J**ava** - Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

1. java was developed by Sun Microsystems) in the year 1995.
2. James Gosling is known as the father of Java.
3. Before Java, its name was Oak
4.. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

example

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

Java Platforms / Editions

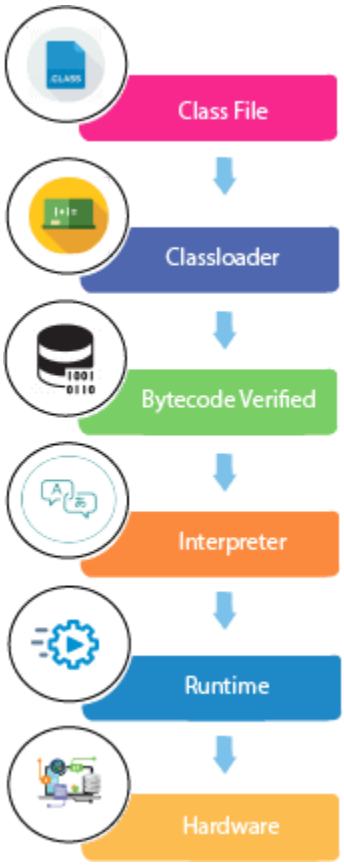
1) Java SE (Java Standard Edition) 2) Java EE (Java Enterprise Edition)
3) Java ME (Java Micro Edition) (mobile application)
4) JavaFX (It is used to develop rich internet applications)

Why Java was named as "Oak"? -> it is symbol of strength. it is national tree of many countries like(usa etc).

Features:-

1. **Simple**
2. **Object-Oriented**
3. **Platform-Independent**
4. **Secure : -** byte code verifier helps to prevent malicious code.
5. **Robust** :- It handle memory management.
6. **Multithreaded**
7. **High Performance**
8. **Dynamic** :- Java applications can load classes and execute code dynamically.
9. **Portable** :- compile one system and run anywhere.
10. **Architecture-Neutral**

STEPS OF JAVA EXECUTION



C++ vs Java

There are many differences and similarities between the C++ programming language and Java. A list of top differences between C++ and Java are given below:

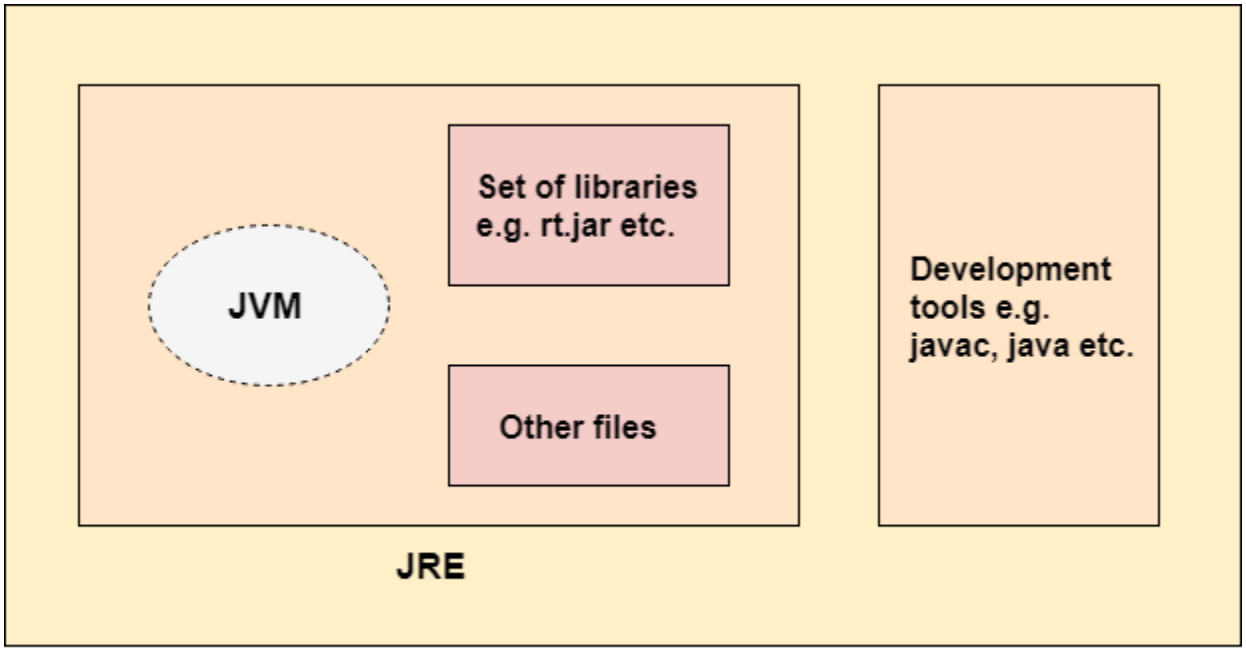
Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.

Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.

Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.

Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

JDK,JRE,JVM



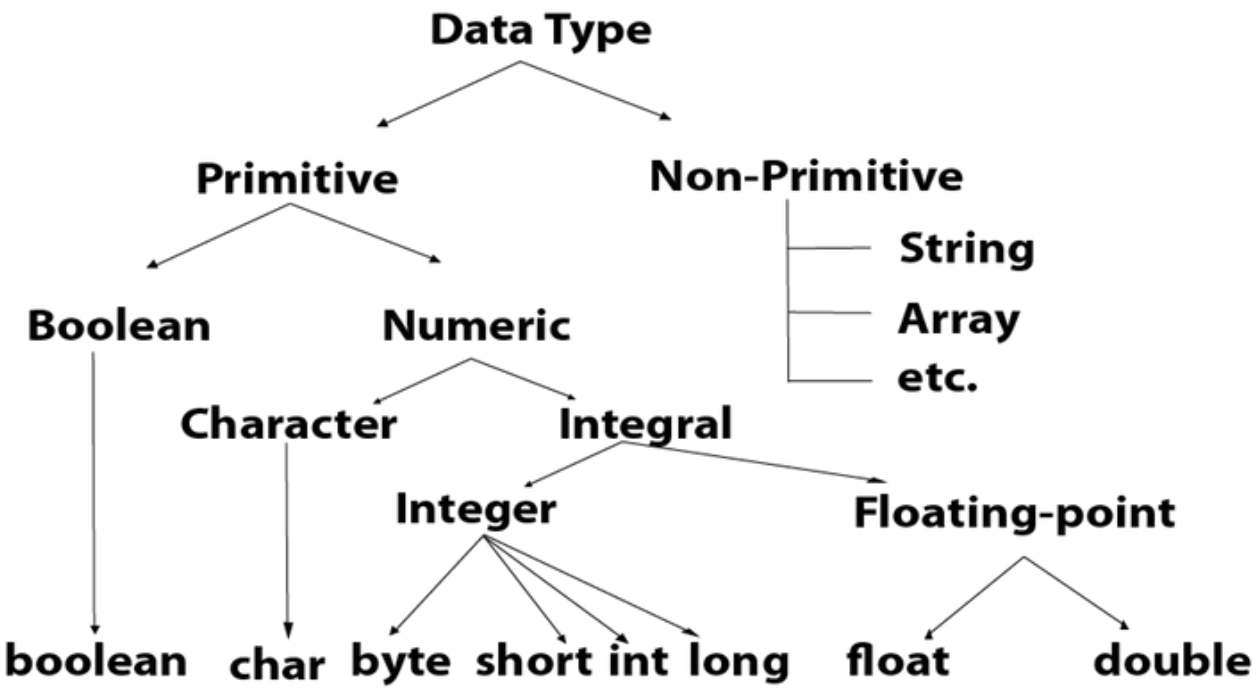
JVM :- Loads code

- **Verifies code**
- **Executes code**
- **Provides runtime environment**

Method area = main method , static variables.
Heap = instance variables , objects(new)
Stack area = local methods , local variables

Data Types in Java

1. **Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.**
2. **Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.**



Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :

Java Left Shift Operator Example

```
1. public class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10<<2);//10*2^2=10*4=40
4. System.out.println(10<<3);//10*2^3=10*8=80
5. System.out.println(20<<2);//20*2^2=20*4=80
6. System.out.println(15<<4);//15*2^4=15*16=240
7. }}
```

