

## **Programming language**

Language that is used to communicate with the computer to perform a specific task is known as a programming language

### **Levels of programming language**

- Low level language
- Assembly level language
- High level language

#### **1. Low level language**

The language that is understandable directly by the computer is known as low level language or machine level language

#### **2. Assembly level language**

The language that is used to communicate with the hardware of the computer such as microprocessor or microcontroller controllers that is known as assembly level language

In assembly level language there are some pre-defined commands developed that is used to perform specific task

Assembly language is understandable to the computer with the help of a translator called as assembler

#### **3. High level language**

The language that is easy to read easy to write and understand and similar to English is known as high level language.

The high-level language is understandable by machine with the help of translator called as compiler or interpreter

The high-level language is further classified into platform dependent and platform independent languages

#### **Platform dependent**

A software is developed in one platform and runs only on the same platform but not in the different platform is known as platform dependent

e.g.: c & c++

### **Q. Why C/C++ language is known as platform dependent languages?**

**Ans:**

- 1) C compiler is one step compiler
- 2) C compiler compiles the same language and generated a file called as .exe file
- 3) This executable file is understandable only by the respective operating system
- 4) This makes C language as platform dependent language

## Platform independent

A software is developed in one platform and runs on all the different platform is called platform independent

### Why Java is a platform independent language?

- 1) Java compiler compiles the Java code and generates a file called as .class file
- 2) The class file can contain an instruction that are written in the form of bytecode
- 3) The bytecode can be executed in any of the operating system that contains JRE
- 4) That is why Java is known as platform independent language but dependent on JRE

### Source file

- The file that is created by the programmer is known as source file
- Source File used for compilation
- The extension of Java source file is **.java**
- The name of the source file should be similar to the class name

### Class file

- The file that is generated by the compiler is known as class file
- Class file is used for execution
- Extension of class file is **.class**
- Class file name is always same as source file
- Class file contents and instructions that written in the form of byte codes

### Bytecode

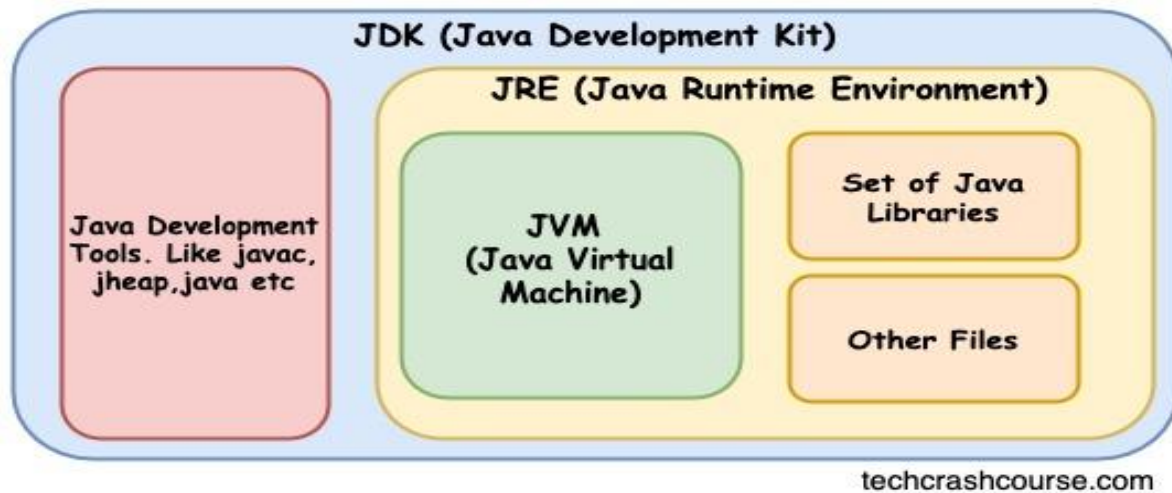
- Byte code is an interpreted language that is neither understandable by programmer nor by the compiler
- It is only understandable by computer that is installed with JRE

Difference b/w compiler & interpreter	
i) Compiler translates the source code to byte code	i) Interpreter translates the byte code to machine code
ii) Compiler verifies the syntax and if there is no syntax error, only then it translates the source code.	ii) Interpreter translates line by line.
iii) It is faster.	iii) It is slower than compiler
iv) It is more efficient	iv) It is less efficient.

## Features of Java

- Simple
- Object oriented
- Platform independent
- Portable
- Robust
- Architecture neutral
- Interpreter language
- High Performance
- Multi-threaded
- Dynamic
- Distributed

## JDK Architecture



### JDK Java Development Kit

- JDK is used to develop the Java application
- It contains compiler, JRE and some development tools

### JRE Java runtime environment

- It is runtime environment where Java byte codes are executed
- It contains class loader, interpreter, some inter inbuilt libraries to run the bytecodes

### JVM Java virtual machine

- It is a virtual machine that is used to execute the byte codes
- It provides runtime environment
- Also used for memory management

### Class loader

- It is used to load the class file from secondary storage device to JRE

### JIT Justin time compiler

- It is used to compile bite codes during runtime to improve the efficiency

## Tokens

- Tokens are the smallest unit of Java program

### Types of tokens

- **Keywords**
- **identifiers**
- **Literals**
- **Operators**

### 1) Keywords

- Keywords are predefined words in Java that are used to perform a specific task
- Keywords should be represented by lower case
- There are 53 keywords in Java
- Being a programmer, we cannot change the functionality of a keyword
- Ex. void, byte, break, char, etc

### 2) Identifiers

The name of Java component is known as Identifiers

- **Rules of an identifier**
  - Identifiers should not start with number
  - Identifier should not contain special characters except \$ and \_
  - Identifiers should not contain space
  - Keywords are not allowed as an identifier

### 3) Literals

- Literals are data or values that are used in Java program
- **Types of literals**
  - Primitive literals
  - Non-primitive literals

## Primitive literals

- Primitive literals are single valued data
- Primitive literals are not treated as object
- Numbers, characters, Booleans are considered as primitive literals
- **Numbers**
  - Numbers are numeric values
  - Integers and floating numbers can be used in Java
- **Characters**
  - Alphabets, numbers and special characters are considered as characters
  - Character should be represented by single quote
  - The length of a character should be one
- **Boolean/ logical values**
  - Boolean are logical values, true and false are considered as Boolean
  - Boolean Should be represented by lower case

## Non-primitive literals

- Non-primitive literals are treated as object in Java
  - String And object references are considered as non-primitive literals
- **String**
    - String is a collection of characters
    - String should be represented by double quotes
    - The length of the string can be any number

## Structure of a Java program

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

## Print statements

1. **Print() method:** Print method is used only to print the data

```
System.out.print("Hello");
```

2. **Println() Method :** Println method is used to print the data and also it prints one new line

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

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

## Datatypes

Data types are used to specify the type of the data to be stored in a variable

- **Type of data types**

- Primitive Data types
- Non-primitive data types

### 1) Primitive data types

- Primitive data types are used to specify the primitive type data such as numbers, characters and Boolean
- Primitive data type is predefined
- Primitive data types are keywords in Java

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

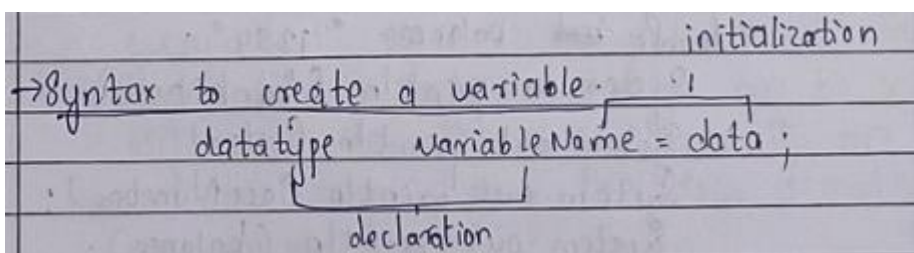
### 2) Non-primitive data type

- Non-Primitive data types are used to specify the non-primitive type such as strings and object reference
- Non-Primitive data types are user defined data types example string, list, set, etc
- **Note:** Every class and interface name in Java is considered as non-primitive data type

- **Variables**

Variable is a container that is used to store the data

### syntax to create a variable



```
class Test {  
    public static void main(String[] args) {  
        short number = 500;  
        double number2 = 70.23;  
        String name = "saurabh";  
        System.out.println(name);  
        System.out.println(number);  
    }  
}
```

## Types of Variables

- Local variable
- Static variable
- Non-Static variable

1) **Local variable** : If a variable is created in any block except the class block is known as local variable

### Characteristics of local variable

- Local variables are local in nature
- Local variables are visible only within the block where it gets created
- Local variables are not initialized with its default values hence before using local variables it should be initialized with the data

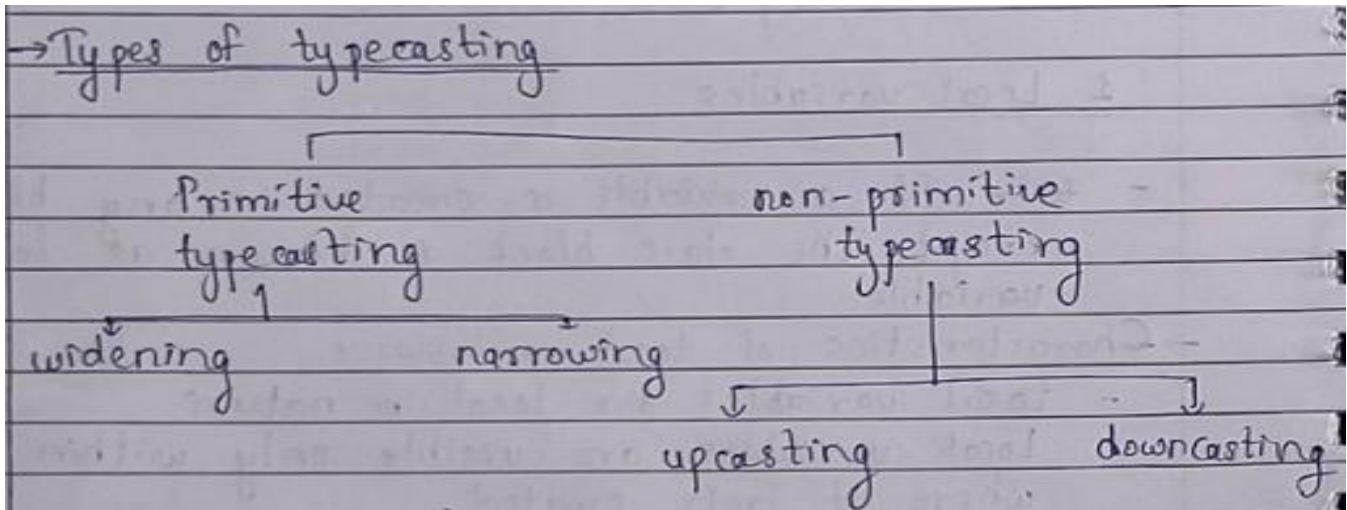
## 2) Static / non-static variables

### Characteristics of static/ Non – static variables

- Static and non-static variables should be created only inside the class block
- The scope of the static and non-static is everywhere since it is global in nature
- Static and non-static variables can be initialized with its default values hence it can be used without initializing the data directly by the programmers

## Typecasting

The process of converting one type of data into another type of data is known as type casting



### Primitive typecasting

The process of converting the primitive type data into another primitive type data is known as primitive typecasting

#### Types of primitive type casting

- Widening
- Narrowing

1) **Widening:** the process of converting lowest range primitive type data to highest range primitive type data is known as Widening

E.g. Byte < (short / char) < int < long < float < double

```
class Test {
    public static void main(String[] args) {
        short num = 30;
        long num1 = num; // widening
        System.out.println(num1);

        int num3 = 75;
        float num4 = num3;

        char ch = 'a';
        long num5 = ch;
        double num6 = ch;
    }
}
```



## 2) Narrowing

- The process of converting high range primitive type data to lowest range primitive type data is known as narrowing
- In narrowing process, there is a chance of getting data loss hence narrowing should be performed explicitly by using typecast operator

**Typecast Operator:** typecast operator is used to convert one type of data to another type of data explicitly

### Syntax:

**Datatype(data)**

### Workflow:

The data passed to the typecast operator is converted to the specified data type

```
class Test {
    public static void main(String[] args) {
        short num = 30;
        long num1 = (long)num; // widening
        System.out.println(num1);

        int num3 = 75;
        float num4 = (float)num3;

        char ch = 'a';
        long num5 = (long)ch;
        double num6 = (double)ch;
        System.out.println(num5);
    }
}
```

## Operators

- Operator is a predefined symbol that is used to perform a specific task
- The operator after performing the task returns data as an output
- Every operator accepts the input or operands

### Type of operators

- Arithmetic operator
- Assignment operator
- Relational / comparison operator
- Logical operator
- Increment/ Decrement operator
- Conditional operator
- Bitwise Operator

#### 1) **Arithmetic operator: + - \* / %**

- Arithmetic operator are used to perform Arithmetic operations
- Arithmetic operator are binary operators

### Result type of arithmetic operators

**Case 1 :** both operands are same type

Byte and byte = int

Short and short = int

Int and int = int

Long and long = long

Float and float = float

Double and double = double

Char and char = int

**Case 2 :** Both operands are different types

If both the operands are different, the result type is always higher range type of operand

Int and int = int

Long and long = long

Char and double = double

Char and byte = int

#### 2) **Relational operator**

Relational comparison operator : >, <, >=, <=, ==, !=

- Relational operators are binary operators
- Result of relational operators is always Boolean type

### 3) Assignment Operator

=, +=, -=, \*=, /=, %=

- Assignment operators are used to assign data to a variable
- The left-hand side of assignment operator should be a variable
- Right hand side of assignment operator should be data

### 4) Logical operator

&& - Logical AND

|| - Logical OR

! - Logical NOT

- Logical operators are used to perform logical operations
- The logical operators accept only Boolean type operands
- The result of logical operators is also Boolean

#### Logical AND

- Logical AND operator returns true only when both inputs are true
- logical AND operator is a binary operator

Op1	Op2	Result
F	F	F
F	T	F
T	F	F
T	T	T

#### Logical OR

- It is a binary operator
- It returns true if any one of the operands is true

Op1	Op2	Result
F	F	F
F	T	T
T	F	T
T	T	T

#### Logical NOT

- It is a Unary operator
- If the operand is true, it returns false and if the operand is false, it returns true

Op1	Result
F	T
T	F

## 5) **Increment operator**

Increment operator is used to increase integer value by 1

### **a. Pre increment**

**Syntax:** ++ variable

In Pre increment the first value is increased by 1 and then it is used

### **a. Post increment**

**Syntax:** variable ++

In Post increment the first value is increased by 1 and then it is used

## 6) **Decrement operator**

- It is a unary operator
- It is used to decrease the value by 1

### **b. Pre decrement**

**Syntax:** -- variable

In Pre **d**ecrement the first value is decreased by 1 and then it is used

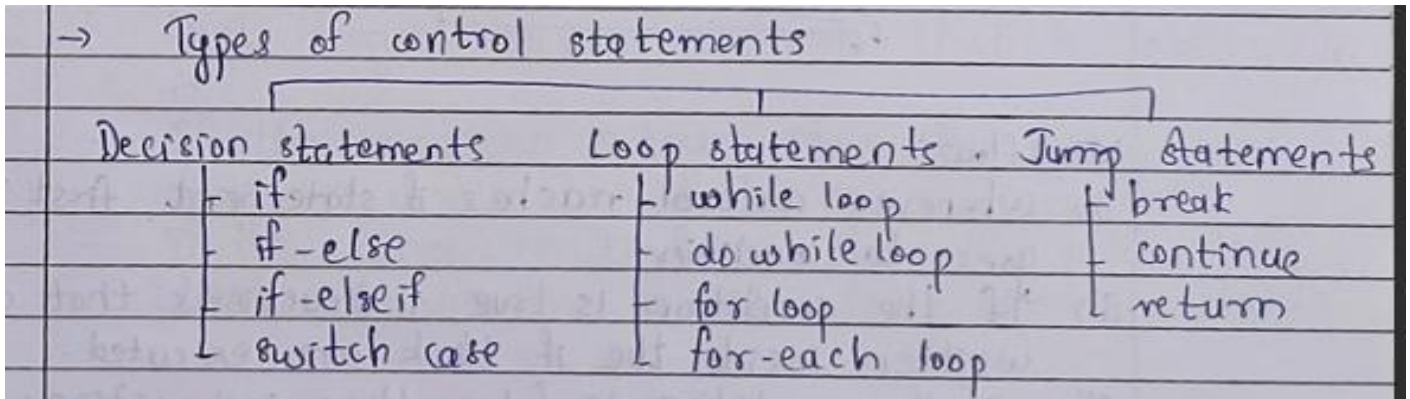
### **b. Post decrement**

**Syntax:** variable --

In Post decrement the first value is decreased by 1 and then it is used

## Control statements

Control statements are used to control the flow of execution



- **Decision statements**

- Decision statement is used to take a decision in Java
- With the help of decision statement a set of instructions are executed or skipped based on the condition

- **Types of decision statements**

- If statement
    - If Else statement
    - If else if statement
    - Switch case

- 1) **If statement**

### Syntax

```
if(condition){  
    // instructions  
}
```

### Workflow

- Whenever control reaches if statement first it goes to condition
- If the condition is true instructions that are written inside the if block is executed
- If the condition is false then the instructions in the if block are skipped

**Example: write Java program to print number only when it is even**

```
class Test {  
    public static void main(String[] args) {  
        int n = 10;  
        if (n % 2 == 0) {  
            System.out.println(n);  
        }  
    }  
}
```

## 2) If else Statement:

### Workflow:

- When the control reaches if statement First it goes to the condition that is written in the if block
- If the condition is true then the instructions are written in the if block are executed
- If the condition is false then the instructions that are written in the else block are executed

**Write a Java program to find out the greatest of two numbers**

```
class Test {  
    public static void main(String[] args) {  
        int n = 5;  
        int m = 10;  
        if (m > n) {  
            System.out.println(m + "is the greatest number");  
        } else {  
            System.out.println(m + "is the smallest number");  
        }  
    }  
}
```

## 3) If else if statement

### Workflow:

- When the control reaches if statement first it goes to the condition that is written in the if block
- If the condition is true then the instructions are written in the if block are executed
- If the condition in the If block is false then the control reaches else if block and it checks the condition and if the condition is true then the else if block is executed
- If the condition is false in the if block and else if block is false then else block is executed

## 4) Switch case :

### Workflow:

- Here data is passed to the switch
- If any of the cases contains the similar value then the instruction in that case is executed
- Along with that the below created cases are also executed
- If none of the cases contains similar value then the default case instructions are executed
- **Note:** To avoid the continuous execution of below cases, break keyword can be used

- **Break keyword:**

- In Java break is a keyword used to exit a loop or switch statement early
- When the break is encountered inside a loop it immediately terminates the loop execution and continues with the next line of code after the loop
- In a switch statement, **break** is used to exit the switch block, preventing fall-through to the next case.
- Without **break**, a loop would continue indefinitely, or a switch would execute all following cases until reaching a **break** or the end of the switch block.
- Using **break** helps control the flow of a program, allowing it to exit loops or switch statements when certain conditions are met.

**Example:**

```
public class Main {  
    public static void main(String[] args) {  
        int day = 3;  
        String dayName;  
  
        switch (day) {  
            case 1:  
                dayName = "Monday";  
                break;  
            case 2:  
                dayName = "Tuesday";  
                break;  
            case 3:  
                dayName = "Wednesday";  
                break;  
            case 4:  
                dayName = "Thursday";  
                break;  
            case 5:  
                dayName = "Friday";  
                break;  
            case 6:  
                dayName = "Saturday";  
                break;  
            case 7:  
                dayName = "Sunday";  
                break;  
            default:  
                dayName = "Invalid day";  
                break;  
        }  
  
        System.out.println("The day is: " + dayName);  
    }  
}
```

## Advantages of switch case

With the help of switch case grouping can be achieved see

```
public class Main {  
    public static void main(String[] args) {  
        int day = 3;  
        String dayName;  
  
        switch (day) {  
            case 1:  
            case 2:  
            case 3:  
            case 4:  
            case 5:  
                dayName = "Weekday";  
                break;  
            case 6:  
            case 7:  
                dayName = "Weekend";  
                break;  
            default:  
                dayName = "Invalid day";  
                break;  
        }  
  
        System.out.println("The day is: " + dayName);  
    }  
}
```

### Note:

- Duplicate cases are not allowed in switch
- Heterogeneous type cases are not allowed
- Long, float, double, Boolean types data are not allowed in switch cases



## Loop statement

Loop statements are used to execute a set of instructions repeatedly

### Types of loop statements

- While loop
- Do while loop
- For loop
- For each loop

#### 1) While Loop

##### Workflow:

- Whenever control reaches while loop first it checks the condition
- Condition is true then the instructions in the loop block are executed
- Once the instructions are executed control goes back to the condition
- Step two and three repeats until the condition became false

```
class Test{
    public static void main(String[] args) {
        int i=1;
        while (i <= 6) {
            System.out.println(i);
            i++;
        }
    }
}
```

#### 2) Do while loop

##### Workflow:

- When you control reaches do while loop first it enters inside the do block and executes the instructions
- Once the instructions in do block are executed control checks the condition
- If the condition is true control goes back to the do block
- Step two and three repeats until condition became false

```
class Test {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println(i);
            i++;
        } while (i <= 6);
    }
}
```

### 3) For loop

#### Workflow:

- Whenever control reaches for loop first it goes to initialization
- After initialization control goes to condition
- If the condition is true instructions in loop block are executed
- Once the instructions are executed control goes to update
- After update control goes back to condition
- Step 3, 4 and 5 are repeated until the condition became false

```
class Test {  
    public static void main(String[] args) {  
  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

## Methods

Methods are a block or set of instructions that are used to perform a special task

### Purpose of main method:

- Main method starts the execution
- The flow of execution
- End the execution of the program

Syntax: `public static void main(String[] args)`

#### i. Access modifier

Access modifier is used to modify the accessibility

##### Levels of access modifiers

- Public
- Protected
- Default
- Private

#### ii. Modifiers

Modifier are used to modify the behaviour

##### Types of modifiers

- Static
- Non-Static
- abstract
- Final
- synchronised
- transient
- Volatile

#### iii. Return type

Return type is a data type that specifies the type of data returned by the method

- Primitive datatypes
- Non - Primitive datatypes
- Void

**Void:** Void is a data type that is used as a return type when the method does not return any data

#### iv. **Formal Arguments**

The variables that are declared in a method signature is known as formal arguments

##### **Types of methods based on formal arguments**

- No argument method
- Parameterized method

**No argument method** :The method without formal argument is known as no argument method

**Parameterized method** : The method with formal argument is known as parameterized method

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

**Note:** Methods will execute only when they are called

##### **Method call flow:**

- Whenever the method is called the execution of the calling method is paused and control is given to the called method
- Once the called method got the control it starts executing
- Once the execution of called method is completed the control is given back to the caller
- Once the caller got the control it resumes its execution

##### **Return Type:**

- Every method after performing a task can return a data back to the caller
- Return type specifies the data type return by the caller

##### **Return:**

- It is a key word
- It is used to return data back to the caller
- Return should be used at the end of the block
- Once the return is executed the execution of current method is stopped and the controller is given back to the caller
- Return can be also used without passing the data

- The purpose of using return without passing data is just to stop the execution of the current method
- If the return type is void then return can be used without passing data

```
class Test {  
    public static void main(String[] args) {  
  
        int res = add(10, 20);  
        System.out.println(res);  
    }  
  
    public static int add(int a, int b) {  
        int res = a + b;  
        return res;  
    }  
}
```

### **Dynamic Read**

The process of reading an input from the user during runtime is known as dynamic read

Steps to Achieve dynamic read

- Import Scanner class  
    Import java.util.scanner;
- Create object for scanner class  
    Scanner sc = new Scanner(System.in);
- Call the method  
    Object\_reference.method\_name();

## → Methods scanner class:

Data	Method name	Return type
byte	nextByte()	byte
short	nextShort()	short
int	nextInt()	int
long	nextLong()	long
float	nextFloat()	float
double	nextDouble()	double
char	nextChar()	char
boolean	nextBoolean()	boolean
string └─ single multi	next()	string
	nextLine()	string

```
import java.util.Scanner;

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

        Scanner sc = new Scanner(System.in);

        int x = sc.nextInt();
        int y = sc.nextInt();

        int res = add(x, y);
        System.out.println(res);
    }

    public static int add(int a, int b) {
        int res = a + b;
        return res;
    }
}
```

## Object oriented programming

1. Static and static members
2. Non-static and non-static members
3. Class - object – constructor
4. Principles of oops
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

- **Static**

- It is a keyword
- It is also known as non-access modifier
- Static keyword is applicable for variables, methods and blocks

- **Characteristics of static**

- Static member can be access with the help of class name
- Only one copy of static members exists in a class
- Object creation is not mandatory to access the static members
- The memory for static members is allocated inside the class static area that is in JRE

### Static Members

- **Static variables**
- **Static methods**
- **Static Block**

### 1) Static Variables

If variable is prefixed with the static keyword and created in the class block then it is known as static variables

#### Characteristics

- Static variables are initialized with its default values
- Only one copy of static variables exists in a class
- Static variables can be accessed with the help of class name
- Object creation is not mandatory to access the static variables
- The memory for static variables is allocated inside the class static area
- Static and local variable name can be similar
- **Note :** If static and local variables having the similar name then priority goes for local variables in this case it is mandatory to prefer the static variables by its class name

```
import java.util.Scanner;

class Test {
    static String name = "Saurabh";
    public static void main(String[] args) {
        System.out.println(name);
        name = "Omkar";
        m1();
    }

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

## 2) Static methods

If a method is Prefix with static then it is known as static method

### Characteristics

- Only one copy of static method is existing in a class.
- Static method can only be accessed with the help of class name.
- Object creation is not mandatory for accessing the static methods.
- Static methods can have any access modifier (public, private, protected, or default), just like instance methods. However, they are commonly used with public access for accessibility.

## 3) Static Block:

A static block in Java is a block of code enclosed within curly braces and preceded by the **static** keyword.

### Characteristics:

- Static blocks are executed only once when the class is loaded into memory, before the execution of the main method or the creation of any objects of the class.
- If there are multiple static blocks in a class, they are executed in the order they appear in the class file(top to bottom).
- Static blocks can have any access modifier (public, private, protected, or default), but they are commonly used with the default access modifier since they are accessed internally by the class.
- In a class any number of static blocks are created.
- Static blocks are non-callable members.



