introduced in JDK 1.0 version and derived in java.util package.
- It is one of the Legacy classes.
- Hashtable contains all unique elements.
- We cannot store a single null key in it as well as null values.
- It is synchronized because of which it is thread safe.
- The Hashtable is an array of list where each list is a bucket and elements which are stored in it uses the hashing process.
- The position of the buckets is identified by calling the hashCode().
- The default initial capacity of Hashtable is 11 and with the default load factor of 0.75.
- Hashtable does not follows the conventions as it was introduced in JDK 1.0 version and conventions are introduced in JDK 1.2 version.

### ★ Constructors of Hashtable:-

i. public Hashtable()
ii. public Hashtable(Map map)
iii. public Hashtable(int initialCapacity)
iv. public Hashtable(int initialCapacity, float loadFactor)

### i. public Hashtable():-

- It constructs a new Hashtable with no mappings in it with the default initial capacity of 11 and the load factor of 0.75.
- E.g.

```
import java.util.*;
class NoArgumentConstructor
{
    public static void main(String[] args)
    {
            Hashtable<Integer,String> table = new Hashtable<Integer,String>();
            table.put(2,"Ramesh");
            table.put(3,"Suresh");
            table.put(1,"Mukesh");
            table.put(4,"Ganesh");
            System.out.println(table);
            for(Map.Entry<Integer,String> entry : table.entrySet())
            {
                    System.out.println(entry);
            }
    }
}
```

**Output:-**

```
{4=Ganesh, 3=Suresh, 2=Ramesh, 1=Mukesh}
4=Ganesh
3=Suresh
2=Ramesh
1=Mukesh
```

**ii. public Hashtable(Map map) :-**

- It constructs a new Hashtable which contains the mappings from the specified map with the initial capacity equivalent to the number of mappings with default load factor of 0.75.
- E.g.

```
import java.util.*;
class MapArgumentConstructor
{
    public static void main(String[] args)
    {
            LinkedHashMap<Integer,String> map = new LinkedHashMap<Integer,String>();
            map.put(2,"Ramesh");
            map.put(3,"Suresh");
            map.put(1,"Mukesh");
            map.put(4,"Ganesh");
            System.out.println(map);
            Hashtable<Integer,String> table = new Hashtable<Integer,String>(map);
            System.out.println(table);
    }
}
```

**Output:-**

```
{2=Ramesh, 3=Suresh, 1=Mukesh, 4=Ganesh}
{4=Ganesh, 3=Suresh, 2=Ramesh, 1=Mukesh}
```

**iii. public Hashtable(int initialCapacity):-**

- It constructs an empty Hashtable with no mappings in it with the specified initial capacity and with the default load factor 0.75.
- E.g.

```
import java.util.*;
class InitialCapacityArgumentConstructor
{
    public static void main(String[] args)
    {
            Hashtable<Integer,String> table = new Hashtable<Integer,String>(10);
            table.put(2,"Ramesh");
            table.put(3,"Suresh");
            table.put(1,"Mukesh");
            table.put(4,"Ganesh");
            System.out.println(table);
            for(Map.Entry<Integer,String> entry : table.entrySet())
            {
                    System.out.println(entry);
            }
    }
}
```

**Output:-**

```
{4=Ganesh, 3=Suresh, 2=Ramesh, 1=Mukesh}
4=Ganesh
3=Suresh
2=Ramesh
1=Mukesh
```

**iv. public Hashtable(int initialCapacity, float loadFactor):-**

- It creates a new empty Hashtable with no mappings in it with the specified initial capacity and with specified load factor.
- E.g.

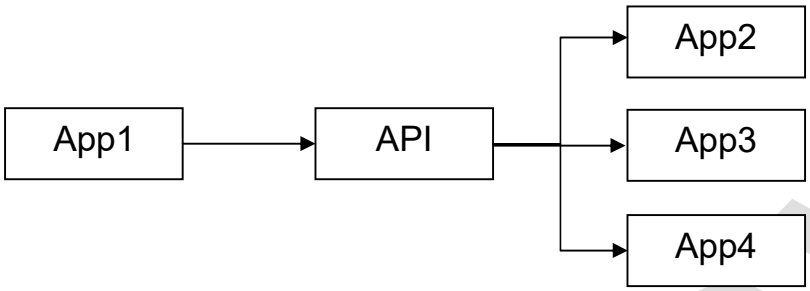
```
import java.util.*;
class LoadFactorArgumentConstructor
{
    public static void main(String[] args)
    {
            Hashtable<Integer,String> table = new Hashtable<Integer,String>(10,0.5f);
            table.put(2,"Ramesh");
            table.put(3,"Suresh");
            table.put(1,"Mukesh");
            table.put(4,"Ganesh");
            System.out.println(table);
            for(Map.Entry<Integer,String> entry : table.entrySet())
            {
                    System.out.println(entry);
            }
    }
}
```