Java Right Shift Operator Example

```
1. public OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}
7. public class OperatorExample {

  public static void main(String args[]) {
    int a = 10;
    int b = 5;
    int c = 20;

    System.out.println(a < b && a++ < c); // false && true = false
    System.out.println(a); // 10 because second condition is not checked

    System.out.println(a < b & a++ < c); // false & true = false
    System.out.println(a); // 11 because second condition is checked
  }
}
```

```
false
10
```

false
11

Summary

- **Short-Circuiting (`&&`): If the first condition of `&&` is `false`, the second condition is not evaluated. In this case, `a` remains unchanged.**
- **Non-Short-Circuiting (`&`): Both conditions are evaluated, so `a` is incremented.**

|| *AND* |

```
public class OperatorExample {
   public static void main(String args[]) {
      int a = 10;
      int b = 5;
      int c = 20;

      System.out.println(a > b || a < c); // true || true = true
      System.out.println(a > b | a < c);  // true | true = true
      // || vs |
      System.out.println(a > b || a++ < c); // true || true = true
      System.out.println(a); // 10 because second condition is not checked
      System.out.println(a > b | a++ < c);  // true | true = true
      System.out.println(a); // 11 because second condition is checked
   }
}
```

Summary

- **Logical OR (`||`): Uses short-circuit evaluation, meaning if the first condition is `true`, the second condition is not evaluated.**
- **Bitwise OR (`|`): Does not use short-circuit evaluation, meaning both conditions are always evaluated.**
- **Outputs:**
    - **`true` (from the `||` operator, first condition `true`)**
    - **`true` (from the `|` operator, both conditions `true`)**

Inheritance:-

single

```
class Hello {
   public  void m1() {
      System.out.println("run");
   }

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

class Main extends Hello {
   public void run() {
      System.out.println("running");
   }

   public static void main(String[] args) {
      Main c = new Main();
      c.run();
      Hello h = new Hello();
      h.m1();
      h.display();
}
}
```

Output: -

Running
Run
Display

Multilevel :-

```
class Animal {
   public  void m1() {
      System.out.println("run");
   }

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

class dog extends Animal{
   public void bark(){
      System.out.println("barking");

   }
   public void sleep(){
      System.out.println("sleep");
   }
}

class Main extends dog{
   public void eat(){
      System.out.println("eating");
   }
   public static void main(String[] args){
    Main m = new Main();
    m .eat();       //eating
```

```
     m.bark();      // barking
     m.m1();      // run

     dog d = new dog();
     d.bark();
     d.display();
    }
}
```

Hierarchical Inheritance

```
class Animal {
    public  void m1() {
        System.out.println("run");
    }

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

class dog extends Animal{
    public void bark(){
        System.out.println("barking");

    }
    public void sleep(){
        System.out.println("sleep");
    }
```

```
}

class Main extends Animal{
    public void eat(){
        System.out.println("eating");
    }
    public static void main(String[] args){
     Main m = new Main();

     m.m1();      // run

     dog d = new dog();
     d.bark();
     d.display();
    }
}
```

POLYMORPHISM:-

1.Method overloading

```
class Cal {
    public static int add(int a, int b) {
        return a + b;
    }

    public static int sub(int a, int b) {
        return a - b;
    }
```

```
    public static int mul(int a, int b) {
        return a * b;
    }
}


class Main extends Cal {
    public static int div(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return a / b;
    }

    public static void main(String[] args) {

        System.out.println("Addition: " + Cal.add(10, 5));
        System.out.println("Subtraction: " + Cal.sub(10, 5));
        System.out.println("Multiplication: " + Cal.mul(10, 5));


        System.out.println("Division: " + Main.div(10, 5));
    }
}
```

METHOD OVERRIDING:- (

```
class A{
    public int m1(int a , int b){
```

```
        return a+b;
    }

}

class Main extends A{
    public  int m1(int a , int b){
        return a + b;
    }
    public static void main(String[] args){
        A a = new A();
        System.out.println(a.m1(10,5));  // 15
        Main b = new Main();
        System.out.println(b.m1(11,5)); //16
    }
}
```

SUPER();

1. Non static method and super instance method call only
2. Static method of super don't call.
3. If you want to call static method in child class without using super () method so you need to provide like that

Ex . A.M1();   // A is a parent class and m1 is static method of parent class

```
class A {
   public static void work() {
      System.out.println("working");
   }
}

class Main extends A {
       void eat() {
       System.out.println("eating");
   }
```

```
    public static void dance() {
        A.work(); // Correct way to call static method from the superclass
        System.out.println("dancing");
    }

    public static void main(String[] args) {
        Main b = new Main();
        Main.dance();
        b.eat();     }
}
```

With use super()

```
class A{
    public void  work(){
        System.out.println("working");
    }

}

class Main extends A{
     void eat(){
         System.out.println("eating");
     }

     void dance(){
         super.work();
         System.out.println("dancing");
     }
    public static void main(String[] args){

        Main b = new Main();
        b.dance();
        b.eat();
    }
}
```