## **Non – Static**

- If a member of a class is not prefixed with the static keyword, then it is known as non-static members
- Non-Static members are known as instance members

### **Non-static members**

- Non-static variables
- Non-Static methods
- Non-static blocks
- Non-Static constructors

#### **Characteristics:**

- 1) Non-static members belong to an object.
- 2) For every object one copy of non-static members are provided.
- 3) Non-static members are accessed with the help of object reference
- 4) Object creation is mandatory to access non-static members
- 5) Memory for non-static members is allocated inside heap area.

#### **1) Non-Static Variables:**

A non-static variable in Java is a variable that is associated with individual instances (objects) of a class. Here are its characteristics:

- 1) Non-static variables are initialized with its default value.
- 2) For every object one copy of non-static variable is provided
- 3) Non-static variable is accessed with the help of object reference
- 4) Object creation is mandatory to access the non-static variables
- 5) The memory for non-static variables is allocated inside heap area

#### **2) Non-static Methods**

Non-static methods are those methods which are not prefixed with static keyword.

#### **Characteristics:**

- 1) Non-statics methods belong to an object.
- 2) For every object one copy of non-static method is provided
- 3) Non-static method is accessed only by using object reference
- 4) Object creation is mandatory to access the non-static methods

#### **3) Non-Static blocks**

If a block is not Prefix with static and created in a class block then it is known as non-static block or initializer or instance block

#### **Characteristics:**

- 1) Non-static blocks are executed once during object loading process
- 2) In a class any number of non-static blocks can be created
- 3) If class contains multiple non-static blocks it executes from top to bottom order for every object creation

- 4) Non-static blocks do not have excess modifiers like public private unprotected and cannot be Invoke directly. They are invoking automatically during object creation

## **Object oriented programming**

Object-oriented programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes. Java is an object-oriented programming language. In Java the real-world object is created in software world and states and behaviour of an object are also represented java follows the principle of oops

Encapsulation, Abstraction, polymorphism, inheritance

### **Class:**

- 1) A class in object-oriented programming serves as a blueprint or template for creating objects.
- 2) It defines the attributes (data fields) and behaviours (methods) that objects of that class will have.
- 3) Classes encapsulate data and behaviours together, providing a clear structure for organizing code.
- 4) They enable code reuse by allowing multiple objects to be created based on the same blueprint.
- 5) Classes promote modularity and maintainability by breaking down complex systems into smaller, manageable components.

### **Object:**

- 1) An object is a concrete instance of a class in object-oriented programming.
- 2) It represents a specific entity or concept and has its own state and behaviour.
- 3) Objects encapsulate data (attributes) and behaviour (methods) related to that entity.
- 4) Objects are created from a class blueprint and exist in memory during program execution.
- 5) Interactions between objects allow for the modelling of real-world scenarios and the implementation of complex systems.

### **Blueprint:**

- 1) Before creating an object blueprint of an object is necessary
- 2) Blueprint provides specifications of an object in Java blueprints means Class
- 3) In the class the states and behavior of an object represented with the help of non-static variables and methods

### **Object Creation:**

- 1) New keyword creates the block of memory in the heap area
- 2) New keyword calls the constructor
- 3) Constructor provides states and behavior for an object
- 4) Once the constructor is executed new keyword returns the address of the object

### **Object Reference:**

- 1) Every object has a unique identification known as object reference
- 2) With the help of Object reference the object is identified and accessed
- 3) Object reference is non-primitive data
- 4) The type of object reference is always same as the class name type

```
public class Marker {
    String color;
    String brand;

    public void write(){
        System.out.println("Marker Used");
    }

    public static void main(String[] args) {
        Marker BlackCamlin = new Marker();
        BlackCamlin.brand = "Camlin";
        BlackCamlin.color = "black";

        Marker Redcello = new Marker();
        Redcello.brand = "Cello";
        Redcello.color = "red";

        Marker Greencamlin = new Marker();
        Greencamlin.color = "green";
        Greencamlin.brand = "Camlin";

        System.out.println(BlackCamlin.color);
        System.out.println(BlackCamlin.brand);

        System.out.println(Greencamlin.color);
        System.out.println(Greencamlin.brand);

        System.out.println(Redcello.color);
        System.out.println(Redcello.brand);

        Redcello.write();
        BlackCamlin.write();
        Greencamlin.write();
    }
}
```

### **This Keyword:**

- 1) This is the keyword
- 2) It is also a special non-static variable i.e. It is also present in all the objects
- 3) This keyword is used to store the address of current object
- 4) Whenever local and non-static variables are similar, non-static variables are referred by this keyword

```
public class Person {
    private String name;

    // Constructor
    public Person(String name) {
        this.name = name; // "this" is used to differentiate between the instance
variable and the parameter with the same name
    }

    // Method to print the name
    public void printName() {
        System.out.println("Name: " + this.name); // "this" is used to refer to the
instance variable name
    }

    public static void main(String[] args) {
        Person person1 = new Person("Alice");
        person1.printName(); // Output: Name: Alice
    }
}
```

## Constructor:

A constructor in Java is a special type of method that is automatically called when an object of a class is initialized. It has the same name as the class and it is used to initialize the newly created object.

### Purpose

- 1) The primary purpose of a constructor is to initialize the newly created object by assigning initial values to its instance variables.
- 2) A constructor is automatically invoked when an object of the class is created using the `new` keyword.
- 3) Constructors do not have a return type, not even `void`. They implicitly return an instance of the class.
- 4) Like methods, constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists.
- 5) This allows flexibility in object creation.
- 6) If no constructor is defined by the programmer, Java provides a default constructor which initializes the instance variables to their default values, for example, zero for numeric types and `null` for object references.

### Types of constructors:

A constructor in Java is a special type of method that is automatically called when an object of the class is initialized. It has the same name as the class and it is used to initialize the newly created object.

#### There are mainly two types of constructors

**default constructor:** A default constructor is a constructor that is automatically provided by Java. If no constructor is explicitly defined in a class, it initializes the object with the default values. The default constructor has no parameters.

**Parameterized constructor:** A parameterized constructor is a constructor with parameters. It allows you to initialize the object with specific values passed as arguments when the object is created. A parameterized constructor provides flexibility in object initialization.

### Purpose of constructor:

- 1) **Initialization:** Constructors are used to initialize newly created objects. They assign initial values to the object's instance variables, ensuring that the object is in a valid state when it is created.
- 2) **Automatic Invocation:** Constructors are automatically called when an object of a class is created using the **`new`** keyword. This ensures that the object is properly initialized before it is used.
- 3) **Object Creation:** Constructors play a crucial role in the object creation process, providing a mechanism to set up the initial state of objects based on the parameters passed during instantiation.
- 4) **Flexibility:** Constructors allow for flexibility in object initialization by supporting parameterized constructors, constructor overloading, and custom initialization logic.

## Constructor Overloading :

- 1) **Definition:** Constructor overloading in Java refers to the practice of defining multiple constructors within a class, each with a different parameter list. This allows for the creation of objects using different initialization parameters, providing flexibility in object creation.
- 2) **Purpose:** The purpose of constructor overloading is to provide multiple ways to initialize objects of a class based on different sets of parameters. It allows for flexibility in object creation and accommodates various initialization scenarios.
- 3) **Name:** The name of constructors in Java is the same as the name of the class.
- 4) **Information:** Constructors in Java do not have a return type, not even **void**, and are invoked implicitly when an object is created using the **new** keyword.
- 5) **Default Constructor:** If no constructor is explicitly defined in a class, Java provides a default constructor. This default constructor initializes the object with default values (e.g., **0** for numeric types, **null** for object references).
- 6) **New Keyword:** The **new** keyword is used in Java to create objects of a class. It allocates memory for the object and invokes the appropriate constructor to initialize the object.
- 7) **Constructor Overloading:** Constructor overloading allows a class to have multiple constructors with different parameter lists. Java determines which constructor to invoke based on the arguments provided during object creation.
- 8) **Purpose of Constructor Overloading:** Constructor overloading provides flexibility in object initialization by allowing objects to be created using different sets of parameters. It enables the creation of objects with custom initialization logic tailored to specific use cases.

```
class AppForm {
    String name;
    String address;
    long cno;
    long ac_no;

    public AppForm(String name, String address, long cno, long ac_no) {
        this.name = name;
        this.address = address;
        this.cno = cno;
        this.ac_no = ac_no;
    }

    public AppForm(String name, String address, long cno) {
        this.name = name;
        this.address = address;
        this.cno = cno;
    }

    public AppForm(String name) {
        this.name = name;
    }

    public void Details() {
        System.out.println(name);
        System.out.println(address);
        System.out.println(cno);
        System.out.println(ac_no);
    }
}
```

```
public static void main(String[] args) {  
    AppForm ap = new AppForm("Saurabh", "phursungi", 7058549525L, 784574L);  
    AppForm ap1 = new AppForm("omkar", "Hadapsar", 7720978876L);  
    AppForm ap2 = new AppForm("Vikas");  
  
    ap.Details();  
    ap1.Details();  
    ap2.Details();  
}  
}
```

### Copy constructor:

**Definition:** A copy constructor is a special type of constructor in object-oriented programming that initializes a new object as a copy of an existing object of the same class. It takes an object of the same class as a parameter and creates a new object with the same state as the passed object.

**Purpose:** The purpose of a copy constructor is to create a new object that is a deep copy of an existing object, allowing for the duplication of object state. This is useful when you need to create a new object with the same data as an existing object without affecting the original object.

**Name:** The name of a copy constructor is the same as the name of the class.

#### Information:

- Copy constructors are typically used to perform deep copies, meaning that the contents (attributes) of the original object are duplicated into the new object.
- They are useful when you need to create a new object with the same state as an existing object, ensuring that changes made to one object do not affect the other.

**New Keyword:** The **new** keyword is used in Java to create objects of a class. Copy constructors are invoked explicitly by passing an existing object as an argument during object creation.

```
class Camera {
    String brand;
    int price;

    public Camera(String brand) {
        this.brand = brand;
    }

    public Camera(String brand, int price) {
        this.brand = brand;
        this.price = price;
    }

    public Camera(Camera c) {
        this.brand = c.brand;
        this.price = c.price;
    }

    public void takephoto() {
        System.out.println("photo clicked");
    }

    public void details() {
        System.out.println(brand);
        System.out.println(price);
    }

    public static void main(String[] args) {

        Camera c1 = new Camera("canon", 89000);
        Camera c2 = new Camera(c1);

        c1.details();
        c2.details();

    }
}
```



## Constructor chaining and how to achieve it?

Constructor chaining in Java refers to the process of calling one constructor from another constructor within the same class or from a superclass constructor. This allows for code reuse and ensures that initialization logic defined in one constructor can be reused by other constructors.

To achieve constructor chaining, you can use the `this()` keyword to call another constructor from within a constructor in the same class, or use the `super()` keyword to call a constructor from the superclass.

### Rule to use `this ()` call statement:

- 1) This `call()` statement should be used only in a constructor
- 2) This `call()` statement should be used as first instruction in the constructor
- 3) Only one `this()` call statement should be allowed in one constructor
- 4) If there are N number of constructors then `this()` call statement should be used in N-1 constructor
- 5) Calling the constructor by itself is not allowed

```
public class Chaining {
    String name;
    String color;
    float price;

    public Chaining(String name) {
        this.name = name;
    }

    public Chaining(String name, String color) {
        this(name);
        this.color = color;
    }

    public Chaining(String name, String color, float price) {
        this(name, color);
        this.price = price;
    }

    public void details() {
        System.out.println(name);
        System.out.println(color);
        System.out.println(price);
    }

    public static void main(String[] args) {
        Chaining c = new Chaining("Saurabh");
        c.details();

        Chaining c1 = new Chaining("Omkar", "red");
        c1.details();
    }
}
```

### Rule to use super () call statement:

- 1) Super() call statement should use only in constructor
- 2) Super() call statement should be the first instruction of the constructor
- 3) In a constructor either this call statement or Super() call statement is allowed
- 4) In constructor chaining at least if there are N number of constructors then at least one constructor should have super call statement

```
class App {
    String name;
    double size;

    public App(String name, double size) {
        this.name = name;
        this.size = size;
    }
}

class Instagram extends App {
    String userId;
    String pass;

    public Instagram(String name, double size, String userId, String pass) {
        super(name, size);
        this.userId = userId;
        this.pass = pass;
    }
}

class Program {
    public static void main(String[] args) {
        Instagram i = new Instagram("Instagram", 85.25, "user", "pass");
        System.out.println(i.name);
        System.out.println(i.size);
        System.out.println(i.userId);
        System.out.println(i.pass);
    }
}
```

**note** if the programmer does not add this call statement and super call statement in a constructor then the compiler adds super call statement by default

## **Encapsulation**

The process of binding data members and behaviours together is known as encapsulation

By using encapsulation, data hiding can be achieved

## **Data hiding**

The process of restricting the direct access and providing indirect access is known as data hiding

### **Advantages:**

- By hiding the data, validation and verification is achieved
- By using private keyword data hiding is achievable

## **Private**

- It is a keyword
- It is also known as access modifier
- It is applicable for variables, methods and constructors
- Private members are accessible only within the class
- It is also known as class level modifiers

## **Indirect access**

- Once the data hidden, indirect access for the hidden data can be provided
- Readable and modifiable access can be provided directly for the hidden data
- Indirect access can be provided by using the method known as getter and setter method

## **Getter method**

- Getter methods are used to provide the readable access for the private data
- Getter method name starts from get

## **Setter Method**

- Setter method is used to provide modifiable access for the private data
- Setter method name starts from set

By using Getter and Setter method the following indirect access can be provided

- Only readable
- Only modifiable
- Both readable and modifiable
- Neither Readable nor modifiable

```

class CreditCard {
    private long cno;
    private int cvv;
    private double limit;
    private int pin;

    public CreditCard(long cno,int cvv, double limit, int pin){
        this.cno = cno;
        this.cvv = cvv;
        this.limit = limit;
        this.pin = pin;
    }

    public long getcno(){
        return cno;
    }

    public long getcvv(){
        return cvv;
    }

    public void setLimit(double limit){
        this.limit = limit;
    }

    public static void main(String[] args) {
        CreditCard c = new CreditCard(7545788, 789, 78595.5, 5698);

        System.out.println(c.getcno());
        System.out.println(c.getcvv());
    }
}

```

### **Tightly encapsulated class**

If all the data members of class are private, then it is known as tightly encapsulated class

### **Singleton class**

A Class ensures only one object or instance by itself is known as Singleton class

The Singleton class should have a private constructor so that object creation outside the class is not possible

There should be a static variable which contains instance of itself and there should be a static method which returns the instance of the same class that is available in static variable

```
public class Singleton {
    // Static variable to hold the single instance of the class
    private static Singleton instance;

    // Private constructor to prevent instantiation from outside
    private Singleton() {
        // Constructor logic, if any
    }

    // Static method to get the single instance of the class
    public static Singleton getInstance() {
        if (instance == null) {
            // If instance is null, create a new instance
            instance = new Singleton();
        }
        return instance;
    }

    // Other methods of the singleton class
    public void showMessage() {
        System.out.println("Hello, I am a Singleton!");
    }

    public static void main(String[] args) {
        // Getting the singleton instance
        Singleton singletonInstance = Singleton.getInstance();

        // Calling a method on the singleton instance
        singletonInstance.showMessage();
    }
}
```

## Relationship

The connection between objects is known as relationship

### Types of relationships

- Has a relationship
- Is a relationship

#### 1. Has a relationship

- The dependency between the object is known as a relationship
- One object is dependent on another object is known as has a relationship

#### Types of has a relationship

- Composition
- Aggregation

#### Composition

- Composition is a strong dependency
- If one object that is main object dependent on another object and if the dependent object does not exist, the main object cannot exist example car-engine, mobile-battery, bank-customers, Instagram-users, etc.

```
class Engine {
    int cc;
    int eno;

    public Engine(int cc, int eno) {
        this.cc = cc;
        this.eno = eno;
    }

    public void start() {
        System.out.println("Engine is started");
    }

    public void stop() {
        System.out.println("Egine is stopped");
    }
}

class Car {
    String brand;
    String color;
    String model;
    Engine e;

    public Car(String brand, String color, String model, Engine e) {
        this.brand = brand;
        this.color = color;
        this.model = model;
    }
}
```

```
        this.e = e;
    }

    public void drive() {
        System.out.println("car is moving");
    }

    public void park() {
        System.out.println("car is parked");
    }
}

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

        Car c1 = new Car("Tata", "black", "nexon", new Engine(1200, 45754));
        Car c2 = new Car("Suzuki", "white", "baleno", new Engine(1100, 454545));

        System.out.println(c1.e.cc);
        System.out.println(c2.color);

        c1.e.start();
        c2.drive();
        c2.e.stop();
        c2.park();

    }
}
```

## Aggregation

- Main object is dependent on another object if the dependent object does not exist then the main object can exist
- This level of dependency is known as aggregation
- It is weak dependency
- Example: mobile-earphone, WhatsApp-status, hotel-swiggy, etc

### Note:

- In aggregation, relationship is established only when it is necessary
- In aggregation, the dependent object is initialized with the help of methods known as helper methods

```
class Earphone {
    String brand;
    String color;

    public Earphone(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }
}

class Mobile {
    String brand;
    int ram;
    int storage;
    Earphone e;

    public Mobile(String brand, int ram, int storage) {
        this.brand = brand;
        this.ram = ram;
        this.storage = storage;
    }

    public void insertEarphone(Earphone e) {
        this.e = e;
    }

    public void removeEarphone() {
        this.e = null;
    }
}

class Aggregation {
    public static void main(String[] args) {
        Mobile m1 = new Mobile("Samsung", 16, 512);
        Mobile m2 = new Mobile("redmi", 8, 256);

        m1.insertEarphone(new Earphone("Boat", "balck"));
    }
}
```



```
        System.out.println(m1.e.brand);
        System.out.println(m2.storage);
        System.out.println(m1.e);
    }
}
```

## 2. Is a Relationship:

- If the relation between objects or classes is like a parent and child relation, such type of relation is known as is a relationship
- By establishing is a relationship between the classes, Inheritance can be achieved

### **Inheritance**

The process of one class acquiring all the properties of another class is known as inheritance

Inheritance can be achieved by using following keywords

- Extends
- Implements

In inheritance only parent class properties are inherited to child classes but child class properties are not inherited to parent class

### **Parent class:**

The class that provides its properties to another class is known as parent superclass

### **Child class**

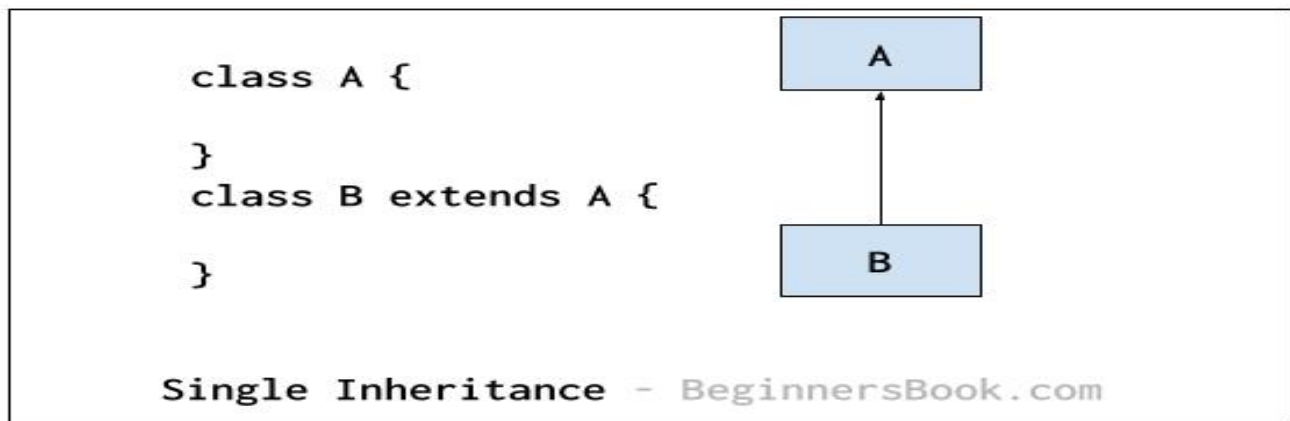
The class that acquires the properties from parent is known as child or derived class

## **Types of Inheritance**

- 1) Single level inheritance
- 2) Multiple level inheritance
- 3) Multi-level inheritance
- 4) Hierarchical inheritance
- 5) Hybrid inheritance

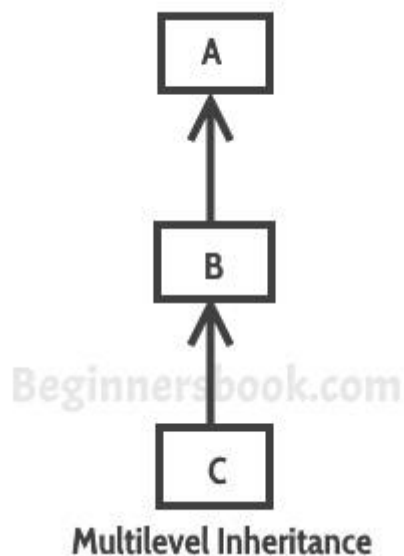
## 1) Single level inheritance

In single Level inheritance, subclass inherits the properties of only one super class is known as single level inheritance



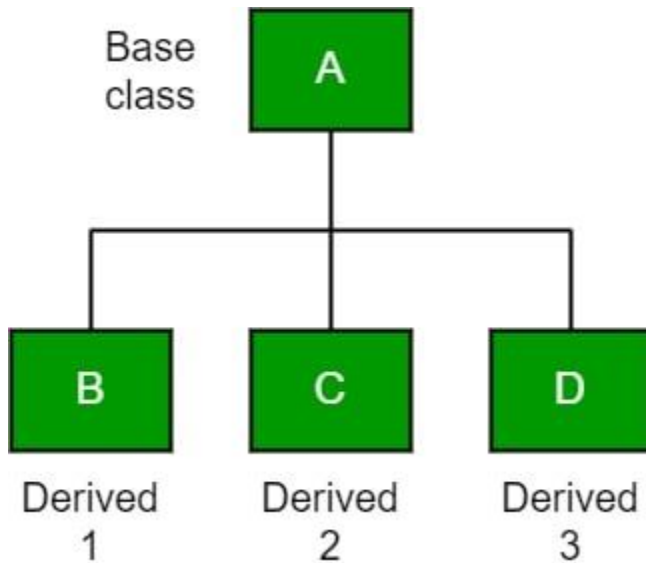
## 2) Multi-level inheritance

In multi-level inheritance the subclass is inheriting from super class and subclass is also superclass of another class



### 3) Hierarchical inheritance

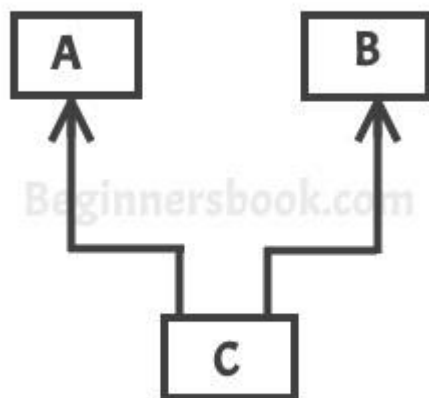
In hierarchical inheritance one superclass is having more than 1 subclass



### 4) Multiple inheritance

If a class directly inheriting properties from more than one parent is known as multiple inheritance

Multiple inheritance is not achievable by using classes because of diamond problem it is achieved only by using interfaces



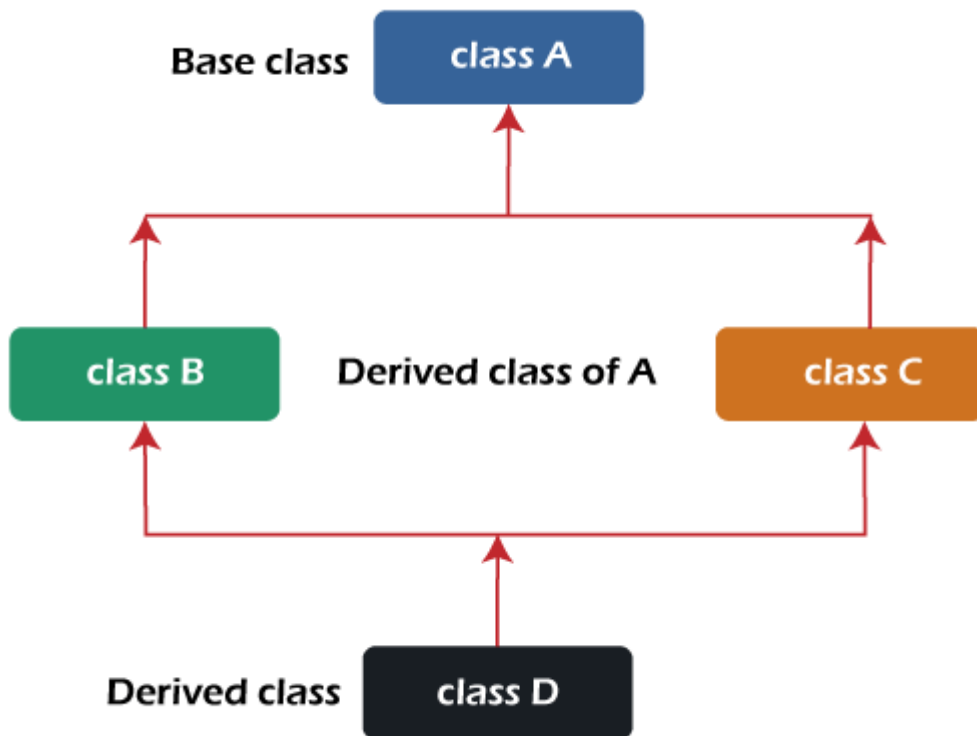
**Multiple Inheritance**

### Diamond problem

Let us consider one scenario like one subclass is inherited multiple super classes

If multiple parents had a member with similar names then the child is confusing to choose the member this state is known as ambiguous state and this problem is called as diamond problem