**Output:-**

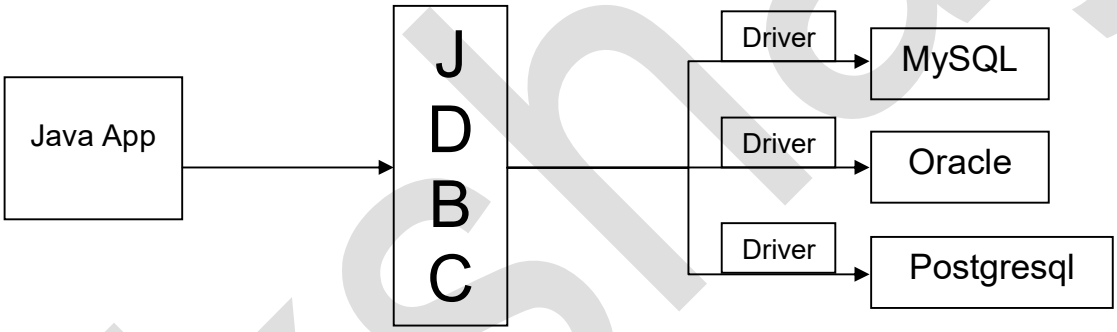
```
{4=Ganesh, 3=Suresh, 2=Ramesh, 1=Mukesh}
4=Ganesh
3=Suresh
2=Ramesh
1=Mukesh
```

# ★ JDBC (Java Database Connectivity):-

- JDBC is an API which is used to connect java application with the database server.
- API (Application Programming Interface) is used to communicate between two application.




- If we want to communicate with particular database it is mandatory to take driver software from respective database server.




★ **Componenets of JDBC :-**
i. Interfaces
ii. Helper Classes
iii. Implementation Classes

★ **Helper Classes :-**
- Helper class is class which is used to create implementation object for interface.
- E.g. DriverManager

★ **Implementation Classes :-**
- The class which gives implementation for abstract methods which are present in the interface.

★ **Driver :-**
- Driver is a software which contains implementation classes needed for interfaces present in the JDBC API.

★ **Steps of JDBC:-**
i. Load / Register the driver
ii. Create connection
iii. Create statement
iv. Execute query
v. Close connection

### i. Load / Register the Driver:-

- In this step we have to load driver class into JVM memory.
- We can do this step by step using forName().

★ **forName():-**

- It is a static method which is present in the class with the name Class itself.
- Class is present in the java.lang package.
- This method will take fully qualified classname of driver in the String format.
- Whenever we use this method it will throw ClassNotFoundException (CheckedException).

★ **Fully qualified name of database driver:-**

#### i. Postgre Driver:-

- org.postgresql.Driver
- e.g.

```
try{
        Class.forName("org.postgresql.Driver");
}
catch(ClassNotFoundException e){
        e.printStackTrace();
}
```

#### ii. Oracle Driver:-

- oracle.jdbc.driver.OracleDriver
- e.g.

```
try{
        Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException e){
        e.printStackTrace();
}
```

#### iii. MySql Driver:-

- com.mysql.jdbc.Driver
- e.g.

```
try{
        Class.forName("com.mysql.jdbc.Driver");
}
catch(ClassNotFoundException e){
        e.printStackTrace();
}
```

### ii. Create Connection:-

- We can create connection by providing " url, username, password " of database.
- To provide these three information we are having getConnection() method.

★ **getConnection():-**

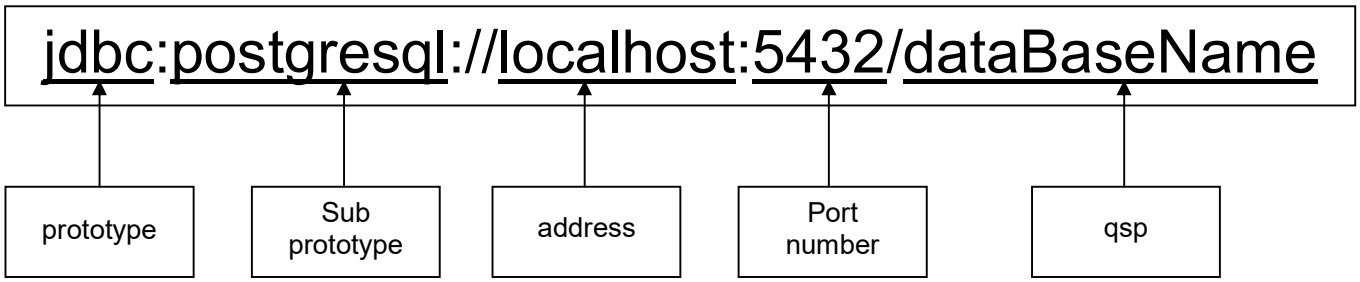
- It is a static method present in DriverManager class.
- This method will take url, username and password in String format.
- DriverManager class is present in java.sql package.
- getConnection() will return us Connection object.
- This method throws SQLException (CheckedException).
- Syntax :-

```
Connection con = DriverManager.getConnection("url","username","password");
```