Java Runtime Polymorphism with Data Member

```
1. class Bike{
2.  int speedlimit=90;
3. }
4. class Honda3 extends Bike{
5.  int speedlimit=150;
6.
7.  public static void main(String args[]){
8.   Bike obj=new Honda3();
9.   System.out.println(obj.speedlimit);//90
10.}
```

> Rule: Runtime polymorphism can't be achieved by data members.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
1. class Dog{
2.  private void eat(){System.out.println("dog is eating...");}
3.
4.  public static void main(String args[]){
5.   Dog d1=new Dog();
6.   d1.eat();  }}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
1. class Animal{
2.  void eat(){System.out.println("animal is eating...");}
3. }
4.
5. class Dog extends Animal{
6.  void eat(){System.out.println("dog is eating...");}
7.
8.  public static void main(String args[]){
9.   Animal a=new Dog();
10. a.eat();
11. }
12.}
```

Abstraction in Java

1.Hide important details and showing only essential info to user or (a method without body).

2.It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

3.An abstract class must be declared with an abstract keyword.

- **It can have abstract and non-abstract methods.**

- **It cannot be instantiated.**
- **It can have constructors and static methods also.**
- **It can have final methods which will force the subclass not to change the body of the method.**
- **Static , final, void method allow in abstract class**
- **Static method call by using classname.methodname();**
- **Other method call by using child class obj,**
-

```
abstract class A{
    abstract void run();  // abstract word mandatory
    void sleep(){  // default void method

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

    public static void write(){
        System.out.println("writting");
    }

   final void read() {
        System.out.println("reading.");
    }

}
```

```
class Main extends A{
    void run(){
        System.out.println("runnnig");
    }

    public void eat(){
        System.out.println("eting");
    }

    public static void main(String[] args){
        Main m = new Main();
        m.run();
        m.eat();
        m.read();
        m.eat();

        //static
        A.write();
    }
}
```

By using abstract class obj

```
abstract class A {
    // Abstract method
    abstract void run();

    // Concrete method
    void sleep() {
        System.out.println("sleeping");
    }

    // Static method
    public static void write() {
        System.out.println("writing");
    }

    // Final method
    final void read() {
        System.out.println("reading.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an anonymous inner class to provide
implementation for the abstract method
        A a = new A() {
            @Override
            void run() {
                System.out.println("running");
            }
        };
```

```
        // Calling instance methods
        a.run();    // Outputs: running
        a.sleep();  // Outputs: sleeping
        a.read();   // Outputs: reading

        // Calling static method
        A.write();  // Outputs: writing
    }
}
```

Interface in Java

An interface in Java is a blueprint of a class. It has static constants and abstract methods, default void.

default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

- **By interface, we can support the functionality of multiple inheritance.**
- **It can be used to achieve loose coupling.**

Example:-

```
interface A {
   default void run(){
      System.out.println("runnig");
   }

    void bark(); // abstract          or abstract void bark();

    static void sleep(){
       System.out.println("sleeping");
    }
}
```

```
class Main implements A{
   public void bark(){                //access specifier takne
      System.out.println("barking");
   }

   public void eat(){
      System.out.println("eating");
   }

   public static void main(String[] args){
      Main m = new Main();
      m.run();
      m.bark();
      A.sleep();

      m.eat();
   }
}
```

Java Encapsulation

Wrapping a data and methods(code) in a single unit.
Not outside world.
It’s resolve implementation level issues
Same as abstraction
Private variables.
Getter and setter method only public (**read-only or write-only**).
Only modifies and access through public getter and setter method.

```
class A {
    private String name;
    private int age;


    public A(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }


    public String getName() {
        return name;
    }


    public void setAge(int age) {
        this.age = age;
    }


    public int getAge() {
        return age;
    }


    public void display() {
        System.out.println("Name is " + name + " and age is " + age);
    }
}

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

        A a = new A("Loki", 21);


        a.display();


        a.setName("Thor");
        a.setAge(25);


        a.display();
```

```
        System.out.println("Updated name: " + a.getName());
        System.out.println("Updated age: " + a.getAge());
    }
}
```

Output