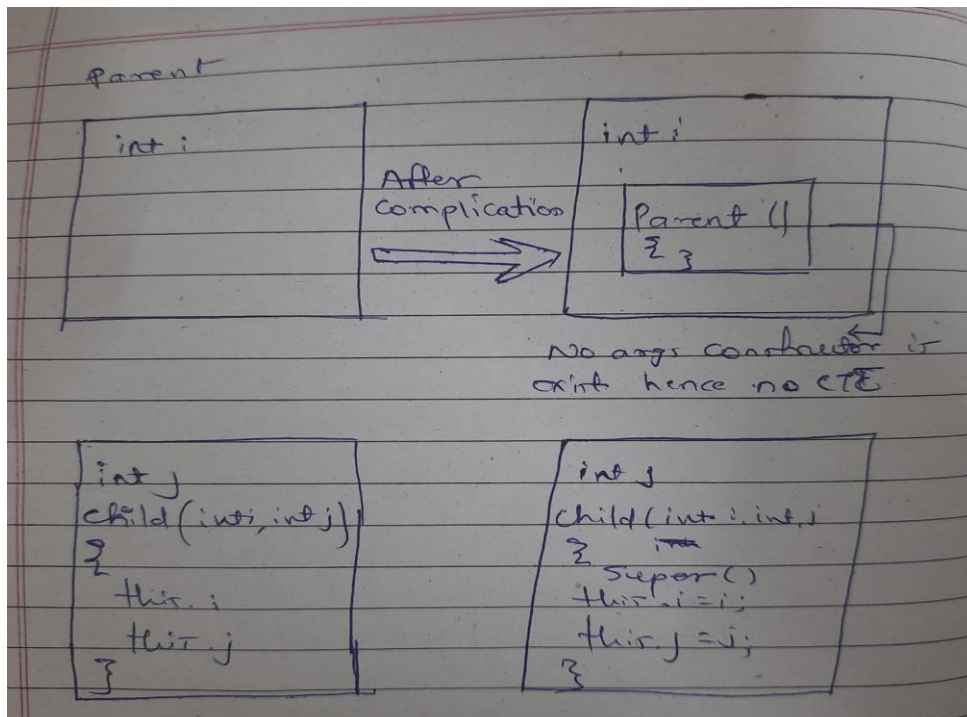
## 5) Hybrid inheritance



It is the combination of more than one inheritance

**Note** In hybrid inheritance combination if there is multiple inherences then it is achieved only through interfaces

### Programming:



**Explanation:**

- Sub class constructor should always call the super class constructor.
- If the programmer doesn't add super call statement in sub class constructor, then by default, Compiler adds the super call statement.
- Compiler always try to call no argument constructor of Super class. Hence, it is recommended to create no argument constructor in super class, or we should call the parameterized constructor of super class.

```
class App {
    String name;
    double size;

    public App(String name, double size) {
        this.name = name;
        this.size = size;
    }
}

class Instagram extends App {
    String userId;
    String pass;

    public Instagram(String name, double size, String userId, String pass) {
        super(name, size);
        this.userId = userId;
        this.pass = pass;
    }
}

class Program {
    public static void main(String[] args) {
        Instagram i = new Instagram("Instagram", 85.25, "user", "pass");
        System.out.println(i.name);
        System.out.println(i.size);
        System.out.println(i.userId);
        System.out.println(i.pass);
    }
}
```

**Non-primitive type Casting**

## Polymorphism

- The word polymorphism is a Greek word in which poly means many and Morphs means forms
- The ability of an object to show multiple behaviours is known as polymorphism

### Types of polymorphism

- Compile Time polymorphism /static binding
- Runtime polymorphism/ dynamic binding

### Compile time polymorphism

- The binding that is achieved during compile time is known as compile time polymorphism
- Compile time Polymorphism can be achieved by using

1. method Overloading
2. Constructor overloading
3. Variable shadowing
4. Method shadowing

### Method overloading

The process of creating more than one method with similar name but different formal arguments in the same class is known as method overloading

```
public class Overloading {  
    public static void main(String[] args) {  
        add(10, 20);  
        add(50, 5, 69);  
        add(20, 63, 41, 5);  
    }  
  
    public static void add(int a, int b) {  
        System.out.println(a + b);  
    }  
  
    public static void add(int a, int b, int c) {  
        System.out.println(a + b + c);  
    }  
  
    public static void add(int a, int b, int c, int d) {  
        System.out.println(a + b + c + d);  
    }  
}
```

## Variable shadowing

The process of creating more than one variable with similar name in inner and outer scope is known as variable shadowing

```
public class Overloading {
    static int a = 20;

    public static void main(String[] args) {
        int a = 40;
        System.out.println(a);
    }
}
```

## Runtime Polymorphism:

The binding achieved during runtime is known as runtime polymorphism.

runtime polymorphism can be achieved by using method overriding.

### Method overriding

The process of creating similar non-static method in super and subclass is known as method overriding.

Or

The process of providing new implementation for the superclass non-static method is known as method overriding.

### Rules to override A method.

- Name and formal arguments of a method should be similar
- Return type of method should be similar
- Only non-static methods can be overridden

**Note** if the child contends overridden method, then if the object is created for child class, then the child class method body only used even though child object is upcasted for the overridden method Child body only used.

```
class Car {
    public void park() {
        System.out.println("car is parked ");
    }

    public void startengine() {
        System.out.println("Runs on fuel");
    }

    public void stopengine() {
        System.out.println("car stoped");
    }
}

public class ElectricCar extends Car {

    @Override
    public void startengine() {
        System.out.println("runs on electricity");
    }

    public static void main(String[] args) {
        Car c1 = new Car();
        c1.startengine();

        ElectricCar c2 = new ElectricCar();
        c2.startengine();

        Car c3 = new ElectricCar();
        c3.startengine(); // child body will execute
    }
}
```



## Abstraction

- The process of hiding unnecessary details and providing only the necessary details is known as obstruction
- The process of providing only the Functionality that is declaration and hiding the implementation is known as abstraction
- Abstraction can be achieved by using abstract keyword and interfaces.

## Abstract

- It is A keyword
- It is also known as non-access modifier
- It is used for classes and non-static methods.

## Abstract method.

- It is a non-static method is prefixed with abstract keywords. Then it is known as abstract method.
- Abstract method should not have a body or implementation or block
- abstract method should be end with;
- The implementation for abstract method should be provided by subclasses.

## Abstract class

- If A class is prefixed with abstract keyword, then it is known as abstract class
- If a class contains even one abstract method, then it is mandatory to make the class as abstract.
- Object Creation is not possible for abstract classes
- If any class inherits the abstract class, then it is mandatory for the subclasses to override the abstract methods of Super Class.

```
abstract class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void walk() {
        System.out.println("Animal is moving");
    }

    abstract public void sound();
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }
}
```

```

    @Override
    public void sound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void sound() {
        System.out.println("meow");
    }
}

public class Abstract {
    public static void main(String[] args) {
        Dog d = new Dog("tom");
        d.sound();

        Animal a = new Dog("rob"); // upcasting
        d.sound();

        Cat c = new Cat("mini");
        c.walk();
    }
}

```

**Note:** if the subclass doesn't want to override the abstract method of Super Class, then it is mandatory to make the subclass as abstract.

## Interfaces

Interface is a component in Java, which is used to achieve 100 percent abstraction and multiple inheritance

### Characteristics of Interfaces.

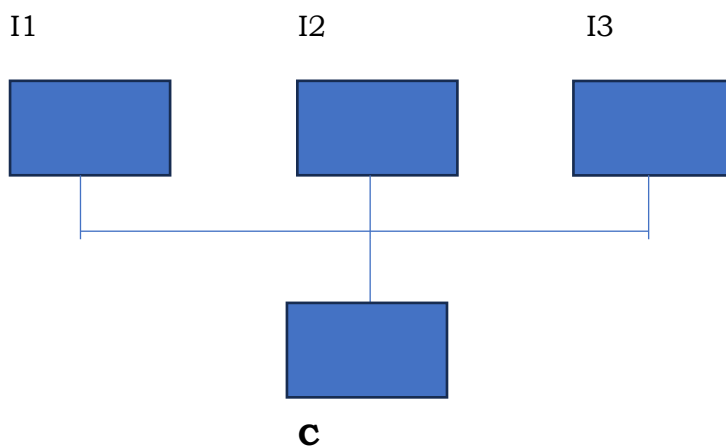
- It is a keyword
- Object creation for interface is not allowed
- Subclass Of an interface is responsible to provide implementation to the abstract methods of an interface.

### Syntax:

```
Interface interfaceName
{
    \\ abstract methods;
}
```

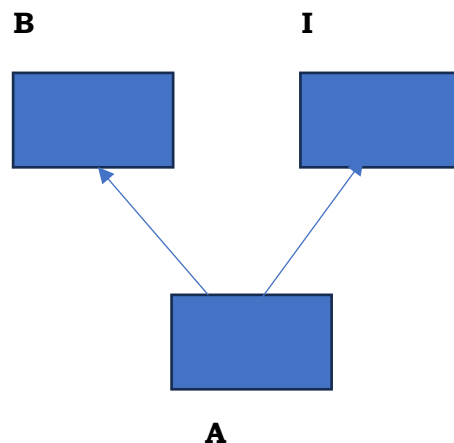
Inheritance among interfaces and the classes.

1)



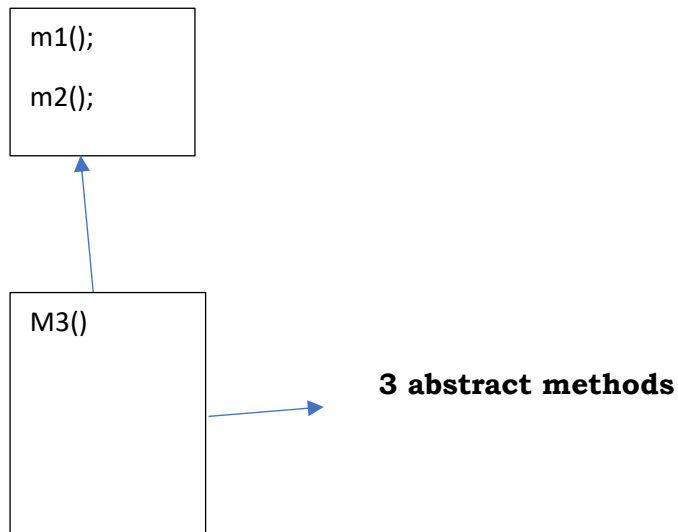
```
class c implements I1,I2,I3
```

2)



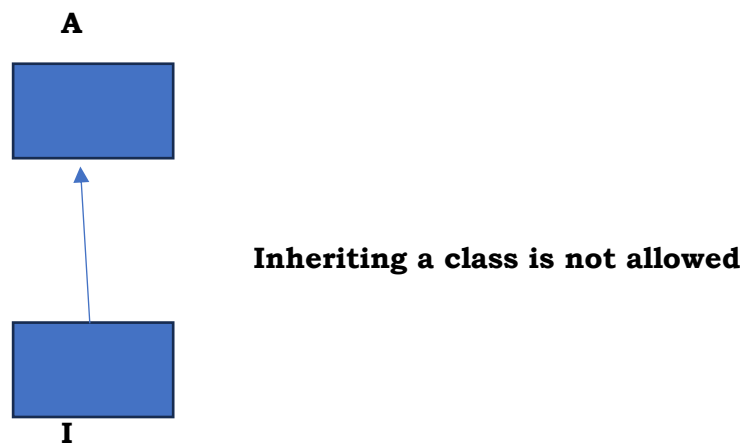
```
class A extends B implements I
```

3) S



```
interface I extends I, I2
```

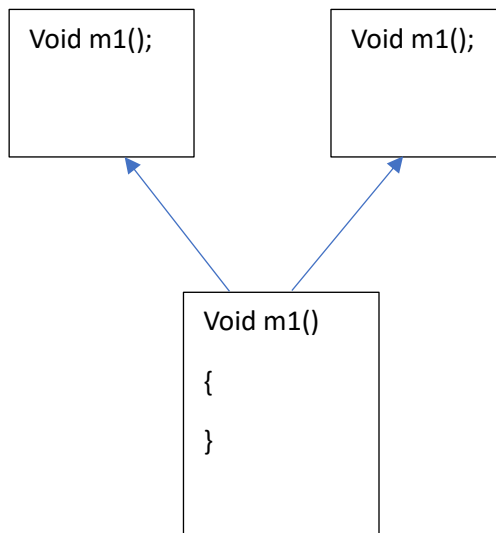
4)



#### Note

Interface can't inherit the class because classmate contain methods which implementation, which is not accepted any interface because. Interface is 100 percent abstraction.

## Multiple Inheritance in Interface



### Example:

```
interface I1 {
    void m1();
}

interface I2 {
    void m1();
}

class A implements I1, I2 {
    @Override
    public void m1() {
        System.out.println("Overridden in A class");
    }
}

class Test {
    public static void main(String[] args) {
        A a = new A();
        a.m1();// A class implementation will execute

        I1 i1 = new A(); // upcasting
        i1.m1();// A class implementation will execute

        I2 i2 = new A(); // upcasting
        i2.m1();// A class implementation will execute
    }
}
```

**Note:**

- In interfaces there is no chance of diamond problem during multiple inheritance because interfaces all the methods are abstract methods.
- The implementation of abstract methods are available in the subclasses.
- Hence there is no confusion and there is no diamond problem.

**Example:**

```
interface Fruit {
    void grow();
}

interface Vegetable {
    void grow();
}

class Chilly implements Fruit, Vegetable {
    @Override
    public void grow() {
        System.out.println("Chilly will grow lengthy");
    }
}

class Test {
    public static void main(String[] args) {
        Chilly c = new Chilly();
        c.grow();
    }
}
```

### Example:

```
interface Message{
    void sendMSG();
}

interface Bank{
    void transferMoney();
    void checkBalance();
}

class JaPay implements Message,Bank{
    @Override
    public void sendMSG(){
        System.out.println("MSG sent");
    }

    @Override
    public void transferMoney(){
        System.out.println("Money Transferred");
    }

    @Override
    public void checkBalance(){
        System.out.println("Balance checked");
    }
}

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

        JaPay jp = new JaPay();
        jp.checkBalance();
        jp.sendMSG();
        jp.transferMoney();

        Message msg = new JaPay();
        msg.sendMSG(); // If the methods belongs to MSG interface then only it will
        upcasted no other methods will execute and will show error

        Bank bank = new JaPay();// If the methods belongs to MSG interface then only
        it will upcasted no other methods will execute and will show error
        bank.checkBalance();
        bank.transferMoney();
    }
}
```

**Members of interface :**

<b>Members</b>	<b>Is Allowed ?</b>
Static Variables	<b>YES ( public static final )</b>
Non-Static variables	<b>NO</b>
Static Methods	<b>YES ( not Inherited )</b>
Non-Static Methods	<b>YES ( public Abstract )</b>
Static Blocks	<b>NO</b>
Non-Static Blocks	<b>NO</b>
Constructors	<b>NO</b>
Default Methods	<b>Yes</b>



## **Java.lang.Object**

It is a predefined class in java.lang package.

It is a super most parent of all the classes in java

In java all the classes by default inherit the java.lang.Object class

Constructor of Object class:

**Object( );**

**Object class contains 11 non-static methods:**

Methods of Object class:

- 1) public String toString()**
- 2) public int hashCode()**
- 3) protected void finalize()**
- 4) public boolean equals(Object O)**
- 5) protected Object clone()**
- 6) public void wait()**
- 7) public void wait(long millisecond)**
- 8) public void wait(long millisecond, int nanosecond)**
- 9) public void notify()**
- 10) public void notifyAll()**
- 11) public Class getClass()**

## 1. public string toString()

- It is defined in java.lang.Object class
- It is used to return the address of an object in **className@HexaDecimal** format.
- If we try to print the address by default toString() method of java.lang.Object classes called.

### Example:

```
Student s1 = new Student("Sheela", 7);  
System.out.println(s1); // s1.toString() // o/p Student@465sdfdf
```

### Purpose of overriding toString() method:

The main purpose of overriding toString() method to return the states of an object instead of returning the address

### Example:

```
class Student{  
    String name;  
    int rollNo;  
  
    public Student(String name, int rollNo){  
        this.name = name;  
        this.rollNo = rollNo;  
    }  
  
    @Override  
    public String toString(){  
        return name + " " + rollNo;  
    }  
  
    public static void main(String[] args) {  
  
        Student s1 = new Student("Sheela", 7);  
        Student s2 = new Student("Laila", 89);  
  
        System.out.println(s1);  
        System.out.println(s2);  
  
    }  
}
```

## 2. public boolean equals ( Object o )

- It is used to compare the address of current object with passed object.
- If both of the object has the same address it returns true else it returns false.

```
Student s1 = new Student("Sheela", 7);
Student s2 = new Student("Sheela", 7);

System.out.println(s1 == s2 ); // false
System.out.println(s1.equals(s2)); // false
```

### Purpose of Overriding equals methods:

The main purpose of overriding equals method is the compare the states of an object instead comparing the address

#### Note:

If the hashCode of two objects are different then equals method should returns false by comparing that two object.

If the hashCode of two objects are same then equals method should returns true by comparing that two object.

Hence, If hashCode method is overridden then it is highly recommended to override equals method. And vice versa

```
public class Student {

    String name;
    int roll;

    public Student(String name , int roll){
        this.name =name;
        this.roll = roll;
    }

    @Override
    public boolean equals(Object o){
        Student s = (Student)o;
        if (name == s.name && roll == s.roll) {
            return true
        }
        else{
            return false;
        }
    }
}
```

```
}
```

```
Student s1 = new Student("Sheela", 7);
Student s2 = new Student("Sheela", 7);

System.out.println(s1 == s2 ); // false
System.out.println(s1.equals(s2)); // true
```

## Example

```
import java.util.Objects;

public class Book {

    String title;
    String author;
    int pages;

    public Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    @Override
    public String toString() {
        return title;
    }

    @Override
    public int hashCode() {
        return Objects.hash(title, author, pages);
    }

    @Override
    public boolean equals(Object o) {
        Book b = (Book) o;
        if (title == b.title && author == b.author && pages == b.pages) {
            return true;
        } else
            return false;
    }
}
```

```

}

public static void main(String[] args) {

    Book b1 = new Book("Java", "Oracle", 500);
    Book b2 = new Book("Java", "Oracle", 500);
    Book b3 = new Book("C++", "ms", 300);

    System.out.println(b1);
    System.out.println(b3);

    System.out.println(b1.hashCode());
    System.out.println(b2.hashCode());

    System.out.println(b1.equals(b2));
}
}

```

```

PS C:\Users\Saurabh\Desktop\java folder> javac Book.java
PS C:\Users\Saurabh\Desktop\java folder> java Book
Java
C++
-1628481993
-1628481993
true
PS C:\Users\Saurabh\Desktop\java folder>

```

## Package:

Package is a folder that contains group of similar classes and interfaces

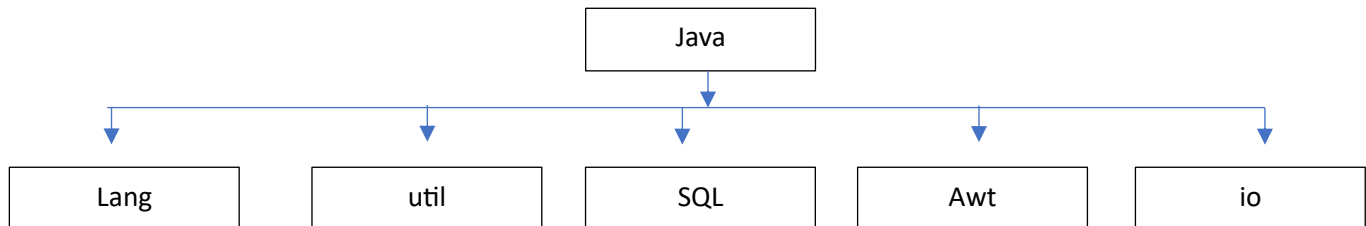
## Types:

- Predefined package
- User defined package

## Predefined package :

The package that is already defined in java is known as predefined package

## Structure:



- The package that is created by the programmer that is known as user defined package
- In java packages can be created by using **package** keyword.

## Syntax:

Package mainPacName.subPack1...subPackn;

- In one source file only one package keyword is allowed
- To place multiple java files in same package we can use same package name for all the java programmes

## Example :

```
package pack1;

public class Replace {
    public static void main(String[] args) {
        System.out.println("hello world !");
    }
}
```

- to use any classes or interfaces from another package **import** keyword can be used
- in one source file any number of import keyword can be used

- in a source file if we use **package** and **import** keyword together then **package** should be the first and **import** should be the next.

### Syntax:

Package pack.subPack1.subPack2...subPackn;

Import pack.subPack.Class/InterfaceName;

Import ....;

Import ....;

.

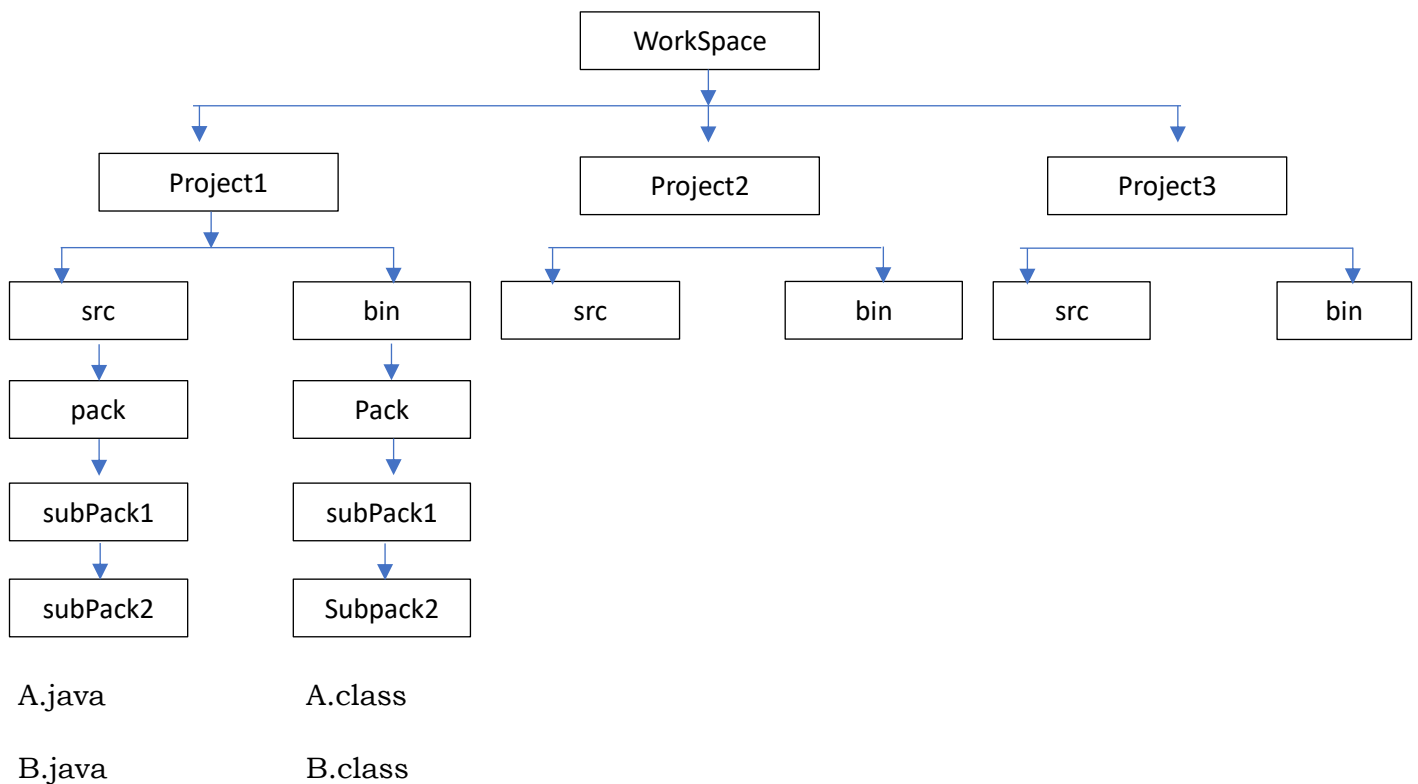
.

Class/Interface ...

{

}

### Java file segregation:



## **Advantage of Package:**

- Easy to search and access
- Easy to maintain
- Secure the file
- Avoids naming conflicts

## **Access Modifiers:**

Access modifiers are used to modify the accessibility

Levels of access modifiers:

**Public > protected > default > private**

### **1) Public :**

It is a keyword

It is used for a class, interface, methods, variable and constructors

Public members can be accessed everywhere

### **2) Protected:**

It is a keyword

It is used only for the methods, variables and constructors.

Protected members can be accessed anywhere within the package but from different package only subclasses can access the protected members

### **3) Default:**

If any members of a class is prefixed with the default keyword then it is known as default members

Default members are access only within the package

### **4) Private:**

It is only used for the methods, variables and constructors.

Private members are accessed within the class

Hence, it is also known as class level modifier



## Scope/visibility of access modifiers

Access Modifiers	Within The Class	Within the Package	Outside the Package	Outside the Package By Subclasses
public	YES	YES	YES	YES
protected	YES	YES	NO	YES
default	YES	YES	NO	NO
private	YES	NO	NO	NO

## Structure of source file

- In one source file any number of classes and interfaces can be created. But there should be at most one public class or interface
- If a source file contains default classes or interfaces then the name of the source file can be anything
- If there is a public class or interface in a source file, then the name of the source file must be public class or interface name
- If one source file contains N number of classes or interfaces then after compilation, we will get N number of class files

```
package pack.subpack;

public class A {

}

class B{

}

class C{

}

class D{

}
```

## Array:

Array is a sequential block of memory that is used to store multiple homogeneous data together.

### Characteristics :

- Array is a homogeneous collection of elements
- The size of an array must be defined at the time of array declaration. Once declared the size of array will be fixed.
- In array elements can be accessed by using its index. Index is an integer value which starts from **0** and ends to **size-1**
- Array accept duplicate elements

**Note:** In java Arrays is an Object.

### Syntax:

**Datatype varName[ ] = new DataType [ length ] ;**

Declaration                      Instantiation

Int array[ ] = new int [5];

0	0	0	0	0
---	---	---	---	---

String array[ ] = new String [5];

Null	Null	Null	Null	null
------	------	------	------	------

### Initializing the values of an array

We can initialize the values of array thorough index.

```
public class A {  
  
    public static void main(String[] args) {  
  
        int arr[] = new int[5];  
        arr[0] = 20;  
        arr[1] = 30;  
        arr[2] = 40;  
        arr[3] = 50;  
        arr[4] = 60;  
  
        System.out.println(arr[0]);  
        System.out.println(arr[1]);  
        System.out.println(arr[2]);  
        System.out.println(arr[3]);  
        System.out.println(arr[4]);  
  
        System.out.println(arr.length);  
    }  
}
```

```
}  
}
```

20	30	40	50	60
----	----	----	----	----

### Accessing the values of array:

To access the values of array we have to use index

```
package pack.subpack;  
  
public class A {  
  
    public static void main(String[] args) {  
  
        String str[] = new String[6];  
  
        str[0] = "Saurabh";  
        str[1] = "Vikas";  
        str[2] = "Vinod";  
        str[3] = "Govind";  
        str[4] = "Mayur";  
        str[5] = "Akash";  
  
        for (int i = 0; i < str.length; i++) {  
            System.out.println(str[i]);  
        }  
  
        System.out.println(str.length);  
  
    }  
}
```

Saurabh	Vikas	Vinod	Govind	Mayur	Akash
---------	-------	-------	--------	-------	-------

## Types of Arrays:

### Primitive Array

The Array which is used to store primitive values is known as primitive array.