★ **Connection interface :-**

- Connection is an interface which is present in java.sql.
- The helper class DriverManager will help Connection interface to create implementation object.
- i.e. Upcasting

★ **URL for postgresql :-**




★ **Username of postgres :-**

- postgres

★ **Password for postgres :-**

- root

**Example :-**

```
class Demo
{
	public static void main(String[] args)
	{
		try{
			Class.forName("org.postgresql.Driver");
			Connection con = DriverManager.getConnection
					("jdbc:postgresql://localhost:5432/qsp","postgres","root");

			System.out.println(con);
		}
		catch(ClassNotFoundException | SQLException e){
			e.printStackTrace();
		}
	}
}
```

**iii. Create Statement :-**

- This step is used to execute SQL queries, it will carry SQL query from java application to database server.
- We can create this statement by using createStatement() method.
- createStatement() is non-static method present in Connection object.
- This createStatement() method will returns Statement object.
- Statement is an interface which is present in the java.sql package.
- E.g.

```
Statement stmt = con.createConnection();
```

**iv. Execute Query:-**

- This step is used to execute the SQL query and to execute the query we are having three methods.
  i. execute("SQL")
  ii. executeUpdate("DML")
  iii. executeQuery("DQL")

i. **public boolean execute("SQL") :-**
- It is non-static method which is present in the Statement object.
- By using this method we can execute any type of query (DML, DQL, DDL).
- The return type this method is boolean.
- It will return true when we execute DQL query (SELECT query).
- It will return false for DML query (INSERT, UPDATE, DELETE).
- E.g.

```
stmt.execute("insert into Student values(1,'Akshay')");
```

### v. Close Connection :-

- We can close the Connection by using close() method present in the Connection object.
- It is not mandatory but database vendors will recommended to close the connection for security purpose and to avoid data leakage.
- E.g.

```
con.close();
```

**Example to insert the data into the database:-**

```
class Demo
{
	public static void main(String[] args)
	{
		try{
			Class.forName("org.postgresql.Driver");
			Connection con = DriverManager.getConnection
					("jdbc:postgresql://localhost:5432/qsp","postgres","root");
			Statement stmt = con.createStatement();
			stmt.execute("insert into Student values(1,'Akshay')");
			System.out.println(con);
			con.close();
		}
		catch(ClassNotFoundException | SQLException e){
			e.printStackTrace();
		}
	}
}
```

**When you are creating a project**
**Package name should be like this :-**

**Syntax:-**
domain.companyName

**E.g.**
com.qsp

## ★ ResultSet:-

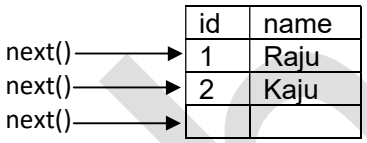
- ResultSet is an interface present in java.sql package.
- ResultSet is used to fetch the data from database.
- We can create ResultSet object by using getResultSet() method present in Statement object.
- Methods of ResultSet :-

### i. next():-

- It is a non-static method present in ResultSet object.
- It is used to check whether next row is present or not.
- And also it will shift the cursor from one row to another row.
- If next row is present it will return true else it will return false.

### ii. getX(String columnName):-

- It is non-static method present in ResultSet object.
- It is used to fetch the data from particular column.
- X indicates datatype of column.
- This method will take column name and returns data which is present in that column.




**Example for retrieving from databse :-**

```
package com.qsp;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Driver {

	public static void main(String[] args) {

		try {
			Class.forName("org.postgresql.Driver");
			Connection con =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/qsp","postgres","root");
			Statement stmt = con.createStatement();
			stmt.execute("select * from student");
			ResultSet rs = stmt.getResultSet();
			while(rs.next()) {
				System.out.println(rs.getInt("id"));
				System.out.println(rs.getString("name"));
			}
			con.close();
		}
		catch (ClassNotFoundException | SQLException e) {
			e.printStackTrace();
		}

	}

}
```