### Non-Primitive Array

The array which is used to store non-primitive values is known as non-primitive array.

String s1 [ ] = new String [ 7 ]

```
package samplepack;

public class Book {
    String name;
    int price;
    public Book(String name, int price) {
        super();
        this.name = name;
        this.price = price;
    }
}

class BookDriver{
    public static void main(String[] args) {
        Book b[] = new Book[5];

        b[0] = new Book("c", 350);
        b[1] = new Book("c++", 450);
        b[2] = new Book("Python", 250);
        b[3] = new Book("sql", 300);
        b[4] = new Book("Java", 500);

        for (int i = 0; i < b.length; i++) {
            System.out.println("Book name: " + b[i].name + " \nBook Price: " + b[i].price + "\n-----");
        }
    }
}
```

```
Book name: c
Book Price: 350
-----
Book name: c++
Book Price: 450
-----
Book name: Python
Book Price: 250
-----
Book name: sql
Book Price: 300
-----
Book name: Java
Book Price: 500
-----
```

searching the book based on name

code:

```
package samplepack;

import java.util.Scanner;

public class Book {
    String name;
    int price;

    public Book(String name, int price) {
        super();
        this.name = name;
        this.price = price;
    }
}

class BookDriver {
    public static void main(String[] args) {
        Book b[] = new Book[5];

        b[0] = new Book("c", 350);
        b[1] = new Book("c++", 450);
        b[2] = new Book("Python", 250);
        b[3] = new Book("sql", 300);
        b[4] = new Book("Java", 500);

        for (int i = 0; i < b.length; i++) {
            System.out.println("Book name: " + b[i].name + "\nBook Price: " + b[i].price + "\n-----");
        }

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the name of book for searching: ");
        String Uname = sc.next();

        for (int i = 0; i < b.length; i++) {
            if (Uname.equals(b[i].name)) {
                System.out.println(Uname + " Present");
                return;
            }
        }

        System.out.println(Uname + " Not fount");
    }
}
```

```
Book name: c
Book Price: 350
-----
Book name: c++
Book Price: 450
-----
Book name: Python
Book Price: 250
-----
Book name: sql
Book Price: 300
```

```
-----  
Book name: Java  
Book Price: 500  
-----  
Enter the name of book for searching:  
c  
c Present
```

**Write a program for read and print the elements of an array :**

```
package pack.subpack;  
  
import java.util.Scanner;  
  
public class A {  
  
    public static void main(String[] args) {  
  
        Scanner sc = new Scanner(System.in);  
  
        System.out.print("Enter the lenght : ");  
        int size = sc.nextInt();  
        int arr[] = new int[size];  
  
        for (int i = 0; i < size; i++) {  
            arr[i] = sc.nextInt();  
        }  
  
        for (int i = 0; i < arr.length; i++) {  
            System.out.print(arr[i] + " ");  
        }  
  
        System.out.println(arr.length);  
  
    }  
}
```

```
Enter the lenght : 5  
10 2 3 04 5  
10 2 3 4 5 5
```

**Write a java program to search the elements of an array:**

**Code:**

```
package pack.subpack;

import java.util.Scanner;

public class A {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the lenght : ");
        int size = sc.nextInt();
        String str[] = new String[size];

        for (int i = 0; i < str.length; i++) {
            str[i] = sc.next();
        }

        boolean exist = false;
        System.out.println("Enter string for searching :");
        String k = sc.next();

        for (String string : str) {
            if (string.equals(k)) {
                exist = true;
                break;
            }
        }
        if (exist) {
            System.out.println("found");
        } else {
            System.out.println("not found");
        }
    }
}
```

<terminated> A [Java Application] C:\Program Files\Java\jdk-20\bin

```
Enter the lenght : 3
Saurabh Omkar Vikas
Enter string for searching :
Omkar
found
```

**Write a java program to search an element from the array and prints its index.**

**Ans:**

```
package pack.subpack;

import java.util.Scanner;

public class A {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the lenght : ");
        int size = sc.nextInt();
        String str[] = new String[size];
        System.out.println("Enter the String : ");
        for (int i = 0; i < str.length; i++) {
            str[i] = sc.next();
        }

        boolean exist = false;
        System.out.println("Enter string for searching :");
        String k = sc.next();

        for (int i = 0; i < size; i++) {
            if (str[i].equals(k)) {
                exist = true;
                if (exist) {
                    System.out.println("found");
                } else {
                    System.out.println("not found");
                }
            }
            System.out.println("The Index is :" + i);
        }
    }
}
```

```
<terminated> A [Java Application] C:\Program Files\Java\jdk-
Enter the lenght : 3
Enter the String :
Saurabh
Omkar
Mayur
Enter string for searching :
Omkar
found
The Index is :1
```



```
// package pack.subpack;

import java.util.Scanner;

public class Array {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter rows and columns");
        int row = sc.nextInt();
        int column = sc.nextInt();

        int[][] arr = new int[row][column];

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

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                arr[i][j] = sc.nextInt();
            }
        }
        System.out.println("The Array:");

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Array
Enter rows and columns
2 2
Enter datas
12 12
45 45
The Array:
12 12
45 45
PS C:\Users\Saurabh\Desktop\java folder> 
```

## Jagged Array :

- Jagged array is also known as irregular array and ragged Array.
- In jagged array every array has irregular or different length

## Syntax:

```
datatype [ ] [ ] [ ] ... [ n ] var = new datatype [ size ] [ size ] [ ];  
var [ index ] [ index ] = new datatype [ type ];
```

## Example:

```
package pack.subpack;  
  
import java.util.Scanner;  
public class JaggedArray {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner (System.in);  
  
        int [][] arr = new int [4][];  
        arr[0] = new int [2];  
        arr[1] = new int [3];  
        arr[2] = new int [7];  
        arr[3] = new int [4];  
  
        System.out.println("Enter the datas");  
  
        for(int i = 0 ; i<arr.length;i++ ) {  
            for(int j = 0 ; j<arr[i].length;j++ ) {  
  
                arr[i][j] = sc.nextInt();  
            }  
        }  
  
        System.out.println(" ----- ");  
  
        System.out.println("The array:");  
  
        for(int i = 0 ; i<arr.length;i++ ) {  
            for(int j = 0 ; j<arr[i].length;j++ ) {  
                System.out.print(arr[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

o/p:

```
Enter the datas
```

```
12 89
```

```
85 96 75
```

```
4 5 6 7 8 9 3
```

```
1 2 3 4
```

```
| -----
```

```
The array:
```

```
12 89
```

```
85 96 75
```

```
4 5 6 7 8 9 3
```

```
1 2 3 4
```

**Declaration and initialization of Array in single line:**

**Syntax:**

```
datatype [ ] [ ] [ ] ... [ n ] var = new { { d1,d2,d3,...,dn} , { d1,d2,d3,...,dn} , { d1,d2,d3,...,dn} }
```

**Program:**

```
package pack.subpack;

import java.util.Scanner;

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

        int[][] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
        System.out.println("The Array:");

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr[i].length; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
The Array:
```

```
1 2 3
```

```
4 5 6
```

```
7 8 9
```

## String

### Strings Literals / String constants:

- String is a collection of characters
- It is represented by double quotes – “ ”
- In java to use String literals, instance i.e. object should be created for any of the following classes.

**Java.lang.String**

**Java.lang.StringBuffer**

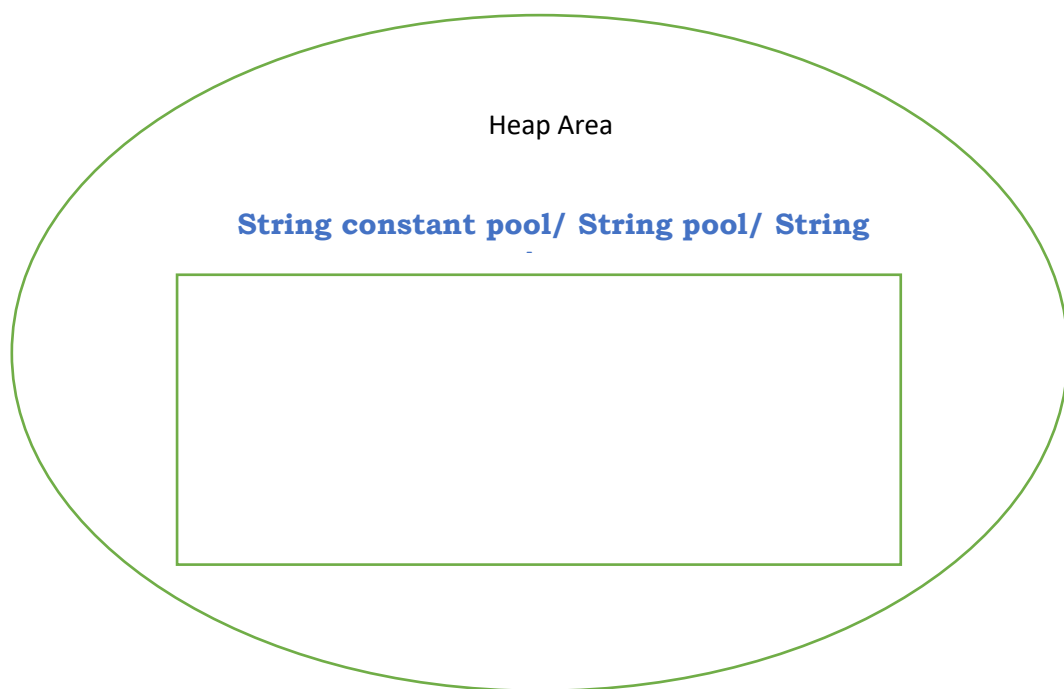
**Java.lang.StringBuilder**

In java if the String literal is used directly then by default object is created for **Java.lang.String** class in **String constant pool**.

### String constant pool:

**String constant pool** is a memory in heap area

In **String constant pool** for every unique String literal one instance are object in created for string class



```
String name1 = "Bindu";
```

```
String name2 = "Sheela";
```

```
String name1 = "Bindu";
```

```
String name1 = "SHEELA";
```

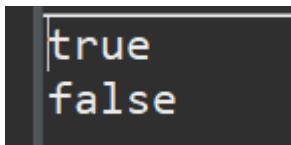
```

package pack.subpack;

public class String1 {
    public static void main(String[] args) {
        String s1 = "Hi";
        String s2 = "Hello";
        String s3 = "hi";
        String s4 = "Hi";

        System.out.println(s1.equals(s3)); // compare states
        System.out.println(s2 == s1); // compare address
    }
}

```



## Java.lang.String

- It is an inbuilt class defined in **java.lang** package
- It is a **final** class
- It is used to represent the String literals.
- In String class **toString()**, **equals()** and **hashCode()** methods of java.lang.Object class are overridden
- It implements **serializable**, **comparable**, **charsequence** interfaces

## Constructors of String class:

- **String( )**

It is used to create a new String object with empty characters.

- **String( byte [ ] bytes)**

It is used to create new string object by decoding the specified array of bytes into a character.

- **String( char [ ] values)**

It is used to create a new String object that contains state that are specified in the characters array.

- **String( String original )**

It is used to create new String object that is initialized with the specified String literals

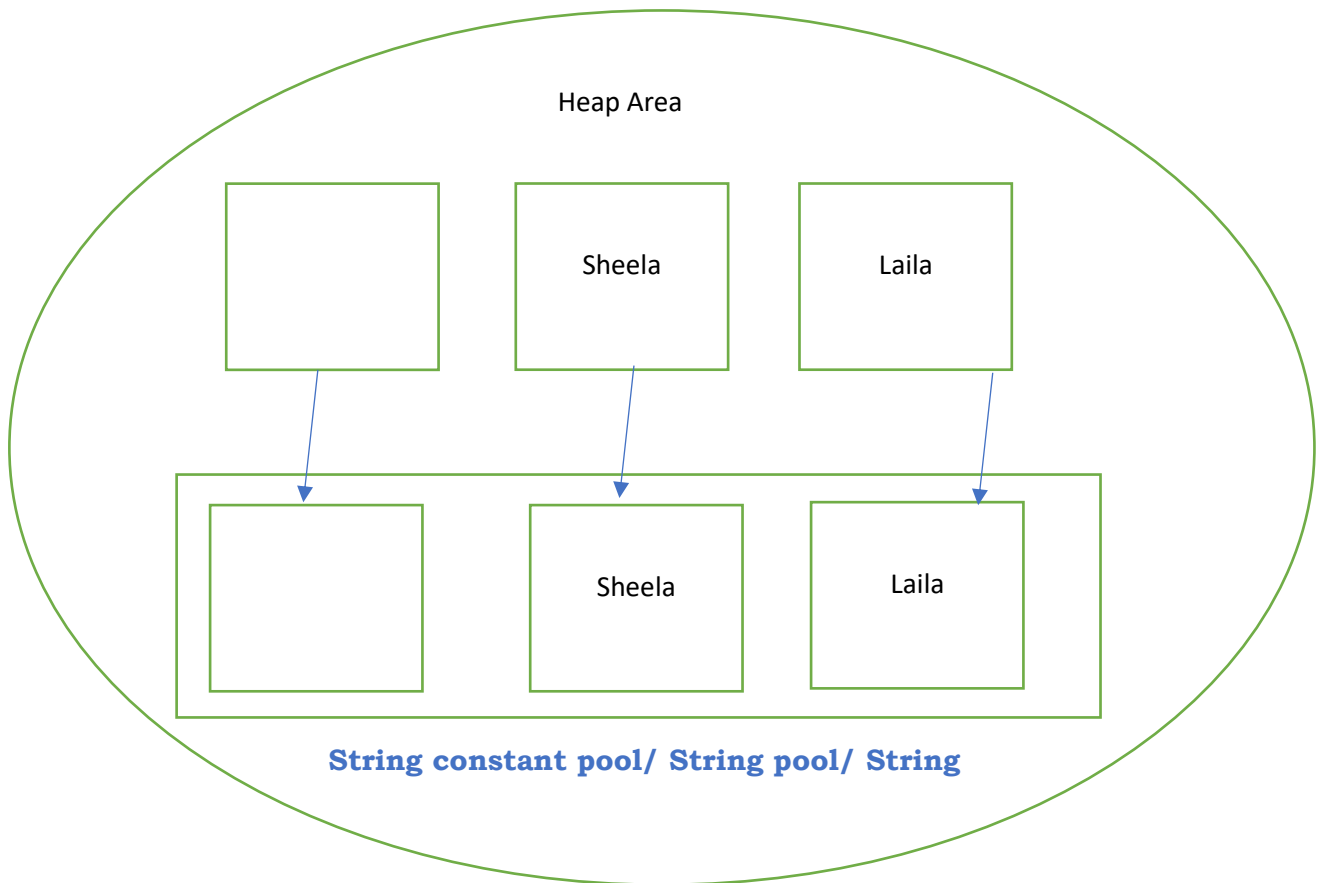
- **String ( String Buffer )**

It creates a new string object that contains a sequence of characters that are currently available as arguments in the specified String Buffer

- **String ( String Builder )**

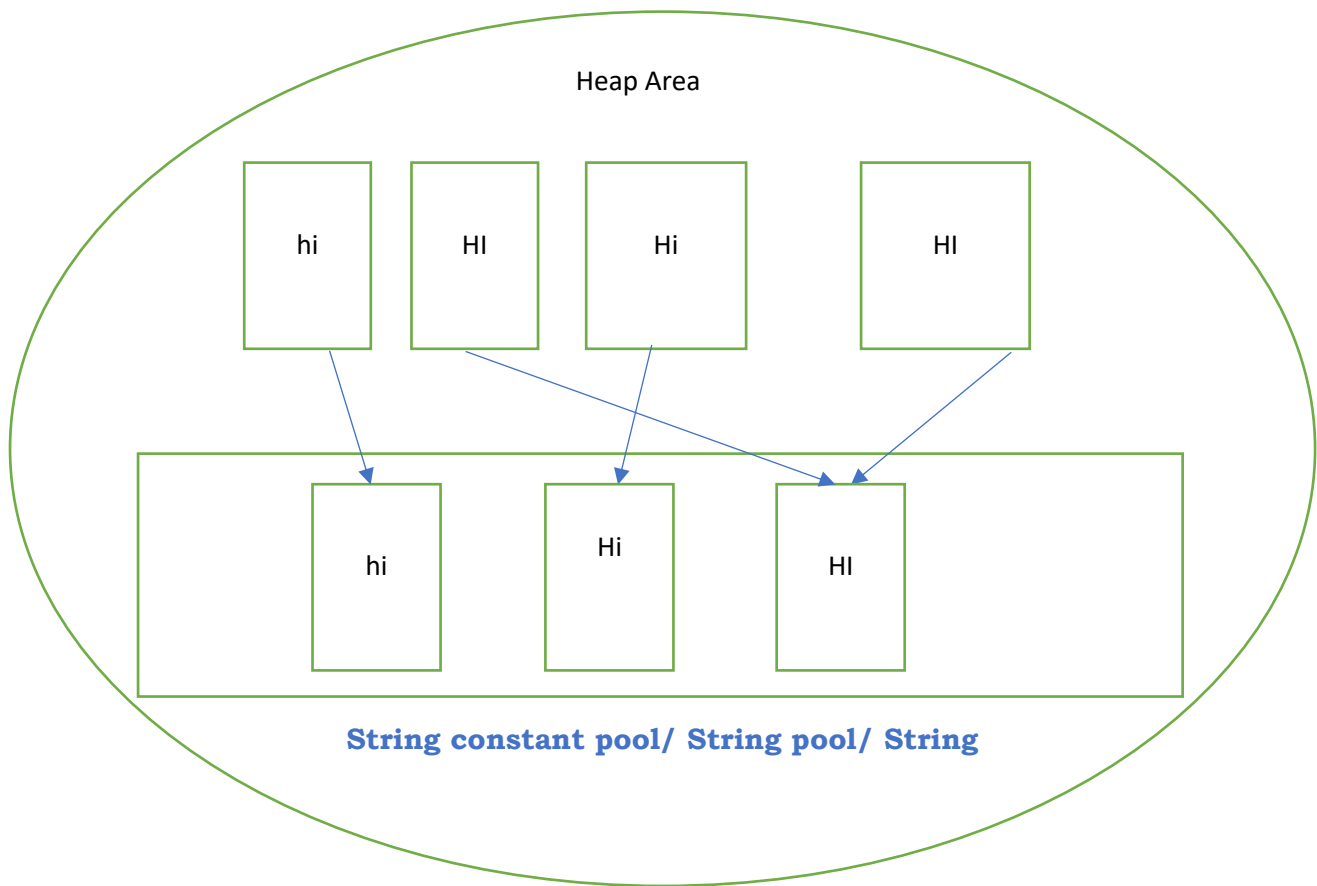
It creates a new string object that contains a sequence of characters that are currently available as arguments in the specified String Builder

```
String s1 = new String();  
String s2 = new String("Sheela");  
String s3 = new String("Laila");
```



```
package pack.subpack;  
  
public class String1 {  
    public static void main(String[] args) {  
        String s1 = new String("hi");  
        String s2 = new String("Hi");  
        String s3 = "HI";  
        String s4 = "hi";  
        String s5 = new String("Hi");  
        String s6 = new String("HI");  
  
        System.out.println(s1 == s5);  
        System.out.println(s3.equals(s5));  
        System.out.println(s6.equals(s3));  
        System.out.println(s6 == s3);  
        System.out.println(s1 == s4);  
    }  
}
```

```
false  
false  
true  
false  
false
```



```

package pack.subpack;

public class String1 {
    public static void main(String[] args) {

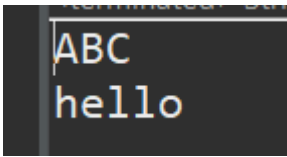
        byte[] arr = { 65, 66, 67 };
        char[] arr1 = { 'h', 'e', 'l', 'l', 'o' };

        String s1 = new String(arr);
        String s2 = new String(arr1);

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

    }
}

```



```

ABC
hello

```

## Methods of string class

### ➤ Public char charAt( int index ):

It is used to return the character present at specified index

```

package pack.subpack;

public class String1 {
    public static void main(String[] args) {
        String str = "Hello Everyone";

        System.out.println(str.charAt(7));
        System.out.println(str.charAt(10));
        System.out.println(str.charAt(1));

    }
}

```

```

v
y
e

```



➤ **Public String toUpperCase():**

It is used to covert the string literals to uppercase.

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello Everyone";  
  
        System.out.println(str.toUpperCase());  
    }  
}
```

```
HELLO EVERYONE
```

➤ **Public String toLowerCase():**

It is used to covert the string literals to lowercase

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello Everyone";  
        System.out.println(str.toLowerCase());  
    }  
}
```

```
hello everyone
```

➤ **Public int length()**

It is used to return the number of characters

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello Everyone";  
        System.out.println(str.length());  
    }  
}
```

➤ **Public boolean equals(String s)**

- It is used to check whether two String literals are same or different if it is same it return true else it returns false
- It is case sensitive

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "sindhu";  
        String str1 = "SINDHU";  
  
        System.out.println(str.equals(str1));  
    }  
}
```

false

➤ **Public boolean equalsIgnoreCase(String s)**

- It compares two strings without considering the case
- If both of the string literals are same it returns true else it returns false

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "sindhu";  
        String str1 = "SINDHU";  
  
        System.out.println(str.equalsIgnoreCase(str1));  
    }  
}
```

true

➤ **Public int indexOf(Char c):**

- It returns the index of the character that has the first occurrence.
- If the character is not present then it return -1;

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "sindhu";  
        String str1 = "SINDHU";  
  
        System.out.println(str.indexOf("s"));  
        System.out.println(str.indexOf("n"));  
        System.out.println(str.indexOf("u"));  
    }  
}
```

0  
2  
5

➤ **Public int index(char c, int startIndex)**

It starts searching the character from the specified index and it returns the index of the character that has the first occurrence.

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "sindhu";  
        String str1 = "SINDHU";  
  
        System.out.println(str.indexOf("n", 1));  
    }  
}
```

2

➤ **Public int lastIndex(char ch):**

It starts searching the character from the last index and it returns the index of the character that has the last occurrence.

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "sheela";  
        String str1 = "SINDHU";  
  
        System.out.println(str.lastIndexOf('e'));  
    }  
}
```

3

➤ **Public String subString( int index ):**

It is used to extract the portion of substring from the specified index.

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello, world!";  
        String sub = str.substring(7);  
  
        System.out.println(sub);  
    }  
}
```

world!

➤ **Public String subString( int startIndex, int endIndex ):**

It is used to extract the portion of substring from the specified range

```
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello, world! how are you";  
        String sub = str.substring(7, 14);  
  
        System.out.println(sub);  
    }  
}
```

world!

➤ **Public String [ ] split ( String s ):**

It is used to split the given string when it finds the matching substring that is specified

```
package pack.subpack;  
  
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello,world,how,are,you";  
        String[] substrings = str.split(",");  
  
        // Iterate over the array and print each element  
        for (String substring : substrings) {  
            System.out.println(substring);  
        }  
    }  
}
```

```
Hello  
world  
how  
are  
you
```

➤ **Public String replace( char oldChar, char newChar ):**

It is used to replace the specified character with the new character

e.g.

```
public static void main(String[] args) {  
  
    String str = "sindhu";  
    String str1 = "SINDHU";  
  
    System.out.println(str.replace('s', 'B'));  
  
}
```

```
Bindhu
```

➤ **public char[] toCharArray():**

it is used to convert the string to character type array

```
package pack.subpack;  
  
public class String1 {  
    public static void main(String[] args) {  
        String str = "Hello";  
        char[] charArray = str.toCharArray();  
  
        // Iterate over the array and print each character  
        for (char c : charArray) {  
            System.out.println(c + " ");  
        }  
        System.out.println(); }  
}
```

```
H  
e  
l  
l  
o
```

➤ **public String trim():**

it is used to remove the unwanted spaces that is present before and after the space

```
package pack.subpack;

public class String1 {
    public static void main(String[] args) {

        String str = " Hello, world! ";
        String trimmed = str.trim(); // trimmed will be "Hello, world!"

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

```
Hello, world!
```

➤ **public static String valueOf( ... ):**

- it is used to convert any specified type of data to string
- it is overloaded method that accepts all type of data's

```
package pack.subpack;

public class String1 {
    public static void main(String[] args) {
        int num = 123;
        String strNum = String.valueOf(num);
        System.out.println(strNum); // Output: 123

        double dbl = 3.14;
        String strDbl = String.valueOf(dbl);
        System.out.println(strDbl); // Output: 3.14

        char ch = 'A';
        String strCh = String.valueOf(ch);
        System.out.println(strCh); // Output: A

        boolean bool = true;
        String strBool = String.valueOf(bool);
        System.out.println(strBool); // Output: true
    }
}
```

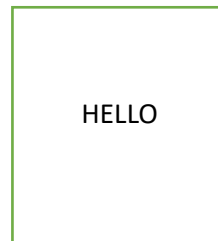
```
123
3.14
A
true
```

## characteristics of String :

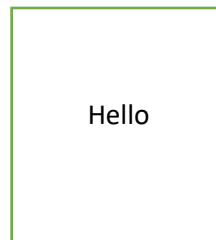
- String objects are immutable or constant in nature i.e. String objects are not modified

**Note :** if we try to modify the String object then the existing object won't be modified instead of that new object is created in heap area and address is returned by the method.

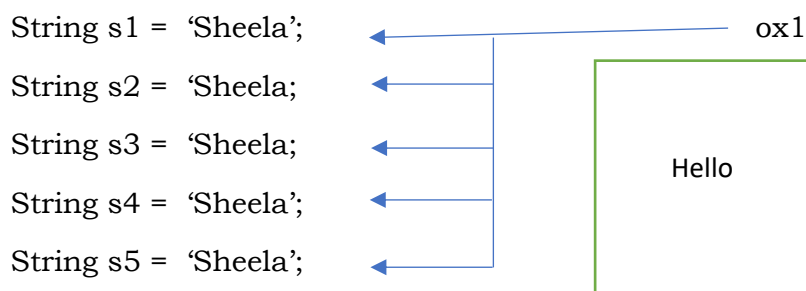
```
String s = 'Hello';  
S = s.toUpperCase();
```



Ox1



- One String object may refer by multiple users
- If we do the modification for one user within the object it will affect the all the users who are referring the same object



To create a mutable object for String literals `java.lang.StringBuffer` and `java.lang.StringBuilder` are used

### **java.lang.StringBuffer**

- it is a final class which is defined in `java.lang` package
- it is used to represent a mutable object for string literals.
- String Buffer objects are thread safe
- In String Buffer class `toString` methods of object class is overridden.
- String Buffer implements `Appendable`, `serializable`, `comparable`, `charsequence` interface

#### **Constructors:**

➤ **StringBuffer():**

It creates an empty string Buffer object with default capacity of 16.

➤ **StringBuffer( String s ):**

It creates a StringBuffer object that is initialized with the specified string literals.

Initial capacity = 16 + number of characters in string literals.

➤ **StringBuffer(int initialCapacity) :**

It creates an empty StringBuffer object with the specified initial capacity

➤ **Note:**

For StringBuffer class objects are created in heap area

```
package pack.subpack;

public class String1 {
    public static void main(String[] args) {

        StringBuffer s1 = new StringBuffer();
        StringBuffer s2 = new StringBuffer("hi");

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

    }
}
```



## Methods of StringBuffer:

### ➤ **Public StringBuffer append( .... );**

- It is the overloaded method.
- It is used to perform concatenation of any type of data with the StringBuffer.

```
StringBuffer sb1 = new StringBuffer("Sheela");
StringBuffer sb2 = new StringBuffer("Laila");
sb1.append(sb2);
System.out.println(sb1);
```

SheelaLaila

### ➤ **Public int capacity()**

It returns the current capacity of StringBugffer.

```
public class String1 {
    public static void main(String[] args) {
        StringBuffer sb1 = new StringBuffer("Sheela");
        StringBuffer sb2 = new StringBuffer("Laila");

        System.out.println(sb1.capacity());
    }
}
```

22

### ➤ **Public char charAt(int index)**

It is used to return the character present at specified index

```
public class String1 {
    public static void main(String[] args) {
        StringBuffer sb1 = new StringBuffer("Sheela");
        StringBuffer sb2 = new StringBuffer("Laila");

        System.out.println(sb1.charAt(4));
    }
}
```

l

➤ **Public StringBuffer delete( int startIndex, int endIndex ):**

It is used to delete the characters from the specified index range

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the worst Wednesday");  
        sb1.delete(13, 19);  
        System.out.println(sb1);  
    }  
}
```

Today is the Wednesday

➤ **Public StringBuffer deleteCharAt(int index) :**

It used to delete the character at the specified index.

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the Wednesday");  
        sb1.deleteCharAt(3);  
        System.out.println(sb1);  
    }  
}
```

Tody is the Wednesday

➤ **Public StringBuffer reverse():**

It us used to reverse the StringBuffer

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.reverse());  
    }  
}
```

yadsendew eht si yadoT

➤ **Public int length():**

It is used to return the number of characters

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.length());  
    }  
}
```

22

➤ **Public String substring (int index ):**

It is used to extract the portion of substring from the specified index.

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.substring(6));  
    }  
}
```

is the wednesday

➤ **Public String substring (int startIndex, int endIndex ):**

It is used to extract the portion of substring from the specified range

```
package pack.subpack;  
  
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.substring(6,12));  
    }  
}
```

is the

➤ **Public int indexOf(String):**

It starts searching the character from the specified index and it returns the index of the character that has the first occurrence.

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.indexOf("y"));  
    }  
}
```

4

➤ **Public int indexOf(String , int startIndex):**

It starts searching the character from the last index and it returns the index of the character that has the last occurrence.

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.indexOf("y", 6));  
    }  
}
```

21

➤ **Public int lastIndexOf(String)**

It starts searching the character from the last index and it returns the index of the character that has the last occurrence.

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.lastIndexOf("a"));  
    }  
}
```

20

➤ **public StringBuffer insert ( int index, .... )**

it is used to insert any type of data at specified index in a StringBuffer.

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Lila");  
        sb1.insert(1, 'a');  
        System.out.println(sb1);  
    }  
}
```

Laila

➤ **Public StringBuffer replace(int start index, int endIndex, String str)**

It is used to replace StringBuffer in a specified index range with the new String

```
public class String1 {  
    public static void main(String[] args) {  
        StringBuffer sb1 = new StringBuffer("Today is the wednesday");  
  
        System.out.println(sb1.replace(9, 12, "worst"));  
    }  
}
```

Today is worst wednesday

➤ **Public void setCharAt( int index, char ch)**

It is used to replace character at specified index

```
StringBuffer sb1 = new StringBuffer("theela");  
sb1.setCharAt(0, 's');  
System.out.println(sb1);
```

sheela

➤ **public void trimToSize()**

it is used to remove the unused capacity of StringBuffer.

```
StringBuffer sb1 = new StringBuffer("theela");  
System.out.println(sb1.capacity());  
sb1.trimToSize();  
System.out.println(sb1.capacity());
```

**Difference between StringBuffer and StringBuilder**

Feature	StringBuffer	StringBuilder
Mutability	Mutable	Mutable
Thread-Safety	Thread-safe (synchronized)	Not thread-safe (not synchronized)
Performance	Slower due to synchronization	Faster, as it's not synchronized
Use Cases	Suitable for multi-threaded environments	Ideal for single-threaded environments
Introduced in	JDK 1.0	JDK 1.5
Efficiency	Less efficient due to synchronization overhead	More efficient due to lack of synchronization overhead

## Exception

- Exception is an unexpected problem that occurs during runtime.
- Hence, it is also known as runtime error.
- It occurs only when there is an abnormal situation in a program.
- Exception occurs only because of some instructions when it is under the abnormal situation.

### Example:

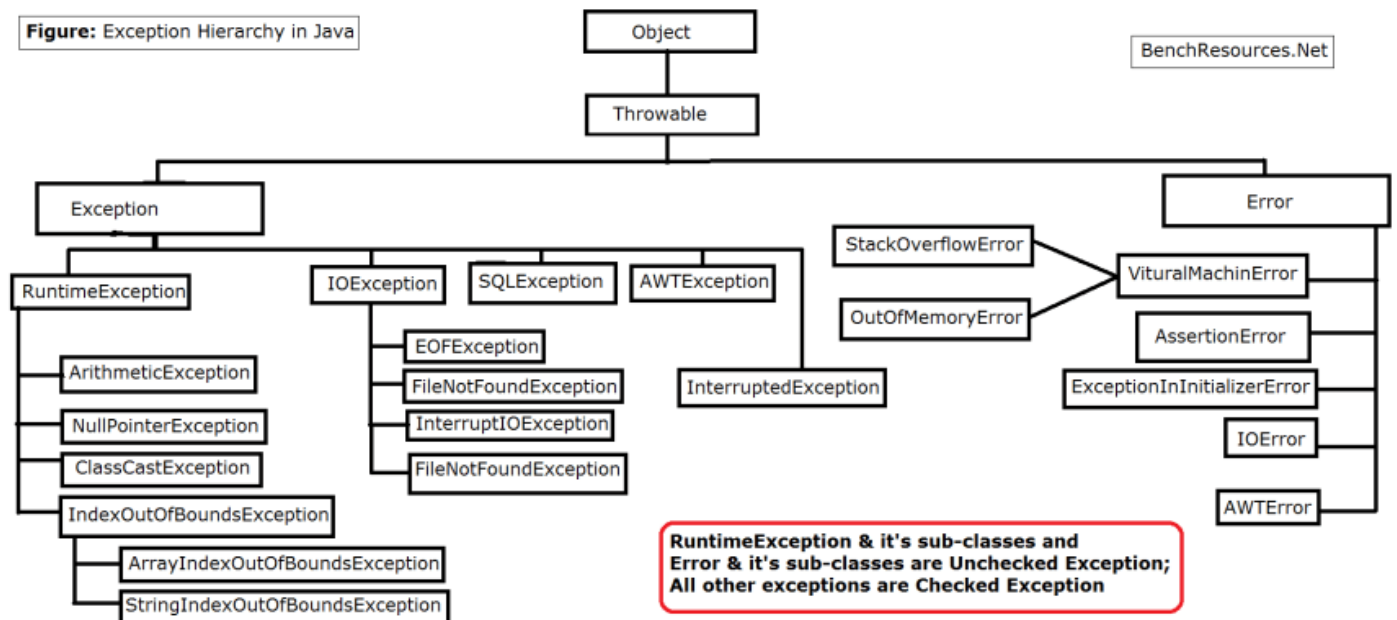
```
class Exception {  
    public static void main(String[] args) {  
  
        System.out.println("Start");  
        int[] arr = { 1, 2, 3, 4, 5, 6 };  
        System.out.println(arr[10]);  
        System.out.println("End...");  
    }  
}
```

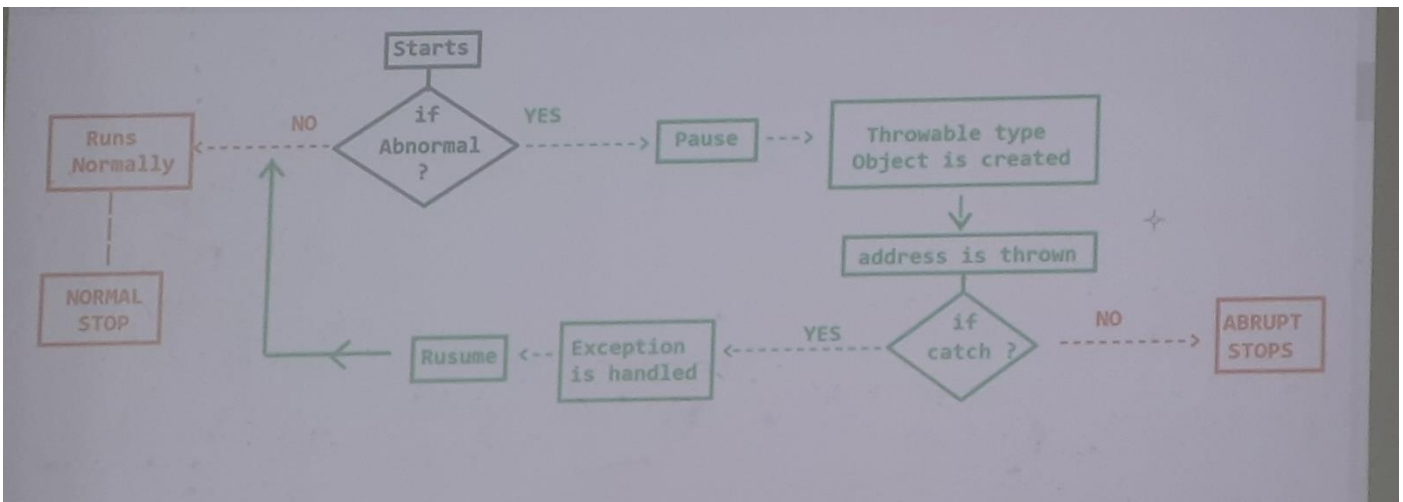
Start

Exception in thread "main" [java.lang.ArrayIndexOutOfBoundsException](#): Index 10 out of bounds for length 6  
at pack.subpack.Exception.main([Exception.java:7](#))

Every exception in java is the inbuilt class of throwable.

Figure: Exception Hierarchy in Java





## Exception Handling:

Exception handling is the mechanism of executing the program normally even in case of exception.

Exception can be handled by using try and catch block

## Syntax:

```

try{
    // risky code
    // instructions depend on risky code
}
Catch( Exception e ){
    // instructions
}

```

## Working process:

**Case 1: Exception not occurs.**

```

try{
    // risky code
    // instructions depend on risky code
}
Catch( Exception e ){
    // instructions
}

```



## Case 2: Exception occurs

```
try{  
    // risky code  
    // instructions depend on risky code  
}  
Catch( Exception e ){  
    // instructions  
}
```

```
class Exception {  
    public static void main(String[] args) {  
  
        System.out.println("Start");  
        int[] arr = { 1, 2, 3, 4, 5, 6 };  
        try {  
            System.out.println(arr[10]);  
        }  
        catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println(e);  
        }  
        System.out.println("End...");  
    }  
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Exception  
Start  
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 6  
End...  
PS C:\Users\Saurabh\Desktop\java folder> █
```

## Null pointer exception

```
class Exception {  
    public static void main(String[] args) {  
        try {  
  
            String str = null;  
            int length = str.length();  
        } catch (NullPointerException e) {  
  
            System.out.println("Caught NullPointerException: " + e.getMessage());  
        }  
    }  
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Exception
Caught NullPointerException: Cannot invoke "String.length()" because "<local1>" is null
PS C:\Users\Saurabh\Desktop\java folder> javac Exception.java
```

String index out of bound exception

```
class Exception {
    public static void main(String[] args) {
        try {
            int [] arr = {1,2,3};
            System.out.println(arr[3]);
        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println( e.getMessage());
        }
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Excep
Index 3 out of bounds for length 3
PS C:\Users\Saurabh\Desktop\java folder> █
```

### Try with multiple catch block

```
try {

} catch (Exception e) {
    // TODO: handle exception
} catch (Exception e) {
    // TODO: handle exception
} catch (Exception e) {
    // TODO: handle exception
} catch (Exception e) {
    // TODO: handle exception
}
```

- While using multiple catch block the throwable catch block from top to bottom order.
- If any of the catch block caught the address then it is not thrown to the below catch block.

**Note.:** while using try with multiple catch block if any catch block is created for parent exception then it should be the last catch block.

### Example:

```
class Exception1 {
    public static void main(String[] args) {
        String s = "Saurabh";
        int[] arr = { 1, 2, 3 };
        try {
            System.out.println(s.charAt(1));
            int a = 10 / 0;
            System.out.println(a);
            System.out.println(arr[829]);
        } catch (NullPointerException e) {
            System.out.println(e);
        } catch (StringIndexOutOfBoundsException e) {
            System.out.println(e);
        } catch (ArithmeticException e) {
            System.out.println(e);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

### Exception Object Propagation:

The movement of exception object from called method to caller when it is not handled is known as exception object Propagation

```
class Exception1 {
    public static void main(String[] args) {

        System.out.println("Main Start");
        try {
            div(10,0);
        } catch (ArithmeticException e) {
            System.out.println("Handled");
        }
        System.out.println("Main End");
    }
    public static void div(int a, int b){
        int s = a/b;
        System.out.println(s);
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Exception1
Main Start
Handled
Main End
PS C:\Users\Saurabh\Desktop\java folder> █
```

```
class Exception1 {
    public static void main(String[] args) {
        String a = null;
        System.out.println("Main Start");
        try {

            System.out.println(length(a));
        } catch (StringIndexOutOfBoundsException e) {
            System.out.println("Handled");
        }
        System.out.println("Main End");
    }

    public static int length(String s) {
        return s.length();
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Exception1
Main Start
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "<parameter1>" is null
    at Exception1.length(Exception1.java:15)
    at Exception1.main(Exception1.java:7)
PS C:\Users\Saurabh\Desktop\java folder> █
```

```
class Exception1 {
    public static void main(String[] args) {
        String a = "Saurabh";
        System.out.println("Main Start");
        try {

            System.out.println(length(a));
        } catch (StringIndexOutOfBoundsException e) {
            System.out.println("Handled");
        }
        System.out.println("Main End");
    }

    public static int length(String s) {
        return s.length();
    }
}
```

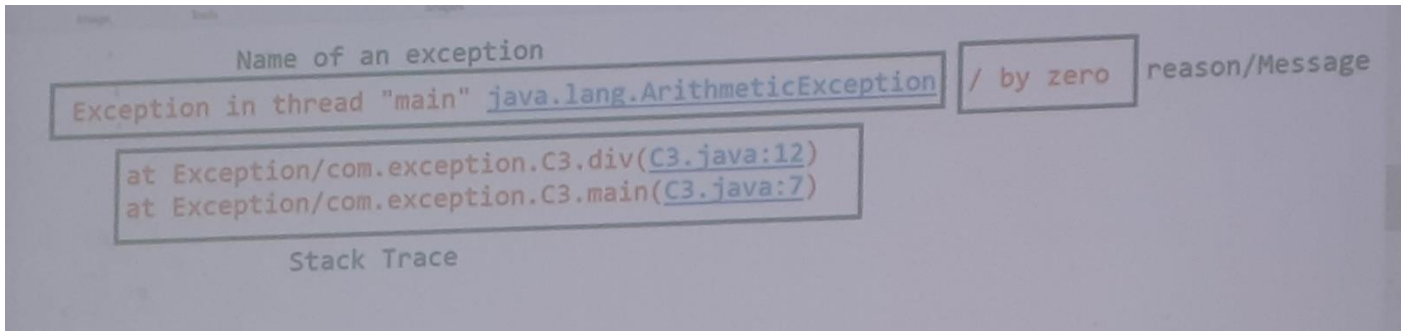
```
Main Start
```

```
7
```

```
Main End
```

```
PS C:\Users\Saurabh\Desktop\java folder> █
```

### Exception message:



### Methods of an exception object

#### Public String getMessage():

It returns the message of an exception.

#### Public void printStackTrace():

It is used to print the message name & object trace of an exception.

```
class Exception1 {  
    public static void main(String[] args) {  
        String s = null;  
        try {  
            System.out.println(s.length());  
        } catch (NullPointerException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

```
Cannot invoke "String.length()" because "<local1>" is null
```

## Types of exception

- **Unchecked Exception**
- **Checked Exception**

### Unchecked Exception

The compiler is unaware about unchecked exception & during compile time, compiler will not check the exception

E.g.: Runtime Exception & its subclass errors & its subclasses are the checked exceptions

### Checked Exception

The compiler is aware about checked exception & during compile time, compiler check the checked exception

E.g.: Interrupted exception, IO Exception, FileNotFoundException, etc.

**Note:** If there is a chance of getting checked exception, then compiler doesn't allow to compiler the program, it forces to either handle or declare the checked exception.

Handling the checked exception:

```
class Exception1 {  
    public static void main(String[] args) {  
        String s = null;  
        try {  
            Thread.sleep(4000);  
        } catch (InterruptedException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

## Declaring the checked exception

In case of not handling the checked exception, it can be declared by using **throws** keyword

### Throws:

- It is a keyword.
- It is used to declare the exception to the caller
- It should be used only in method declaration
- If the method declares exception, then the caller of the method should either handle or declare
- By using throws, any of the exception can be declared

### Syntax:

Method declaration(.....) throws E1,E2,E3,...En

```
class Exception1 {
    public static void main(String[] args) {
        String s = null;
        try {
            m1();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void m1() throws InterruptedException{
        Thread.sleep(4000);
    }
}
```

```
class Exception1 {
    public static void main(String[] args) throws InterruptedException {
        m1();
    }

    public static void m1() throws InterruptedException {
        Thread.sleep(4000);
    }
}
```

## Custom Exception

The exception that is created by the programmer is known as **custom exception** or **user defined exception**

### Steps to create custom exception

1. Create a class and make it as subclass of throwable
2. Override getMessage() method to return the message of an exception
3. Throw the exception object wherever it is needed by using the syntax below

#### Throw new constructor()

#### Throw

It is a keyword

It is used to throw the exception object externally by the programmer

Example:

```
class UnderAgeException extends ArithmeticException {
    @Override
    public String getMessage() {
        return "First grow then marry...";
    }
}

class OverAgeException extends ArithmeticException {
    @Override
    public String getMessage() {
        return "Service not reachable";
    }
}

class Metromany {
    public void VerifyAge(int age) {
        if (age < 22) {
            throw new UnderAgeException();
        } else if (age > 40) {
            throw new OverAgeException();
        } else
            System.out.println("Welcome to heaven");
    }
}

class Exception1 {
    public static void main(String[] args) {
        Metromany m = new Metromany();
        try {
            m.VerifyAge(21);
        } catch (UnderAgeException | OverAgeException e) {
```



```

        System.out.println(e.getMessage());
    }
}
}

```

```

PS C:\Users\Saurabh\Desktop\java folder> java Exception1
Welcome to heaven
PS C:\Users\Saurabh\Desktop\java folder> javac Exception1.java
PS C:\Users\Saurabh\Desktop\java folder> java Exception1
First grow then marry...
PS C:\Users\Saurabh\Desktop\java folder> 

```

### Finally Block:

**Finally Block** is used along with try and catch or only with try block

**Finally Block** block is used to write the cleanup codes.

Finally blocks will execute in all the below mentioned situations

1. Exception occurs
2. Exception occurs handled
3. Exception not occurs not handled

Example:

```

class Exception1 {
    public static void main(String[] args) {
        try {
            int[] arr = { 1, 2, 3 };
            System.out.println(arr[10]);
        } catch (StringIndexOutOfBoundsException e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("Finally");
        }
    }
}

```

```

PS C:\Users\Saurabh\Desktop\java folder> java Exception1
Finally
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 3
    at Exception1.main(Exception1.java:5)
PS C:\Users\Saurabh\Desktop\java folder> 

```

## Collection

### Wrapper class:

A wrapper Class is used to convert the primitive to non-primitive and non-primitive back to the primitive.

### Boxing

- The process of wrapping the primitive type data to non-primitive type data i.e. wrapper object is known as boxing.
- This boxing can be performed implicitly i.e. automatically. Hence it is also known as Autoboxing.
- From 1.9 version of JDK, boxing can be performed.

### Unboxing

- The process of unwrapping non-primitive type data i.e. wrapper object back to the primitive is known as unboxing.
- From 1.9 version of JDK unboxing can be performed implicitly i.e. automatically. Hence it is known as Auto unboxing.
- To perform the boxing and unboxing for all the primitive data types dedicated classes are created in java and those classes are known as wrapper class.

byte	->	Byte
short	->	Short
int	->	Int
long	->	Long
float	->	Float
double	->	Double
char	->	Char
boolean	->	Boolean

## Collection framework

### Limitations of Array:

- Array accept only homogeneous type values.
- The size of an array is fixed.
- Inserting and removing an element in array is not possible because it affects the size.
- Array is one data structure hence multiple elements cannot be stored by using different data structures in array.
- **Note:** To overcome these limitations, we can use Collection framework to store multiple elements.

### Collection :

Collection framework contains set of classes and interfaces. That is used to store multiple objects together.

### Characteristics

- In collection framework only non-primitive values are allowed. It is used to store the heterogeneous object together
- The size is not fixed.
- Inserting and removing the multiple objects is possible
- In collection framework. **CRUD** Operation can be performed.

### Root interfaces of collection:

Collection has two root interfaces

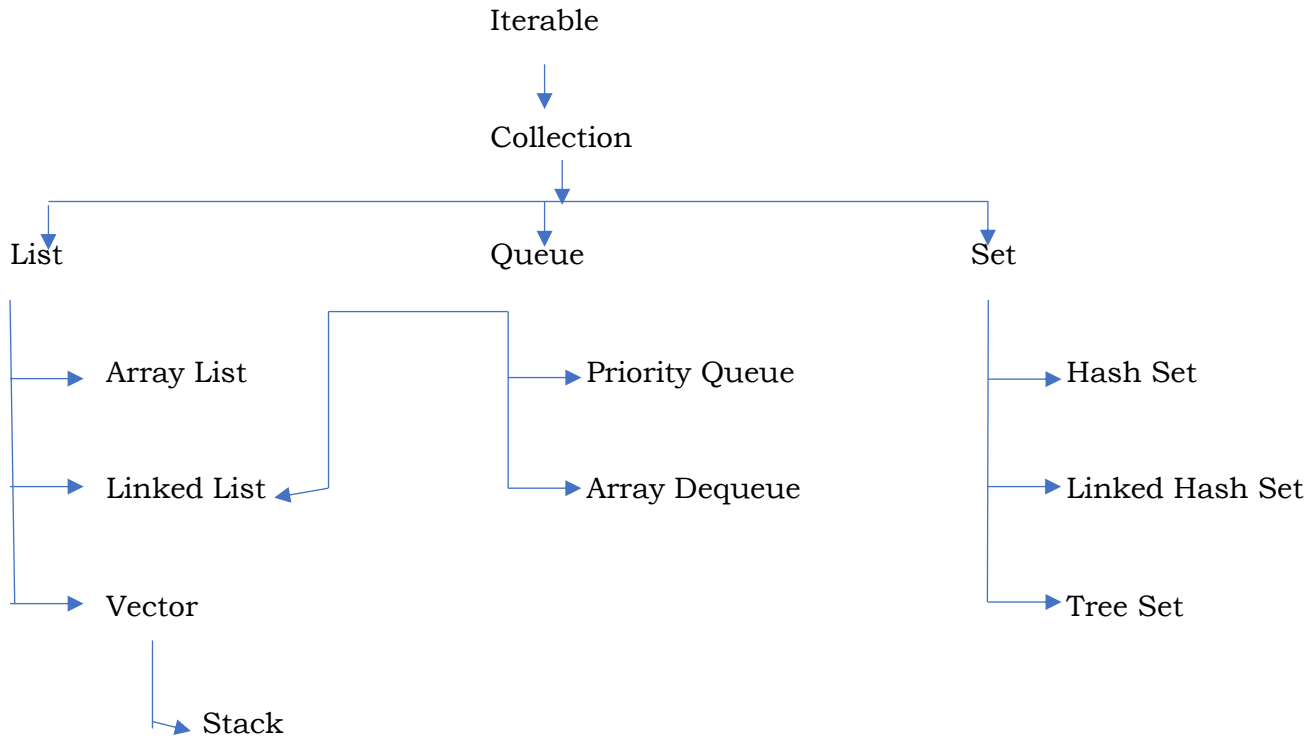
**Java.util.Collection**

**Java.util. Map**

## Java.util.Collection

- It is an interface defined in java.util.package
- It is used to store multiple objects
- List, Queue and set are the sub interfaces of collection
- By using collection interface CRUD operations are performed on objects

### Collection Hierarchy:



## Abstract methods of java.util.Collection

Function	Method Name	Return type
ADDING	add(Object o)	boolean
	addAll(Collection c)	boolean
REMOVING	remove(Object o)	boolean
	removeAll(Collection c)	boolean
	retainAll(Collection c)	boolean
	clear	void
ACCESSING	iterator()	iterator
SEARCHING	contains(Object o)	boolean
	containsAll(Collection c)	boolean
OTHERS	size()	int
	isEmpty()	boolean
	toArray()	Object[]
	hashCode()	int
	toString()	string
	equals(Object o)	boolean

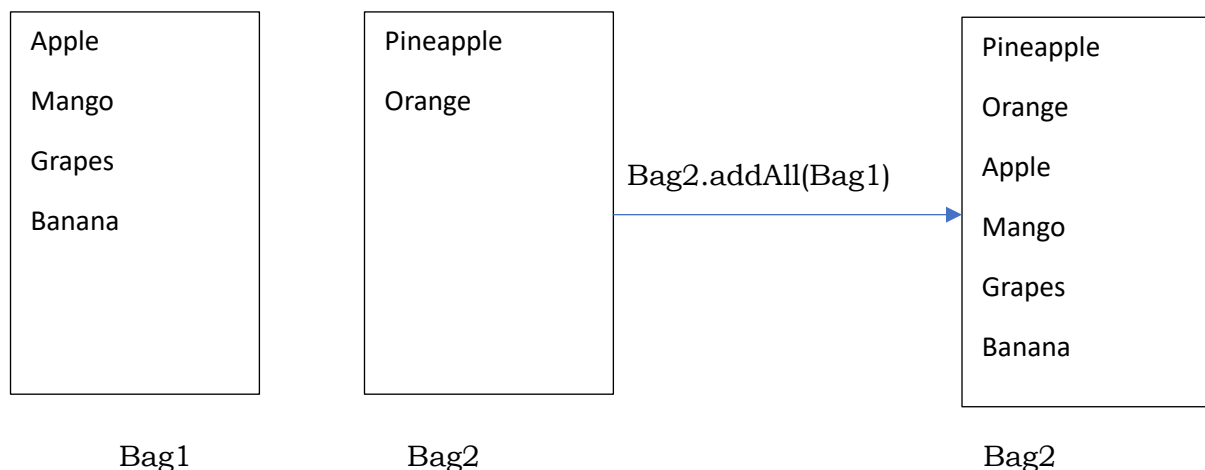
➤ **public abstract boolean add(Object o):**

It is used to add an object to the collection

➤ **public abstract boolean addAll(Collection c):**

It is used to add a group of objects from one collection to another collection

### Example:

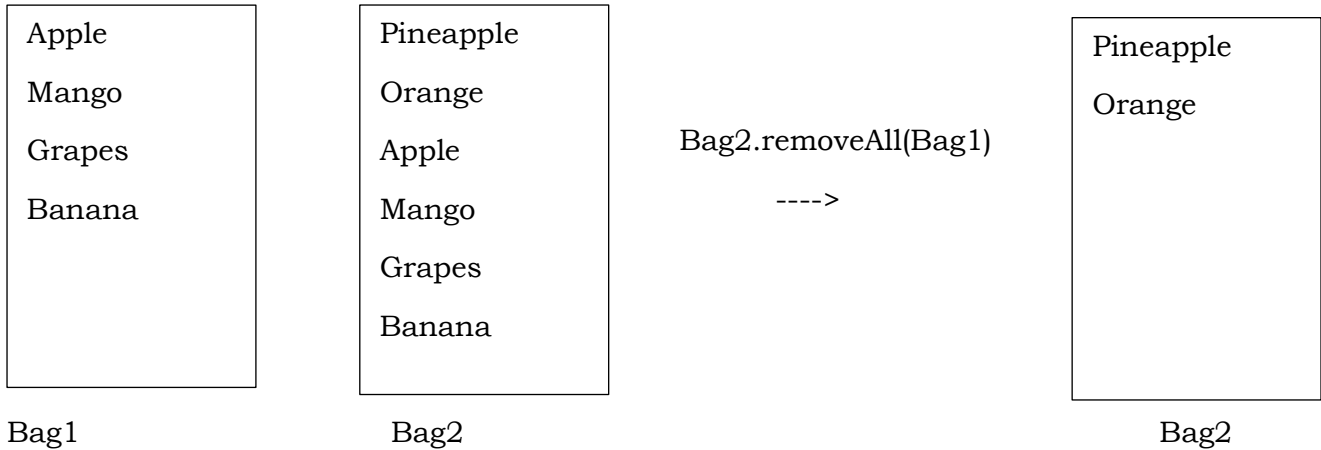


➤ **public abstract boolean remove(Object o):**

It is used to remove an object from the collection

➤ **public abstract boolean removeAll(Collection c):**

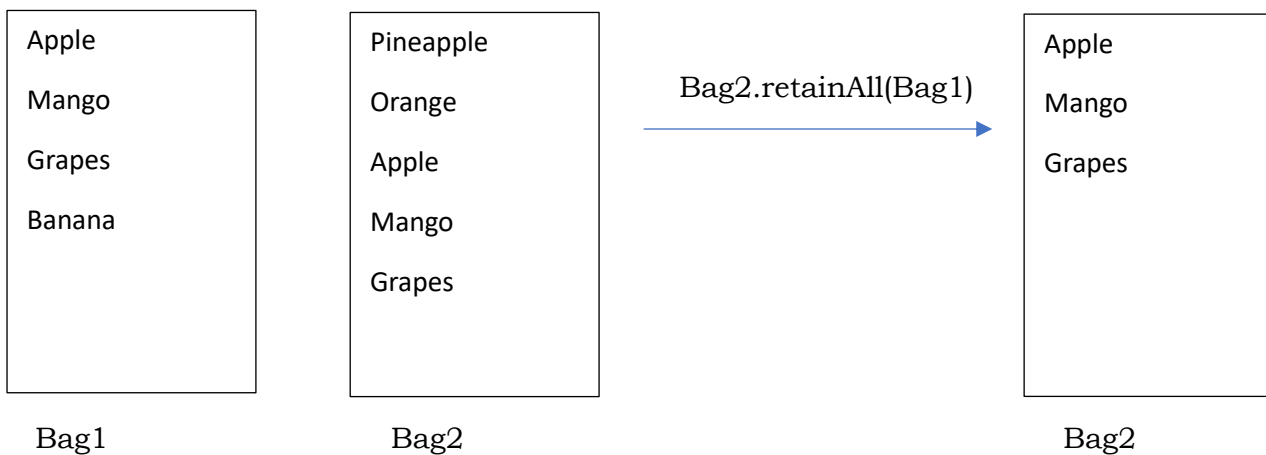
It is used to remove all the objects from one collection that are available in another collection



➤ **Public abstract Boolean retainAll(Collection c)**

It is used to retain all the objects in one collection that are available in another collection

Other than the retained objects, remaining objects are removed



➤ **Public abstract void clear();**

It is used to clear all the object in the collection

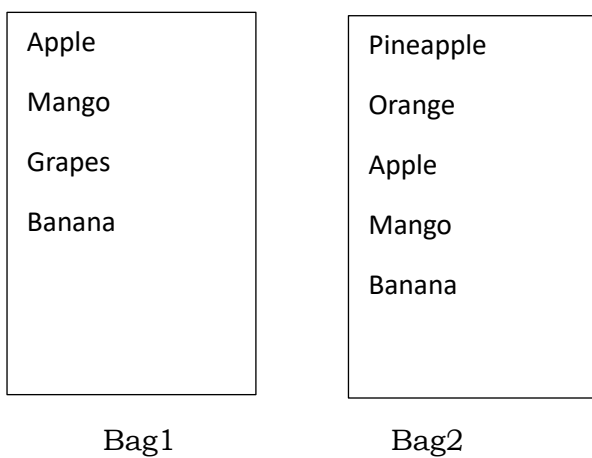
➤ **Public abstract Boolean contains(Objects o)**

It is used to search an object in the collection

If the collection contains an object, it returns true else it returns false

➤ **Public abstract Boolean containsAll(Objects o)**

If one collection contains all the objects of another collection if return true else it returns false



Bag2.containsAll(Bag2); // false

// banana is not available in Bag2

- **List**

### **Java.util.List;**

- It is the sub interface of java.util.Collection
- List contains the methods inherited from the collection and also its own methods
- It is used to store multiple objects together.
- It is introduced in 1.2 version of JDK.

### **Characteristics:**

#### **NIDHI**

- 1) Any number of **nulls** can be inserted
- 2) In a list **insertion order** of an object is maintained
- 3) **Duplicate** objects are allowed
- 4) **Heterogeneous** objects can be inserted
- 5) Objects can be accessed by using an **Index**

### **Abstract methods of List Interface**

List contains the methods which are inherited from java.util.Collection interface and also its own declared methods that are listed below:

<b>Function</b>	<b>Method Name</b>	<b>Return type</b>
<b>ADDING</b>	add(int index, object o)	void
	addAll(int index, Collection c)	boolean
<b>REMOVE</b>	remove(int index)	boolean
<b>ACCESS</b>	get(int index)	object
	listIterator()	ListIterator
	listIterator(int index)	ListIterator
<b>SEARCH</b>	indexOf(Object)	int
	lastIndexOf(object)	int
<b>REPLACE</b>	Set( int index, Object o)	Object

➤ **public abstract void add(int index, Object) :**

It is used to add an object at the specified index.

➤ **public abstract void addAll(int index, Collection) :**

It is used to add all the objects from one Collection to another collection from the specified index.



➤ **public abstract object get(int index) :**

it is used to return the object from the specified index.

➤ **Public abstract int indexOf(object) :**

It is used to return the index of the object that has the first occurrence

It starts searching from beginning to the last.

If the object is not present then it returns -1;

➤ **public abstract int LastIndexOf(object) :**

it is used to return the index of an object that has the last occurrence.

It starts searching from the end.

If the object is not available in the collection both **indexOf()** and **LastIndexOf()** method return **-1**.

## ArrayList

- **ArrayList** is a concrete implementing class of list and collection interface.
- All the methods of collection and list interface will be inherited to ArrayList.
- The characteristics of ArrayList is same as a list.
- ArrayList class is present in java.util package.
- To use the methods of collection and list we need to create an instance for ArrayList.

```
package programs;

import java.util.ArrayList;

public class String1 {

    public static void main(String[] args) {

        ArrayList al = new ArrayList();
        al.add(100);
        al.add("@");
        al.add(true);
        al.add("Java");

        System.out.println(al);

        al.add(null);

        System.out.println(al);

        ArrayList a2 = new ArrayList();

        System.out.println(a2);
        a2.addAll(al);
        a2.add(2, 7.5);

        System.out.println(a2);

    }
}
```

```
Storage\fe93013c8038e4779905417218914218\
[100, @, true, Java]
[100, @, true, Java, null]
[]
[100, @, 7.5, true, Java, null]
PS E:\Q-Notes\JAVA>
```

```

package programs;

import java.util.ArrayList;

public class String1 {

    public static void main(String[] args) {

        ArrayList al = new ArrayList();
        al.add("Hello");
        al.add('a');
        al.add(3.0);
        al.add(false);

        System.out.println(al);

        ArrayList a2 = new ArrayList();

        System.out.println(a2);
        a2.add(true);
        a2.add(500);

        a2.addAll(1, al);

        System.out.println(a2);

    }
}

```

```

C:\Program Files\Java\jdk-1.8.0_101\bin>java String1
[Hello, a, 3.0, false]
[]
[true, Hello, a, 3.0, false, 500]
PS E:\Q-Notes\JAVA>

```

```
package programs;

import java.util.ArrayList;

public class String1 {

    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList();

        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        al.add(null);

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

### **Java.util.ArrayList :**

It is the subclass of java.util.List interface.

ArrayList class is introduced in 1.2 version of JDK

It is used to store multiple objects in the sequential order

It is implemented by using growable Array / Resizable Array data structure



## Characteristics:

### NIDHI

- 1) Any number of nulls can be inserted
- 2) In a list insertion order of an object is maintained
- 3) Duplicate objects are allowed
- 4) Heterogeneous objects can be inserted
- 5) Objects can be accessed by using an Index

## Constructors of Array List:

- 1) **ArrayList()** : It is used to create one **empty ArrayList** object with default initial capacity of 10.

**ArrayList al = new ArrayList();**

--	--	--	--	--	--	--	--	--	--

- 2) **ArrayList(Collection c)** : It is used to create a new ArrayList object that is initialized with the objects in the specified Collection.

**ArrayList al2 = new ArrayList( al1 );**

**Al ->**

10	20	30	40	50
----	----	----	----	----

- 3) **ArrayList ( int initialCapacity)** : it is used to create a new empty ArrayList object with specified initial capacity

**ArrayList al = new ArrayList( 5 );**

**Al ->**

--	--	--	--	--

## Example :

```
import java.util.ArrayList;

public class Store {
    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
    }
}
```

```
        al.add(50);

        System.out.println(al); // al.toString() --> [10,20,30,40,50]
    }
}
```

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

o/p :

```
Note: Recompile with -Xlint:unchecked for details
● PS C:\Users\Saurabh\Desktop\java folder> java S
[10, 20, 30, 40, 50]
○ PS C:\Users\Saurabh\Desktop\java folder> 
```

## Example

```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();
        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        System.out.println(al1);

        ArrayList al2 = new ArrayList<>();
        al2.add("car");
        al2.add("Cycle");
        al2.add("UFO");

        System.out.println(al1.contains("Bike"));
        System.out.println(al1.containsAll(al2));
        System.out.println(al1.indexOf("Bus"));
        System.out.println(al1.lastIndexOf("Bus"));

    }
}
```

**o/p:**

```
PS C:\Users\Saurabh\Desktop
[Car, Bus, Bike, Cycle, Bus]
true
false
1
4
Live Share  Java: Ready
```

## Example

```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();

        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        ArrayList al2 = new ArrayList<>();

        al2.add("Car");
        al2.add("Cycle");
        al2.add("UFO");
        al2.add("Rocket");

        al2.retainAll(al1);
        System.out.println(al2);

    }
}
```

**o/p :**

```
PS C:\Users\Saurabh\Desktop\java folder> java Remove.java
Note: Remove.java uses unchecked or unsafe op
Note: Recompile with -Xlint:unchecked for det
PS C:\Users\Saurabh\Desktop\java folder> java Remove.java
[Car, Cycle, UFO]
PS C:\Users\Saurabh\Desktop\java folder>
```



```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();

        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        System.out.println(al1.get(0));
        System.out.println(al1.get(1));
        System.out.println(al1.get(2));
        System.out.println(al1.get(3));
        System.out.println(al1.get(4));
        System.out.println(al1.get(5));

    }
}
```

```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();

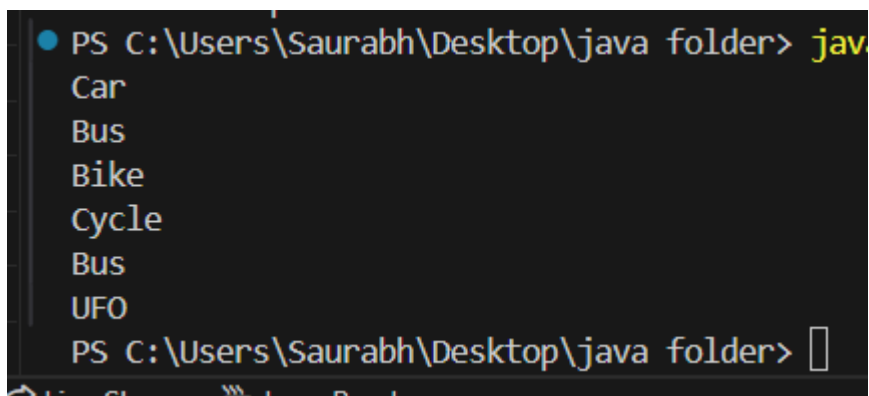
        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        int size = al1.size();

        for (int i = 0; i < size; i++) {
            System.out.println(al1.get(i));
        }

    }
}
```

**O/P :**



```
PS C:\Users\Saurabh\Desktop\java folder> javac
Car
Bus
Bike
Cycle
Bus
UFO
PS C:\Users\Saurabh\Desktop\java folder> 
```

### **Public abstract Iterator iterator( ) :**

- It is used to create the object of java.util.Iterator type.
- it is used to the objects in the collection
- it creates java.util.Iterator type object and returns the address.
- It is used to traverse or iterate the objects in the collection
- It is declared in java.util.Iterable Interface

### ➤ **Java.util.Iterator**

- It is an interface that is defined in java.util package
- It is used to Iterate or Traverse the objects on the collection

### **Abstract methods of Iterator:**

#### ➤ **Public abstract Object next():**

It is used to return the object that is present next to the cursor and it moves the cursor to the next position

#### ➤ **Public abstract Boolean hasNext() :**

- It is used to check whether any object is present next to the cursor or not
- If the object is available next to the cursor, it returns true else, it returns false

#### ➤ **Public abstract Boolean remove() :**

It is used to remove the object that is recently/ currently iterated by next() method.

It should be called only after calling the next method.

#### **Note :**

During Iteration if we try to modify the actual collection by using collection method then we get concurrent modification exception

- **ConcurrentModificationException**

If the remove method is called without calling the next method, then we get illegal state exception

- **IllegalStateException**

## Access :

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

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        Iterator i = al.iterator();
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
    }
}
```

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

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        Iterator i = al.iterator();
        while (i.hasNext()){
            System.out.println(i.next());
        }
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java A
10
20
30
40
50
```

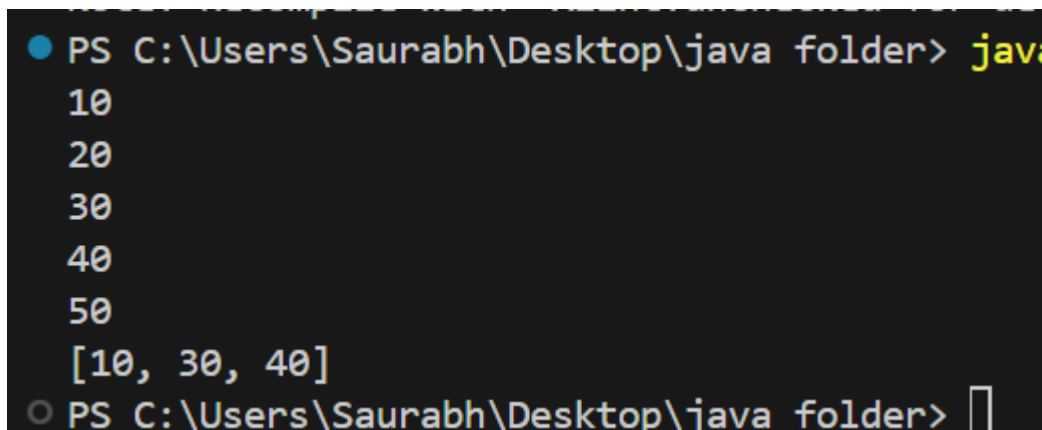
## Remove :

```
import java.util.ArrayList;
import java.util.Iterator;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        Iterator i = al.iterator();
        System.out.println(i.next());
        System.out.println(i.next());
        i.remove();
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
        i.remove();
        System.out.println(al);
    }
}
```



The screenshot shows a command prompt window with the following output:

```
PS C:\Users\Saurabh\Desktop\java folder> java
10
20
30
40
50
[10, 30, 40]
PS C:\Users\Saurabh\Desktop\java folder>
```

The output demonstrates that the elements 20 and 50 were removed from the original ArrayList [10, 20, 30, 40, 50], leaving the final state as [10, 30, 40].

## Limitations of java.util.Iterator;

- Iterating more than one time is not possible
- By using Iterator object iteration/traversing can be done only one time.
- Starting the iteration from specified position is not possible.
- Iteration can be done in forward direction.
- Traversing in backward direction is not possible.
- During iteration adding and replacing the object is not possible.
- To overcome all the limitations of **Iterator**, java.util.ListIterator can be used

➤ **ListIterator**

➤ **Public ListIterator listIterator();**

- It is used to create object of ListIterator type.
- It is defined in java.util.List Interface.
- It places the cursor in 0<sup>th</sup> position In ListIterator object

➤ **Public ListIterator listIterator( int cursorPosition );**

- It is used to create object of ListIterator type
- It is declared in java.util.List Interface.
- It places the cursor in specified position In ListIterator object.

**Java.util.ListIterator**

- It is sub interface of java.util.iterator interface
- It is used to traverse the list.

**Characteristics :**

- By using ListIterator traversing can be done in both direction
- By using ListIterator, any number we can traverse by moving the cursor in both the direction
- In ListIterator adding and replacing during Iteration is possible

**Methods:**

List Iterator contains the methods that are inherited from java.util.Iterator and also its own declared methods

**Inherited Abstract methods**

- Public abstract object next()
- Public abstract boolean hasNext()
- Public abstract void remove()

**Declared methods**

➤ **Public abstract object previous();**

It is used to return the object that is present previous to the cursor and it moves the cursor to the previous position

➤ **Public abstract boolean hasPrevious();**

It is used to check whether any object is present previous to the cursor or not

If the object is available previous to the cursor else it returns false

➤ **Public abstract void add( Object o ):**

It is used to add the object in collection during iteration in the cursor pointing position

➤ **Public abstract void set(Object o):**

It is used to replace the object that is currently iterated by next() or previous() method.

**Access:**

```
package samplepack;

import java.util.ArrayList;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(15);
        al.add(32);
        al.add(16);
        al.add(14);

        ListIterator li = al.listIterator();

        System.out.println(li.next());
        System.out.println(li.next());
        System.out.println(li.next());
        System.out.println(li.previous());
        System.out.println(li.previous());

        li.set(45); // 20 is replaced with 45
        li.set(50); // 45 is replaced with 50
        li.set(11); // 50 is replaced with 11

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

**O/P:**

```
10
20
15
15
20
[10, 45, 15, 32, 16, 14]
```

```

package samplepack;

import java.util.ArrayList;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(15);
        al.add(32);
        al.add(16);
        al.add(14);

        ListIterator li = al.listIterator(al.size());

        while (li.hasPrevious()) {
            System.out.println(li.previous());
        }

    }
}

```

**O/P:**

```

14
16
32
15
20
10

```

```

import java.util.ArrayList;
import java.util.ListIterator;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        ListIterator li = al.listIterator();
        while (li.hasNext()) {
            System.out.println(li.next());
        }
    }
}

```



```

    }

    while (li.hasPrevious()) {
        System.out.println(li.previous());
    }
}
}

```

```

● PS C:\Users\Saurabh\Desktop\java folder> java Access
10
20
30
40
50
+++++
50
40
30
20
10
PS C:\Users\Saurabh\Desktop\java folder>

```

### With position

```

import java.util.ArrayList;
import java.util.ListIterator;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        ListIterator li = al.listIterator(4);
        System.out.println(li.next());
    }
}

```

```

● PS C:\Users\Saurabh\Desktop\java folder> java Access
50
PS C:\Users\Saurabh\Desktop\java folder>

```

## Replace:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Replace {
    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        ListIterator li = al.listIterator(3);
        System.out.println(li.previous());
        System.out.println(li.previous());
        System.out.println(li.previous());
        li.set(75);
        System.out.println(li.next());

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

```
PS C:\Users\Saurabh\Desktop\java folder> java Replace
30
20
10
75
[75, 20, 30, 40, 50]
PS C:\Users\Saurabh\Desktop\java folder> 
```

### Example:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Access {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Saurabh");
        names.add("Ketan");
        names.add("Chetan");
        names.add("Omkar");
        names.add("Vikas");

        ListIterator<String> li = names.listIterator(names.size());
        while (li.hasPrevious()) {
            String name = li.previous();

            if (name.equals("Vikas")) {
                li.set("Vicky");
            }
        }

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

### O/P :

```
● PS C:\Users\Saurabh\Desktop\java folder> javac Access.java
● PS C:\Users\Saurabh\Desktop\java folder> java Access
[Saurabh, Ketan, Chetan, Omkar, Vicky]
PS C:\Users\Saurabh\Desktop\java folder>
```

### Example:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Access {
    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        al.add(60);
        al.add(30);

        ListIterator li = al.listIterator();
        while (li.hasNext()) {
            if ((Integer) li.next() == 30) {
                li.set(60);
                break;
            }
        }
        System.out.println(al);
    }
}
```

### O/P:

```
Note: Access.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
● PS C:\Users\Saurabh\Desktop\java folder> java Access
[20, 60, 40, 50, 60, 30]
PS C:\Users\Saurabh\Desktop\java folder> 
fwd-i-search: _
```

### ➤ For each loop:

For each loop/ advanced for loop is used only for array and collection

#### Syntax :

```
for ( datatype VarName : collection/Array){  
    // instructions  
}
```

- The type of variable declared in for each loop should be similar to the type of collection or arrays
- The number of iterations of for each loop is always equals to the numbers of elements of array or collection

#### Workflow:

For every iteration one element is taken from the specified collection or array in a sequential order and store in the variable declared in for each loop.

#### Example:

```
import java.util.ArrayList;  
  
public class Access {  
    public static void main(String[] args) {  
        ArrayList<Integer> al = new ArrayList<>();  
        al.add(20);  
        al.add(30);  
        al.add(40);  
        al.add(50);  
        al.add(60);  
        al.add(30);  
  
        for (Object o : al) {  
            System.out.println(o);  
        }  
    }  
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Access  
20  
30  
40  
50  
60  
30  
PS C:\Users\Saurabh\Desktop\java folder> █
```

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Access {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Saurabh");
        names.add("Ketan");
        names.add("Chetan");
        names.add("Omkar");
        names.add("Vikas");

        for (Object o : names) {
            System.out.println(o);
        }
    }
}
```

```
● PS C:\Users\Saurabh\Desktop\java folder> java Access
Saurabh
Ketan
Chetan
Omkar
Vikas
PS C:\Users\Saurabh\Desktop\java folder> 
```

```

package samplepack;

import java.util.ArrayList;
import java.util.Iterator;

public class Student {
    String name;
    int id;

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

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(new Student("Sheela", 25));
        al.add(new Student("Laila ", 7));
        al.add(new Student("Sheela", 25));
        al.add(new Student("Sindhu", 3));
        al.add(new Student("Sheela", 25));
        al.add(new Student("Laila ", 7));

        // accessing

        // Access one object
        Student s = (Student) al.get(1);
        System.out.println(s.name);
        System.out.println(s.id);

        // Access all the objects
        for (Object o : al) {
            Student stu = (Student) o;
            System.out.println(stu.name);
        }
        System.out.println();
        System.out.println("+++++++");
        // Searching

        boolean exist = false;
        for (Object o : al) {
            Student st = (Student) o;
            if (st.name == "Sheela") {
                exist = true;
                break;
            }
        }
        if (exist) {
            System.out.println("found");
        } else {
            System.out.println("Not found");
        }
    }
}

```

```

        System.out.println();
        System.out.println("++++++++++++++++");
        // remove

        Iterator i = al.iterator();
        while (i.hasNext()) {
            Student stu1 = (Student) i.next();
            if (stu1.name == "Sheela") {
                i.remove();
                System.out.println("Removed " + stu1.name);
                break; // only one object is removed
            }
        }
        System.out.println();
        System.out.println("++++++++++++++++");
        System.out.println(al);
    }
}

```

### O/P:

```

Laila
7
Sheela
Laila
Sheela
Sindhu
Sheela
Laila

++++++++++++++++
found

++++++++++++++++
Removed Sheela

++++++++++++++++
[Laila , Sheela, Sindhu, Sheela, Laila ]

```



```

import java.util.ArrayList;

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

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        for (Object o : al) {
            Employeee e = (Employeee) o;
            System.out.println(e.name);
            System.out.println(e.id);
            System.out.println(e.salary);
            System.out.println("-----");
        }
    }
}

```

```

PS C:\Users\Saurabh\Desktop\java folder> jav
Saurabh
1
50000.0
-----
vikas
2
49000.0
-----
Mayur
3
44000.0
-----
Omkar
4
45900.0
-----
PS C:\Users\Saurabh\Desktop\java folder> 

```

**Write a java program to search employee based on ID.**

**Code:**

```
import java.util.ArrayList;
import java.util.Iterator;

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

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        int id = 1;
        boolean exist = false;

        Iterator i = al.iterator();
        while (i.hasNext()) {
            Employeee e = (Employeee) i.next();
            if (e.id == 1) {
                System.out.println("found");
                exist = true;
            }
        }

        if (exist != true) {
            System.out.println("not found");
        }
    }
}
```

```
● PS C:\Users\Sau  
found  
○ PS C:\Users\Sau
```

**Write a java program to print name and salary of employee based on employee id.**

**Code:**

```
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class Employeee {  
    String name;  
    int id;  
    double salary;  
  
    public Employeee(String name, int id, double salary) {  
        super();  
        this.name = name;  
        this.id = id;  
        this.salary = salary;  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
  
    public static void main(String[] args) {  
        ArrayList al = new ArrayList<>();  
        al.add(new Employeee("Saurabh", 1, 50000));  
        al.add(new Employeee("vikas", 2, 49000));  
        al.add(new Employeee("Mayur", 3, 44000));  
        al.add(new Employeee("Omkar", 4, 45900));  
  
        int id = 1;  
        boolean exist = false;  
  
        Iterator i = al.iterator();  
        while (i.hasNext()) {  
            Employeee e = (Employeee) i.next();  
            if (e.id == 1) {  
                System.out.println(e.name);  
                System.out.println(e.salary);  
                exist = true;  
            }  
        }  
  
        if (exist != true) {  
            System.out.println("not found");  
        }  
    }  
}
```

- PS C:\Users\Saurabh> Saurabh  
50000.0
- PS C:\Users\Saurabh>

**Write a java program to increase salary of all the employees by 10%**

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class Employeee {

    String name;
    int id;
    double salary;

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" + "name='" + name + '\'' + ", id=" + id + ", salary=" +
salary + '}';
    }

    public void increaseSalary(int percentage) {
        this.salary += salary * percentage / 100;
    }

    public static void main(String[] args) {
        ArrayList<Employeee> al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        System.out.println("Salaries before increasing are : ");
        for (Employeee e : al) {
            System.out.println(e);
        }

        System.out.println("-----");
        for (Employeee e : al) {
            e.increaseSalary(10); // increasing the salaries by 10%
        }
    }
}
```

```
    }  
  
    System.out.println("Salaries after increasing are : ");  
    for (Employee e : al) {  
        System.out.println(e);  
    }  
  
}  
  
}
```

```
Salaries before increasing are :  
Employee{name='Saurabh', id=1, salary=50000.0}  
Employee{name='vikas', id=2, salary=49000.0}  
Employee{name='Mayur', id=3, salary=44000.0}  
Employee{name='Omkar', id=4, salary=45900.0}  
-----  
Salaries after increasing are :  
Employee{name='Saurabh', id=1, salary=55000.0}  
Employee{name='vikas', id=2, salary=53900.0}  
Employee{name='Mayur', id=3, salary=48400.0}  
Employee{name='Omkar', id=4, salary=50490.0}
```

## Types of Collection

### 1. Non- generic collection

- Non- generic collection accepts Object type data
- In non-generic collection by default all the objects are upcasted to java.lang.Object type.

### 2. Generic collection

- Generic collection accepts only homogeneous data's.
- In generic collection objects are not upcasted to java.lang.Object type.

#### Syntax:

```
datatype <NPDT> var = new Constructor<NPDT>();  
  
OR  
  
datatype <NPDT> var = new Constructor<>();  
  
OR  
  
datatype <NPDT> var = new Constructor();
```

#### Example:

```
package samplepack;  
  
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class Student {  
    String name;  
    int id;  
  
    public Student(String name, int id) {  
        super();  
        this.name = name;  
        this.id = id;  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
  
    public static void main(String[] args) {  
        ArrayList<Student> al = new ArrayList();  
        al.add(new Student("Sheela", 25));  
        al.add(new Student("Laila ", 7));  
        al.add(new Student("Sheela", 25));  
        al.add(new Student("Sindhu", 3));  
        al.add(new Student("Sheela", 25));  
        al.add(new Student("Laila ", 7));  
  
        // Access all the objects
```

```

        for (Student s : al) {
            System.out.println(s.name);

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

        // Searching

        boolean exist = false;
        for (Student s1 : al) {
            if (s1.name == "Sheela") {
                exist = true;
                break;
            }

        }

        if (exist) {
            System.out.println("found");
        } else
            System.out.println("Not found");

        System.out.println();
        System.out.println("++++++++++++++++");
        // remove

        Iterator i = al.iterator();
        while (i.hasNext()) {
            Student stu1 = (Student) i.next();
            if (stu1.name == "Sheela") {
                i.remove();
                System.out.println("Removed " + stu1.name);
                break; // only one object is removed
            }

        }
        System.out.println();
        System.out.println("++++++++++++++++");
        System.out.println(al);
    }
}

```

**Write a java program to remove the employee based on their Id**

**Ans:**

```

package pack.subpack;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class Employeee {

    String name;
    int id;
    double salary;
}

```

```

public Employee(String name, int id, double salary) {
    super();
    this.name = name;
    this.id = id;
    this.salary = salary;
}

@Override
public String toString() {
    return "Employee{" + "name='" + name + '\'' + ", id=" + id + ", salary=" +
salary + '}';
}

public void increaseSalary(int percentage) {
    this.salary += salary * percentage / 100;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    ArrayList<Employee> al = new ArrayList<>();
    al.add(new Employee("Saurabh", 1, 50000));
    al.add(new Employee("vikas", 2, 49000));
    al.add(new Employee("Mayur", 3, 44000));
    al.add(new Employee("Omkar", 4, 45900));
    System.out.println("Employees List : ");
    for (Employee e : al) {
        System.out.println(e);
    }

    System.out.println("- - - - -");
    System.out.print("Enter ID to remove : ");

    int removeId = sc.nextInt();
    Iterator i = al.iterator();
    while (i.hasNext()) {
        Employee e = (Employee) i.next();
        if (e.id == removeId) {
            i.remove();
            System.out.println("Employee with ID : " + removeId + " has
Removed");
            break;
        }
    }
    System.out.println("- - - - -");
    System.out.println("Updated Employees List : ");
    for (Employee e : al) {
        System.out.println(e);
    }
}
}

```



```
Employees List :  
Employee{name='Saurabh', id=1, salary=50000.0}  
Employee{name='vikas', id=2, salary=49000.0}  
Employee{name='Mayur', id=3, salary=44000.0}  
Employee{name='Omkar', id=4, salary=45900.0}
```

```
- - - - -  
Enter ID to remove : 2  
Employee with ID :2 has Removed
```

```
- - - - -  
Updated Employees List :  
Employee{name='Saurabh', id=1, salary=50000.0}  
Employee{name='Mayur', id=3, salary=44000.0}  
Employee{name='Omkar', id=4, salary=45900.0}
```

## Sorting of the list:

List can be sorted by using java.util.collections sort method.

### public static void sort(list l):

It is used to sort the object in the list.

This method follows the natural sorting order.

It sort the objects only when the objects are java.lang.Comparable type.

#### ➤ Comparable Interface

### java.lang.Comparable:

- It is an Interface that is defined in java.lang package.
- Since it contains only one abstract method ,It is the functional Interface.

### Abstract Methods of Comparable:

#### ➤ Public abstract int compareTo(Object o)

**Note:** To sort any object in the collection it should be of java.lang.Comparable type.

```
package samplepack;

import java.util.ArrayList;
import java.util.Collections;

public class Main {

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(15);
        al.add(10);
        al.add(32);
        al.add(16);
        al.add(20);
        al.add(14);

        Collections.sort(al);

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

```
[10, 14, 15, 16, 20, 32]
```

```

package samplepack;

import java.util.ArrayList;
import java.util.Collections;

public class Student implements Comparable<Student> {
    String name;
    int id;

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

    public String toString() {
        return name;
    }

    @Override
    public int compareTo(Student s) {
        if (id > s.id) {
            return 3;
        } else if (id < s.id) {
            return -1;
        } else
            return 0;
    }

    public static void main(String[] args) {
        ArrayList<Student> al = new ArrayList();
        al.add(new Student("Saurabh", 14));
        al.add(new Student("Govind", 45));
        al.add(new Student("Mayur", 41));
        al.add(new Student("Vedant", 24));
        al.add(new Student("Vikas", 69));

        Collections.sort(al);

        System.out.println(al);
    }
}

```

```
[Saurabh, Vedant, Mayur, Govind, Vikas]
```

- If the current object state is greater than passed object state should return positive integer
- If the current object state is less than passed object state it should return negative integer
- If the current object state is same to the passed object state it should return zero

### Sorting the products based on the price:

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;

public class Products implements Comparable {
    String name;
    String brand;
    int pid;
    double price;

    public Products(String name, String brand, int pid, double price) {
        super();
        this.name = name;
        this.brand = brand;
        this.pid = pid;
        this.price = price;
    }

    public String toString() {
        return name;
    }

    public int compareTo(Object o) {
        Products p = (Products) o;
        if (pid > p.pid) {
            return 96;
        } else if (pid < p.pid) {
            return -89;
        } else {
            return 0;
        }
    }

    public static void main(String[] args) {
        ArrayList<Products> al = new ArrayList();
        al.add(new Products("Mobile", "Samsung", 83989, 70000.0));
        al.add(new Products("Light", "Philips", 5968, 1200));
        al.add(new Products("Shoe", "Bata", 839, 1500));
        al.add(new Products("Laptop", "HP", 89, 70000.0));

        Collections.sort(al);
        System.out.println(al);
    }
}
```

```
[Laptop, Shoe, Light, Mobile]
```

## Sort the students based on Rank

```
package samplepack;

import java.util.ArrayList;
import java.util.Collections;

public class Student implements Comparable<Student> {
    String name;
    int id;

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

    public String toString() {
        return name;
    }

    @Override
    public int compareTo(Student s) {
        return name.compareTo(s.name);
    }

    public static void main(String[] args) {
        ArrayList<Student> al = new ArrayList<Student>();
        al.add(new Student("Saurabh", 14));
        al.add(new Student("Govind", 45));
        al.add(new Student("Mayur", 41));
        al.add(new Student("Vedant", 24));
        al.add(new Student("Vikas", 69));

        Collections.sort(al);

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

```
[Govind, Mayur, Saurabh, Vedant, Vikas]
```

## Sort the students based on Rank

```
package pack.subpack;

import java.lang.Comparable;
import java.util.ArrayList;
import java.util.Collections;

public class Students implements Comparable {
    String name;
    int rollNo;
    int rank;

    public Students(String name, int rollNo, int rank) {
        super();
        this.name = name;
        this.rollNo = rollNo;
        this.rank = rank;
    }

    public String toString() {
        return name;
    }

    public int compareTo(Object o) {
        Students s = (Students) o;
        if (rank > s.rank)
            return 89;
        else if (rank < s.rank)
            return -89;
        else
            return 0;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(new Students("Saurabh", 226, 1));
        al.add(new Students("Govind", 275, 3));
        al.add(new Students("Mayur", 265, 5));
        al.add(new Students("Tanmay", 245, 4));
        al.add(new Students("Pratik", 285, 2));

        Collections.sort(al);
        System.out.println(al);
    }
}
```

```
<terminated> Students [pava Application] C:\Program Files\Java\jdk-20\bin\java.exe
[Saurabh, Pratik, Govind, Tanmay, Mayur]
```

## Sort the Book based on title

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;

public class Book implements Comparable {
    String title;
    String author;
    int pages;

    public Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    @Override
    public String toString() {
        return title;
    }

    @Override
    public int compareTo(Object o) {
        Book b = (Book) o;

        return title.compareTo(b.title);
    }

    public static void main(String[] args) {
        ArrayList<Book> al = new ArrayList<>();

        al.add(new Book("The Animal love", "Darshan", 561));
        al.add(new Book("Rules to become Romeo", "Saurabh", 541));
        al.add(new Book("Men will be Women ", "Vedant", 511));
        al.add(new Book("Must read in your 18", "Shiv", 461));

        Collections.sort(al);
        System.out.println(al);
    }
}
```

<terminated> Book [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 2, 2024, 8:38:36 AM – 8:38:40 AM) [pid: 12524]

[Men will be Women , Must read in your 18, Rules to become Romeo, The Animal love]

## Sort the bags based on colour

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;

public class Bag implements Comparable<Bag> {
    String color;
    int compartments;

    public Bag(String color, int compartments) {
        super();
        this.color = color;
        this.compartments = compartments;
    }

    @Override
    public String toString() {
        return color;
    }

    @Override
    public int compareTo(Bag b) {
        return -color.compareTo(b.color);
    }

    public static void main(String[] args) {
        ArrayList<Bag> al = new ArrayList<>();
        al.add(new Bag("Red", 4));
        al.add(new Bag("Green", 5));
        al.add(new Bag("Blue", 3));
        al.add(new Bag("Pink", 6));
        al.add(new Bag("Black", 4));

        Collections.sort(al);

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

```
[Red, Pink, Green, Blue, Black]
```



**Sort(List)** method will not sort the list if it is not comparable type

We can sort only based on natural sorting order i.e. Comparable.

By using Comparable multiple sorting mechanisms can not be provide

To overcome all these limitation we can use java.util.Comparator

### ➤ **Comparator Interface**

#### **Limitations of comparable:**

- By using comparable interface objects can be sorted by using any one of the states
- To sort the objects based on different states is not possible i.e. multiple sorting option cannot be provided by comparable interface.
- If the objects are not comparable then sort method will not sort the objects in the list
- To overcome all the limitations of comparable interface java.util.Comparator can be used

#### **java.util.Comparator**

- It is an interface provided in java.util package
- It provides external sorting mechanism to sort the objects
- By using this interface any number of sorting mechanism can be provided
- It is the functional interface.
- **sort ( List l , comparator c ) – method** of collection uses the comparator interface of sorting

#### **Abstract Methods**

➤ **Public abstract int compare( Object o1, Object o2 );**

#### **Characteristics:**

- Even though objects are not comparable type by using comparator objects can be sorted
- By using comparator objects can be sorted by using multiple ways

**sort ( List l , comparator c ) - method**

**TreeSet( Comparator c ) – constructor**

**TreeMap( comparator c ) – constructor**

Uses the comparator mechanism to sort the objects

#### **Steps to sort the list by using comparator:**

1. Create a class and implement comparator interface
2. Override compare method
3. Call the **collections.sort ( )** and pass the list as well as object of comparator type class

### Example:

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public final class Products {
    String name;
    String brand;
    int pid;
    double price;

    public Products(String name, String brand, int pid, double price) {
        super();
        this.name = name;
        this.brand = brand;
        this.pid = pid;
        this.price = price;
    }

    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList<Products> al = new ArrayList<>();
        al.add(new Products("Mobile", "Samsung", 83989, 70000.0));
        al.add(new Products("Light", "Philips", 5968, 1200));
        al.add(new Products("Shoe", "Bata", 839, 1500));
        al.add(new Products("Laptop", "HP", 89, 70000.0));

        System.out.println("Sorting based on price in asc order");
        Collections.sort(al, new SortByPriceAsc());
        System.out.println(al);

        System.out.println("Sorting based on price in desc order");
        Collections.sort(al, new SortByPricedesc());
        System.out.println(al);

        System.out.println("Sorting based on brand in asc order");
        Collections.sort(al, new SortByBrandAsc());
        System.out.println(al);
    }
}

class SortByPricedesc implements Comparator<Products> {
    @Override
    public int compare(Products p1, Products p2) {
        if (p1.price > p2.price)
            return -1;
        else if (p1.price < p2.price)
            return 1;
        else
            return 0;
    }
}
```

```

    }
}

class SortByBrandAsc implements Comparator<Products> {
    @Override
    public int compare(Products p1, Products p2) {
        return p1.brand.compareTo(p2.brand);
    }
}

class SortByPriceAsc implements Comparator<Products> {
    @Override
    public int compare(Products p1, Products p2) {
        return (int) (p1.price - p2.price);
    }
}

```

```

Sorting based on price in asc order
[Light, Shoe, Mobile, Laptop]
Sorting based on price in desc order
[Mobile, Laptop, Shoe, Light]
Sorting based on brand in asc order
[Shoe, Laptop, Light, Mobile]

```

```

package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public final class Products {
    String name;
    String brand;
    int pid;
    double price;

    public Products(String name, String brand, int pid, double price) {
        super();
        this.name = name;
        this.brand = brand;
        this.pid = pid;
        this.price = price;
    }

    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList<Products> al = new ArrayList<>();
        al.add(new Products("Mobile", "Samsung", 83989, 70000.0));
        al.add(new Products("Light", "Philips", 5968, 1200));
        al.add(new Products("Shoe", "Bata", 839, 1500));
        al.add(new Products("Laptop", "HP", 89, 70000.0));

        // lambda function/ expression
        System.out.println("Sorting based on price in asc order");
        Collections.sort(al, (p1,p2)-> (int)(p1.price-p2.price));
        System.out.println(al);

        System.out.println("Sorting based on brand");
        Collections.sort(al, (p1,p2)-> (p1.brand.compareTo(p2.brand)));
        System.out.println(al);

        System.out.println("Sorting based on name");
        Collections.sort(al, (p1,p2)-> p1.name.compareTo(p2.name));
        System.out.println(al);
    }
}

```

```

<terminated> Products [Java Application] C:\Program Files\Java\jdk-20\
Sorting based on price in asc order
[Light, Shoe, Mobile, Laptop]
Sorting based on brand
[Shoe, Laptop, Light, Mobile]
Sorting based on name
[Laptop, Light, Mobile, Shoe]

```

## Linked List

### Advantage of array list

Array list is best suitable for random accessing

### Dis-Advantage of array list

In array list inserting the objects in the middle and removing the object takes more time because if any element is inserted or removed then remaining elements will shift their places/positions.

## Linked List

- It is the implementing the subclass of **java.util.List** Interface
  - It is used to store multiple objects together
  - Linked list is implemented by using **doubly linked list** data structure
  - In linked list objects are stored in the form of **nodes** and one node is linked with the previous and the next node
- 
- **Node:**
  - Node is an object
  - Every node contains an address, pointers and values
  - In doubly linked list every node contains two pointers and one value
  - One pointer is used to store the address of previous node and another pointer is used to store the address of next node

Address of previous node	Data	Address of next node
--------------------------	------	----------------------

The linked list is introduced in 1.2 vision of JDK.

### Characteristics

#### NIDHI

- 1) Any number of nulls can be inserted
- 2) In a list insertion order of an object is maintained
- 3) Duplicate objects are allowed
- 4) Heterogeneous objects can be inserted
- 5) Objects can be accessed by using an Index

#### Note:

In linked list to provide index support internally index pointer is created in doubly linked list data structure

## Constructors

➤ **LinkedList()** :

It creates an empty linked list object

➤ **LinkedList( Collection c )** :

It creates a new linked list object that is initialized with objects in the specified collection

```
LinkedList l = new LinkedList();
```

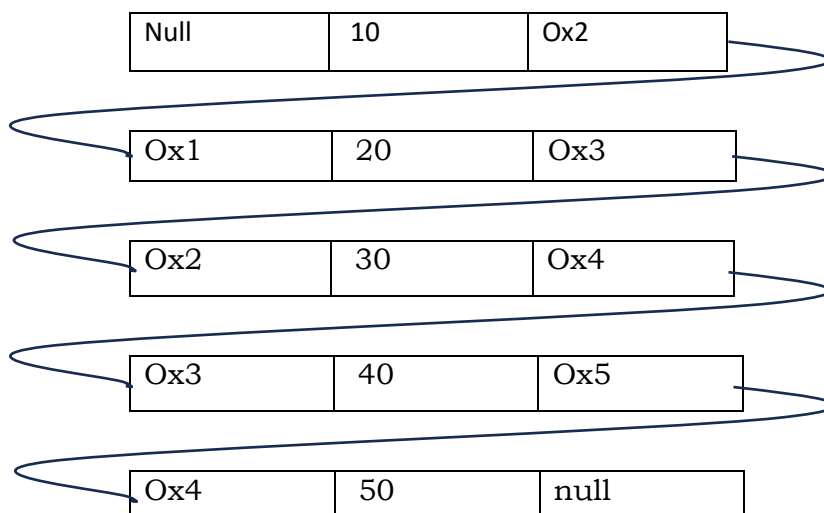
```
l.add ( 10 );
```

```
l.add ( 20 );
```

```
l.add ( 30 );
```

```
l.add ( 40 );
```

```
l.add ( 50 );
```



## Crate food object and add in the linked list object

```
package pack.subpack;

import java.util.Collections;
import java.util.LinkedList;

public class Food {
    String name;
    int price;

    public Food(String name, int price) {
        super();
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        LinkedList<Food> l = new LinkedList<>();
        l.add(new Food("Vadapav", 12));
        l.add(new Food("Dabeli", 15));
        l.add(new Food("Panipuri", 20));
        l.add(new Food("Bhel", 18));
        l.add(new Food("Samosa", 22));

        System.out.println("Sorting based on the Name:");
        Collections.sort(l, (f1, f2) -> f1.name.compareTo(f2.name));
        System.out.println(l);

        System.out.println("Sorting based on the price:");
        Collections.sort(l, (f1, f2) -> f1.price - f2.price);
        System.out.println(l);
    }
}
```

```
<terminated> Food [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May
Sorting based on the Name:
[Bhel, Dabeli, Panipuri, Samosa, Vadapav]
Sorting based on the price:
[Vadapav, Dabeli, Bhel, Panipuri, Samosa]
```

## Vector

### Java.util.Vector:

- It is an implementing class of java.util.List interface.
- It is introduced in 1.0 version hence, it is legacy class
- It is used to store multiple objects together
- It is implemented in growable array data structure
- It is synchronized (one thread can access at a time)
- It also implements

### Characteristics

#### NIDHI

- Multiple nulls can be inserted
- Heterogeneous objects are allowed
- Insertion order is maintained
- Duplicate objects are allowed
- Objects can be accessed by using index

### Constructors:

- **Vector():** It creates an empty vector object with the default initial capacity of 10.
- **Vector( Collection c ) :** It creates a new vector object that is initialized with the objects in the specified collection
- **Vector ( int initialCapacity) :** It creates a new vector object with specified capacity.
- **Vector( int initialCapacity, int incrementCapacity):**  
It creates an empty vector object with specified initial capacity and incremental capacity

### Methods:

Vector contains all the methods of list interface and also its own legacy methods such as

- **Add(Object o)**
- **Capacity(),**
- **ElementAt(int index)**
- **firstElement()**
- **insertElementAt(Object o,int index)**

**Note :** It is recommended to use ArrayList in place of vector



## Difference between Vector and ArrayList

Feature	Vector	ArrayList
Synchronization	Vector is synchronized.	ArrayList is not synchronized by default.
Performance	Relatively slower due to synchronization.	Faster compared to Vector in non-threaded environment.
Growth	Doubles its size when it exceeds capacity.	Increases by 50% of its current size when it exceeds capacity.
Legacy	Part of the original Java collections framework.	Introduced in Java 2 (JDK 1.2) as part of the Collections Framework.
Thread Safety	Thread-safe.	Not thread-safe.
Iterator	Fail-safe iterator.	Fail-fast iterator.

## Stack

### Java.util.Stack

- It is the subclass of java.util.Vector
- Stack represents first in last out or last in first out of objects
- In stack elements are added from the top of stack and removed from the top of the stack
- It is introduced in **1.0** version of **JDK**

### Constructor

- **Stack()** :     **It creates an empty stack**

### Methods

Stack contains the methods that are inherited from java.util class and its own methods.

- **Public Object push( Object o ) :**

It is used to push the object into the top of stack

- **Public Object pop() :**

It is used to remove the object at the top of the stack and it return that object as a value

- **Public Object peek() :**

It is used to return the object at the top of the stack without removing

- **Public int search ( Object o ) :**

- It returns one based position from the stack .
- If the object is not present in the stack it return **-1**

- **Public boolean empty() :**

- It is used to return to check whether the stack is empty or not empty
- If the stack is empty it return true if no it return false.

```
package pack.subpack;

import java.util.Stack;

public class Stack1 {
    public static void main(String[] args) {
        Stack<Integer> s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);
        s.push(50);

        System.out.println(s);

        s.pop();
        s.pop();

        System.out.println(s);

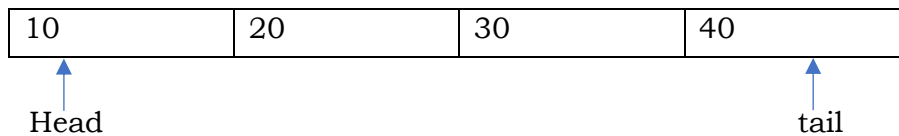
        System.out.println(s.peek());
        System.out.println(s.empty());
        System.out.println(s.search(10));
    }
}
```

```
terminated> Stack1.java Application: C:\pr...
[10, 20, 30, 40, 50]
[10, 20, 30]
30
false
3
```

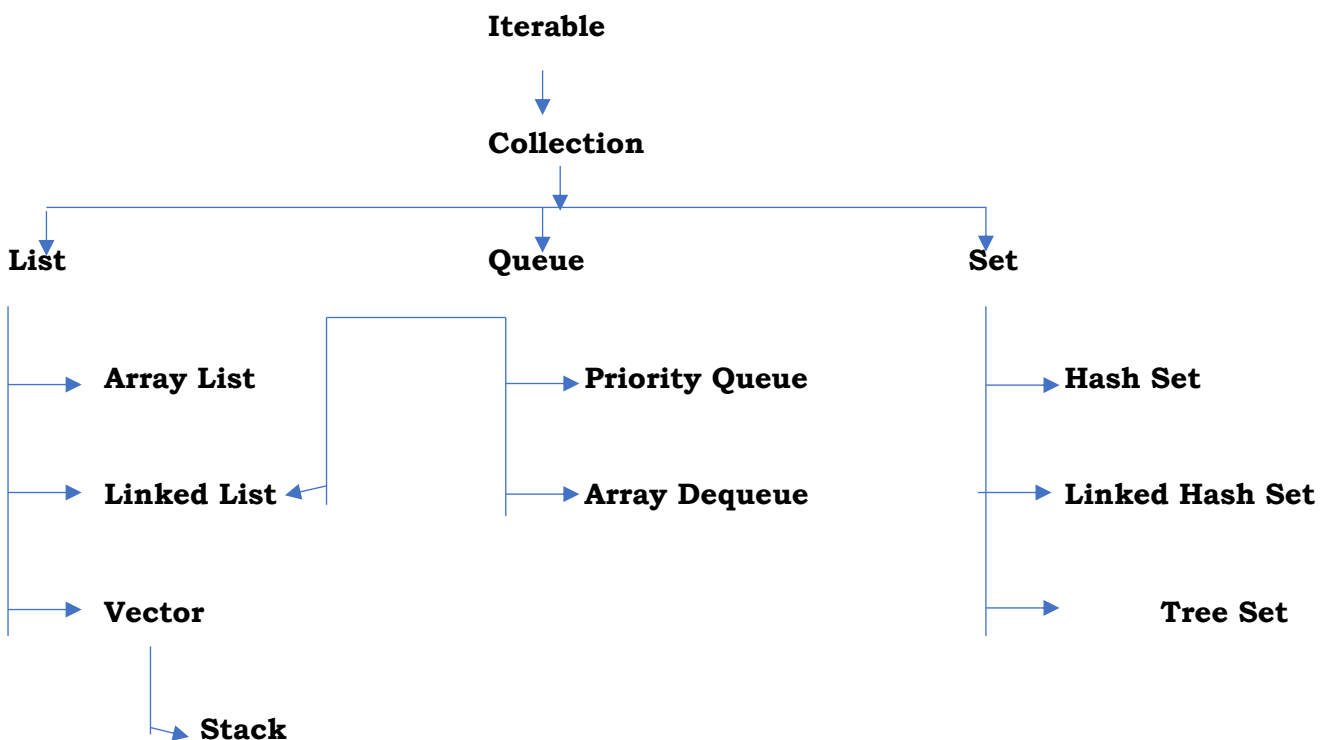
## Queue

### Java.util.Queue;

- It is an interface defined in java.util package
- It is the sub interface of java.util.Collection
- It is used to store multiple objects together.
- Queue follows **first in first out(FIFO)** or **last in last out(LILO)**
- In queue elements are added in tail side and removed or accessed from head side
- In queue except the linked list we cannot access the elements with its index.



- Implementing classes are:
  - PriorityQueue
  - LinkedList
  - ArrayDeque
- It is introduced in 1.5 version of **JDK**



## Abstract methods of queue:

	<b>Throws Exception</b>	<b>Value</b>
<b>ADD</b>	add( Object o )	Offer ( Object o )
<b>REMOVE</b>	remove()	poll ()
<b>ACCESS</b>	Element ()	peek()

Queue contains the method from java.util.Collection and also the bellowed mentioned images

### 1. Adding:

#### ➤ **Public abstract Boolean add( Object o ):**

- It is used to add an object into the queue from the tail side
- If the element is added it return true else it returns false
- If it is not possible to add the object, it throws exception

### Cases:

- **ClassCastException:** If the specified object of the class is prevented from the queue it throws ClassCastException
- **NullPointerException:** if the specifies queue is not accepting null it throws NullPointerException

#### ➤ **Public abstract boolean offer( Object o ):**

- It is used to add the object into the queue from the tail side
- If the element is added it returns true else it returns false

### 2. Remove:

#### ➤ **Public abstract object remove():**

- It is used to remove the object from the queue that is present in head
- Once after removing the object remove method returns the object
- If the queue is empty then the remove method throws NoSuchElementException.

#### ➤ **Public abstract object poll():**

- It is used to remove the object of queue that is present in head
- Once after removing the object poll method returns the object
- If the element is removed it returns the removed elemnt
- If the queue is empty then the poll method returns null

### 3. Access:

#### ➤ Public abstract object element():

- It is used to return the object at the head of the queue.
- If the queue is empty then it throws NoSuchElementException.

#### ➤ Public abstract object peek():

- It is used to return the object at the head of the queue.
- If the queue is empty then it returns null.

## Linked List

### Java.util.LinkedList

- It is an implementing class of java.util.List and java.util.Queue interface
- In linked list elements are stored in the form of nodes where each node is linked with previous and next node
- Doubly linked list is the underlying data structure of linked list
- Linked list behaves like a list as well as the queue

```
package pack.subpack;

import java.util.Queue;
import java.util.LinkedList;
public class Queue1 {
    public static void main(String[] args) {
        LinkedList<Integer> l = new LinkedList<>();
        l.offer(10);
        l.offer(20);
        l.offer(30);
        l.offer(40);
        l.offer(50);

        System.out.println(l.peek());
        l.poll();
        l.poll();

        System.out.println(l.element());
        l.remove();
        System.out.println(l.element());
    }
}
```

```
10
30
```



## ArrayDeque

### Java.util.ArrayDeque:

- It implements java.util.Deque interface which is the child of queue interface
- ArrayDeque is implemented by using resizable array
- In array deque operations can be done in both end of the queue both head and tail end of the queue.
- Adding, removing and accessing can be done in both head and tail side of the queue.
- It is introduced in 1.6 version of JDK

### Constructors:

#### ➤ **ArrayDeque() :**

It creates an empty arrayDeque object with initial capacity of 16.

#### ➤ **ArrayDeque( Collection c):**

It creates an arrayDeque object with the specified collection.

#### ➤ **ArrayDeque( int initialCapacity ):**

It creates an empty arrayDeque object with the specified initial capacity .

### Characteristics:

- Null is not accepted
- Heterogeneous objects can be added
- Insertion order is maintained
- Duplicated objects are accepted
- Elements cannot be accessed by using index.



**Methods:**

Functionality	Method Signature	Return Type
<b>ADD</b>	add( Object o )	boolean
	addFirst ( Object o )	void
	addLast( Object o )	void
	Offer( Object o )	boolean
	OfferFirst(Object o)	boolean
	offerLast( Object o)	boolean
<b>REMOVE</b>	remove()	object
	removeFirst()	object
	removeLast()	object
	poll()	object
	pollFirst()	object
	pollLast()	object
<b>ACCESS</b>	element()	object
	getFirst()	object
	getLast()	object
	peek()	object
	peekFirst()	object
	peekLast()	object

```

package pack.subpack;

import java.util.ArrayDeque;
import java.util.Iterator;
public class Queue1 {
    public static void main(String[] args) {
        ArrayDeque<String> a = new ArrayDeque<>();

        a.offer("Shhela");
        a.offerFirst("Laila");
        a.addFirst("Mala");
        a.addLast("Sindhu");

        System.out.println(a.getLast());
        System.out.println(a.getFirst());
        System.out.println(a.peekFirst());
        a.pollLast();
        System.out.println(a);
    }
}

```

```

Sindhu
Mala
Mala
[Mala, Laila, Shhela]

```

```

package samplepack;

import java.util.ArrayDeque;

public class ArrayDequeEx {
    public static void main(String[] args) {
        ArrayDeque<Integer> ad = new ArrayDeque<>();
        ad.add(10);
        ad.add(40);
        ad.offer(30);
        ad.offerFirst(45);
        ad.pollLast();

        System.out.println(ad.getLast());
        System.out.println(ad.peekLast());
        System.out.println(ad);
    }
}

```

```

40
40
[45, 10, 40]

```

## Priority Queue

### Java.util.PriorityQueue

- It is the subclass of java.util.Queue interface.
- It is implemented by using priority heap data structure.
- A priority queue in Java is an abstract data type.
- It functions similar to a regular queue or stack.
- However, it has a crucial distinction: each element has an associated priority.
- Elements with higher priorities are dequeued before those with lower priorities.
- Priority determines the order of dequeuing, not the order of insertion.
- Consequently, elements may be dequeued in a different order from their insertion order.
- It is introduced in 1.5 version of JDK.
- In priority queue all the objects that are inserted are sorted.
- Hence, in priority queue the objects that are should be java.lang.Comparable type or The object should be sorted by using java.util.Comparator.

### Key Points :

- It is implemented using a priority heap or binary heap.
- Elements are ordered based on their natural ordering (if they implement Comparable) or using a Comparator.
- The head of the priority queue is always the least (or highest, depending on ordering) element.
- Operations like insertion (enqueue) and removal of the head (dequeue) are performed efficiently in  $O(\log n)$  time complexity.

### Characteristics:

Null cannot be inserted

Insertion order is not maintained.

Duplicate objects are accepted.

Heterogeneous objects are not accepted.

Elements can not be accessed by using its index.

### Constructors:

#### 1. Default Constructor:

Constructs an empty priority queue with an initial capacity of 11 elements. If more elements are added than this initial capacity, the priority queue will automatically resize itself.

```
PriorityQueue<E> pq = new PriorityQueue<>();
```

#### 2. Constructor with Initial Capacity:

Constructs an empty priority queue with the specified initial capacity. The capacity is the number of elements the priority queue can initially store without resizing.

```
PriorityQueue<E> pq = new PriorityQueue<>(int initialCapacity);
```

### 3. Constructor with Comparator:

Constructs an empty priority queue with the specified initial capacity that orders its elements according to the specified comparator.

```
PriorityQueue<E> pq = new PriorityQueue<>(Comparator<? super E> comparator);
```

### 4. Constructor with Collection and Comparator:

Constructs a priority queue containing the elements of the specified collection, ordered according to the specified

```
PriorityQueue<E> pq = new PriorityQueue<>(Collection<? extends E> c, Comparator<? super E> comparator);
```

### 5. Constructor with Collection:

Constructs a priority queue containing the elements of the specified collection. The elements are ordered according to their natural sorting order.

```
PriorityQueue<E> pq = new PriorityQueue<>(Collection<? extends E> c);
```

Here E represents the type of elements stored in the priority queue.

### Abstract methods of priority queue:

➤ **Public abstract boolean add( object O ):**

Insert the specified element into the priority queue.

➤ **Public abstract boolean offer( Object O ) :**

Insert the specified element into the priority queue.

➤ **Public abstract boolean remove ( object O ):**

Removes a single instance of the specified element from this queue, if it is present.

➤ **Public abstract object poll ():**

Retrieves and removes the head of this queue, or returns null if this queue is empty.

➤ **Public abstract object peek():**

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

➤ **Public abstract object [ ] toArray():**

Returns an array containing all of the elements in this queue.

➤ **Public abstract boolean contains ( object o ):**

Returns true if this queue contains the specified element.

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        // Creating a PriorityQueue
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Adding elements to the PriorityQueue
        pq.add(10);
        pq.add(20);
        pq.add(15);

        // Printing the elements of the PriorityQueue
        System.out.println("Elements of PriorityQueue: " + pq);

        // Removing elements from the PriorityQueue
        int removedElement = pq.poll();
        System.out.println("Removed element: " + removedElement);

        // Printing the elements after removal
        System.out.println("Elements of PriorityQueue after removal: " + pq);

        // Peeking the head of the PriorityQueue
        int head = pq.peek();
        System.out.println("Head of PriorityQueue: " + head);
    }
}
```

Elements of PriorityQueue: [10, 20, 15]

Removed element: 10

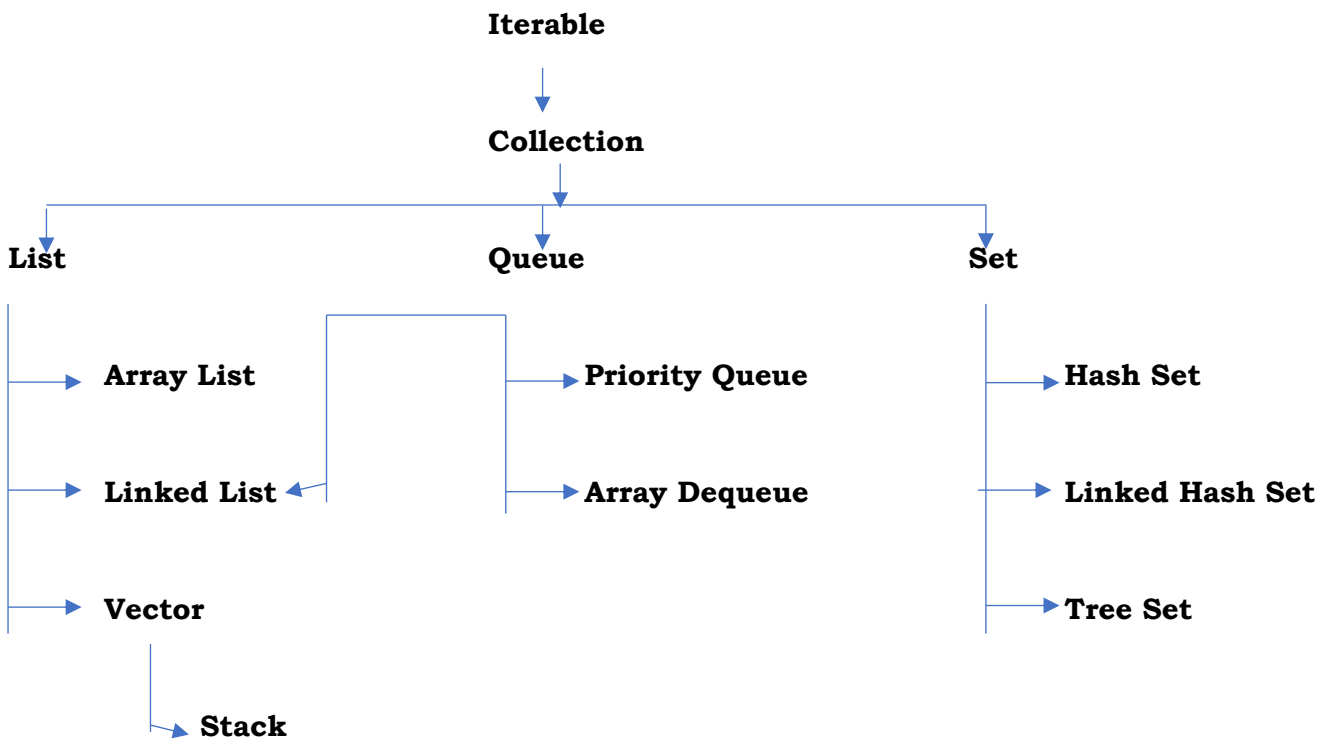
Elements of PriorityQueue after removal: [15, 20]

Head of PriorityQueue: 15

## Set

### Java.util.set

- It is an interface defined in java.util package
- It is the sub-interface of java.util.collection
- It is used to store multiple unique objects together
- Set doesn't allow duplicate objects and it is not index based
- HashSet, LinkedHashSet and TreeSet are the implementing classes of set interface



### Java.util.HashSet

- It is the implementing class of java.util.Set interface.
- It is implementing by using hash Table data structure.
- It follows hashing mechanism to store/add the objects.
- It is introduced in 1.2 version of **JDK**.

### Characteristics

#### NIDHI

- Only one null can be inserted.
- Insertion order is not maintained.
- Duplicate objects are not accepted.
- Heterogeneous objects are allowed.
- Elements cannot be accessed by using its index, hence it not index based.

## Removing duplicate objects:

- HashSet removes the duplicate object with the help of hashCode and equals method.
- If the hashCode and equals method are not overridden then the duplicate objects are removed based on the address.
- If hashCode and equals method are overridden then the duplicate objects are removed based on the states of the objects

## constructors

### ➤ **HashSet()**

It creates an empty hashset object with the default initial capacity of 16 and load factor of 0.75

### ➤ **HashSet( int initialCapacity )**

It creates an empty HashSet object with specified initial capacity with load factor 0.75

### ➤ **HashSet( int initialCapacity , float loadFactor )**

It creates an empty HashSet object with specified initial capacity and load factor

### ➤ **HashSet( Collection c )**

It creates a HashSet Object with the specified collection

```
package pack.subpack;

import java.util.HashSet;
import java.util.Iterator;

public class HashSet1 {
    public static void main(String[] args) {
        HashSet hs = new HashSet<>();
        hs.add(20);
        hs.add("Sheela");
        hs.add(77.28);
        hs.add(99.22);
        hs.add(null);
        hs.add("Laila");
        hs.add("Laila");

        Iterator i = hs.iterator();

        while(i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Sheela  
99.22  
20  
77.28  
Laila

```
package pack.subpack;

import java.util.HashSet;
import java.util.Objects;

public class Students {
    String name;
    int id;

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

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, id);
    }

    @Override
    public boolean equals(Object o) {
        Students s = (Students) o;
        if (name.equals(s.name) && id == s.id)
            return true;
        else
            return false;
    }

    public static void main(String[] args) {
        HashSet<Students> hs = new HashSet();
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

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

[Sindhu, Laila, Sheela, Shakela, Indhu]



## Linked Hash set

### Java.util.LinkedHashSet

- It is the subclass of java.util.HashSet.
- It is implemented by using hash table and doubly linked list data structure.
- In LinkedHashSet insertion order is maintained
- It is introduced in 1.4 version of JDK

### Characteristics:

- Only one null can be inserted
- Insertion order is maintained
- Duplicate objects are not accepted
- Heterogeneous objects are allowed
- Elements cannot be accessed by using its index

### Constructor

#### ➤ **LinkedHashSet()**

It creates an empty LinkedHashSet object with the default initial capacity of 16 and load factor of **0.75**

#### ➤ **LinkedHashSet( int initialCapacity )**

It creates an empty LinkedHashSet object with specified initial capacity with load factor 0.75

#### ➤ **LinkedHashSet( int initialCapacity , float loadFactor )**

It creates an empty LinkedHashSet object with specified initial capacity and load factor

#### ➤ **LinkedHashSet( Collection c)**

It creates a LinkedHashSet Object with the specified collection

### Removing Duplicate objects

**LinkedHashSet** removes duplicate objects with the help of hashCode and equals method

```
package pack.subpack;

import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;

public class LinkedHashSet1 {
    public static void main(String[] args) {
        LinkedHashSet hs = new LinkedHashSet<>();
        hs.add(20);
        hs.add("Sheela");
        hs.add(77.28);
        hs.add(99.22);
        hs.add(null);
        hs.add("Laila");
        hs.add("Laila");

        Iterator i = hs.iterator();

        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

```
20
Sheela
77.28
99.22
null
Laila
```

```

package pack.subpack;

import java.util.LinkedHashSet;
import java.util.Objects;

public class Students {
    String name;
    int id;

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

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, id);
    }

    @Override
    public boolean equals(Object o) {
        Students s = (Students) o;
        if (name.equals(s.name) && id == s.id)
            return true;
        else
            return false;
    }

    public static void main(String[] args) {
        LinkedHashSet<Students> hs = new LinkedHashSet();
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

        System.out.println(hs);
    }
}

```

```
[Sheela, Laila, Shakela, Sindhu, Indhu]
```

## TreeSet

- It is implementing class of java.util.Set interface
- In tree set by default all the objects are sorted based in natural sorting order
- Hence, the object that are added into the tree set should be java.lang.comparable.
- It is introduced in 1.2 version of **JDK**.
- It is implemented by using navigable set and TreeMap

### Characteristics:

- Null cannot be inserted
- If we try to add the null we will get null pointer Exception.
- Insertion order is not maintained
- Duplicate objects are not allowed
- Heterogeneous objects are nor accepted, if we try to add heterogeneous objects we will get class cast exception.
- Elements cannot be accessed by using its index

### Note:

- **If the objects that are added to the tree set is not comparable type we get Class Cast Exception**
- **To add the object to the tree set which is not comparable we can use comparator**

### Constructors:

#### ➤ **TreeSet():**

It creates an empty treeset objects that is comparable and it follows the natural ordering of an element for sorting

#### ➤ **TreeSet(Collection c):**

It creates a new treeset objects that contains the objects in the specified collection which is sorted based on natural ordering that is comparable.

#### ➤ **TreeSet(Comparator ):**

It Creates an empty TreeSet object that sort all the objects based on specified comparator.

**Note:** when we use this constructor to add the object all the objects are sorted based on the comparator not based on the comparable

### Example:

```
package samplepack;

import java.util.Iterator;
import java.util.TreeSet;

public class Main {

    public static void main(String[] args) {
        TreeSet ts = new TreeSet<>();
        ts.add(178);
        ts.add(174);
        ts.add(1774);
        ts.add(12);
        ts.add(179);

        Iterator i = ts.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

```
12
174
178
179
1774
```

### Descending order:

```
package samplepack;

import java.util.Comparator;
import java.util.Iterator;
import java.util.TreeSet;

class SortByDecs implements Comparator<Integer>{

    @Override
    public int compare(Integer o1, Integer o2) {
        return -(o1-o2);
    }

}

public class Main {
```

```
public static void main(String[] args) {  
    TreeSet<Integer> ts = new TreeSet<>(new SortByDecs());  
    ts.add(178);  
    ts.add(174);  
    ts.add(1774);  
    ts.add(12);  
    ts.add(179);  
  
    Iterator i = ts.iterator();  
    while (i.hasNext()) {  
        System.out.println(i.next());  
    }  
}
```

```
1774  
179  
178  
174  
12
```

### Example:

```
package pack.subpack;

import java.util.TreeSet;

public class Students implements Comparable<Students> {
    String name;
    int id;

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

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int compareTo(Students s) {
        return id - s.id;
    }

    public static void main(String[] args) {
        TreeSet hs = new TreeSet();
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

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

```
[Sheela, Indhu, Laila, Shakela, Sindhu]
```

### Sort By id in descending order:

```
package pack.subpack.set;

import java.util.Comparator;

public class SortByIdDesc implements Comparator<Students> {

    public int compare(Students o1, Students o2) {

        return -(o1.id - o2.id);
    }
}
```

```
}
```

```
package pack.subpack.set;

import java.util.TreeSet;

public class Students {
    String name;
    int id;

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

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        TreeSet hs = new TreeSet(new SortByIdDesc());
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

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

```
[Sindhu, Shakela, Laila, Indhu, Sheela]
```



## MAP

### Java.util.Map

- It is an interface defined in java.util package
- It is used to store multiple objects together along with that for every object one unique identification is provided known as key
- Every key is associated with the values

E.g.:

District	–	pin code
Country	–	Country code
Employee	-	ID
Student	–	RollNo
Account	–	Ac/No.

- In map objects are stored in the form of key and value pairs.
- Key should be unique but value can be repeated
- One key and Value pair is known as entry

Key	value
-----	-------

Entry

- Hence, in a map multiple entries can be added

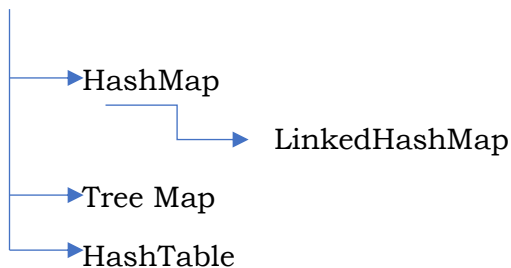
### HashMap

### LinkedHashMap

### TreeMap

### HashTable -- assignment

Map



## Abstract methods

These are the implementing classes of Map interface

Functionality	Method Name	Return Type
Adding Entries	put(Object key, Object Value)	Object ( value )
	putAll( Map m)	void
Removing Entries	remove ( Object Key)	Object ( value )
	clear()	void
Search	containsKey( Object Key)	boolean
	containsValue ( Object Value )	boolean
Replace	replace( Object Key, Object Value)	Object ( value )
	replace( Object Key, Object OldValue, Object NewValue )	boolean
Access	get( Object Key )	Object ( value )
	values()	Collection
	keySet()	Set
	entrySet()	Set

## Java.util.HashMap

- It is implementing class of **Java.util.Map interface**.
- It is implemented by using HashTable
- It is introduced in **1.2 version of JDK**
- It is used to store the objects in the form of key and value pairs

### Characteristics:

- Only one null can be used as a key
- Insertion order of entry is not maintained
- Duplicate keys not allowed , If we try to use duplicate keys then the existing values is replaced with new value
- Heterogeneous keys can be inserted
- It is not index based

## Constructors:

➤ **HashMap():**

Initial capacity is 16.

Load factor is 0.75.

➤ **HashMap(int initialCapacity):**

Default load factor is 0.75

➤ **HashMap(int initialCapacity, float loadfactor):**

➤ **HashMap( Map m):**

## Example:

```
package pack.subpack.set;

import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap hs = new HashMap();

        hs.put(5, "Sheela");
        hs.put(6, "Laila");
        hs.put(true, "Job");
        hs.put(7.5, 78);
        hs.put('i', 79);
        hs.put("One", 1);
        hs.put(null, "nothing");
        System.out.println(hs);
        hs.put(5, "Sindhu");//It replaces Sheela with Sindhu

        System.out.println(hs);

        System.out.println(hs.values());
        System.out.println(hs.keySet());
        System.out.println(hs.entrySet());
    }
}
```

```
{null=nothing, 5=Sheela, 6=Laila, One=1, i=79, 7.5=78, true=Job}
{null=nothing, 5=Sindhu, 6=Laila, One=1, i=79, 7.5=78, true=Job}
[nothing, Sindhu, Laila, 1, 79, 78, Job]
[null, 5, 6, One, i, 7.5, true]
[null=nothing, 5=Sindhu, 6=Laila, One=1, i=79, 7.5=78, true=Job]
```

Use Iterator in map:

```
package samplepack;

import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;

public class Main {

    public static void main(String[] args) {
        LinkedHashMap<Integer, String> hs = new LinkedHashMap<>();
        hs.put(5, "Sheela");
        hs.put(7, "Laila");
        hs.put(8, "Chetan");
        hs.put(9, "Vikas");
        hs.put(10, "Saurabh");
        hs.put(1, "Shakeela");

        Set<Integer> s = hs.keySet();

        Iterator i = s.iterator();
        while (i.hasNext()) {
            System.out.println(hs.get(i.next()));
        }
        System.out.println("-----");

        for (Integer integer : s) {
            System.out.println(hs.get(integer));
        }
    }
}
```

```
Sheela
Laila
Chetan
Vikas
Saurabh
Shakeela
```

-----

```
Sheela
Laila
Chetan
Vikas
Saurabh
Shakeela
```