```
Name is Loki and age is 21
Name is Thor and age is 25
Updated name: Thor
Updated age: 25
```

Access Modifiers in Java

1. **Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.**
2. **Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.**
3. **Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.**
4. **Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.**

5. Understanding Java Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Wrapper classes in Java

***convert primitive into object and object into primitive*.**

java is not considered a 100% pure object-oriented programming (OOP) language.

because it includes primitive data types, which are not objects.

Java support primitive data types not object like c++.

Wrapper class in java : - `java.lang` package

```
import java.lang.*;

class Main{

    public static void main(String[] args){

        int a = 10;

        Integer j = a;    // autoboxing

        //or Integer j = Integer.valueof(a);

        System.out.println(j);



        Integer b = new Integer(25);

        int x = b;          // unboxing

        // or int x = b.intValue();
```

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

JDBC

1 . java database connectivity.

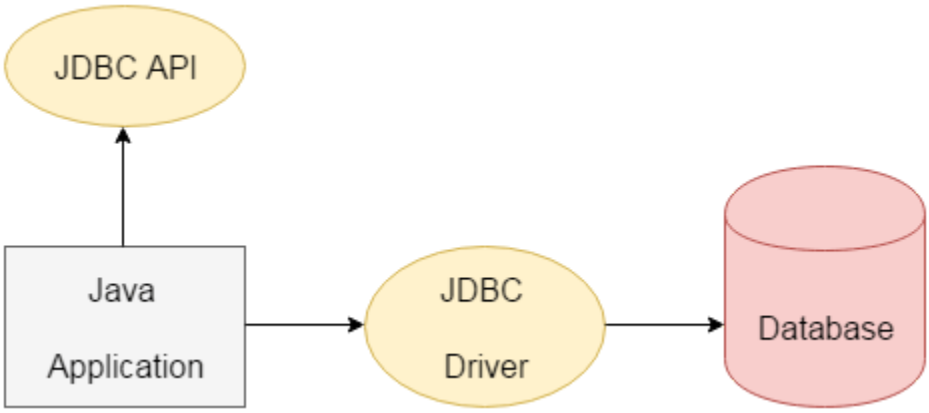
2. It uses jdbc driver for connect to database.

3. JDBC is a Java API to connect and execute the query with the database.

4. Java api is used to access data , save ,modify ,delete and update

5. 4 Driver

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver



- **Driver interface**
- **Connection interface**
- **Statement interface**
- **PreparedStatement interface**
- **CallableStatement interface**
- **ResultSet interface**
- **ResultSetMetaData interface**
- **DatabaseMetaData interface**
- **RowSet interface**

A list of popular *classes* of JDBC API are given below:

- **DriverManager class**
- **Blob class**
- **Clob class**
- **Types class**

Connectivity with mysql

```
1. import java.sql.*;
2. class MysqlCon{
3. public static void main(String args[]){
4. try{
5. Class.forName("com.mysql.jdbc.Driver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:mysql://localhost:3306/sonoo","root","root");
8. Statement stmt=con.createStatement();
```

```
9.  ResultSet rs=stmt.executeQuery("select * from emp");
10. while(rs.next())
11. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
12. con.close();
13. }catch(Exception e){ System.out.println(e);}
14. }
15. }
```

Use preparedstatement

```
import java.sql.*;



public class Main {

   public static void main(String[] args) {



      String url = "jdbc:mysql://localhost:3306/yourdatabase";

      String user = "username";

      String password = "password";



      Connection con = null;
```

```
PreparedStatement pstmt = null;



try {


    Class.forName("com.mysql.cj.jdbc.Driver");




    con = DriverManager.getConnection(url, user, password);

    System.out.println("Connected to the database");


    // 1. Using PreparedStatement for INSERT


    String insertSQL = "INSERT INTO yourtable (col1, col2) VALUES (?, ?)";

    pstmt = con.prepareStatement(insertSQL);

    pstmt.setString(1, "value1"); // Set parameter 1
```

```
        pstmt.setInt(2, 123); // Set parameter 2

        int rowsInserted = pstmt.executeUpdate();

        System.out.println("Insert Success, Rows affected: " + rowsInserted);



        // 2. Using PreparedStatement for UPDATE



        String updateSQL = "UPDATE yourtable SET col_name = ? WHERE
condition_column = ?";

        pstmt = con.prepareStatement(updateSQL);

        pstmt.setString(1, "newValue"); // Set parameter 1

        pstmt.setInt(2, 123); // Set parameter 2

        int rowsUpdated = pstmt.executeUpdate();

        System.out.println("Update Success, Rows affected: " + rowsUpdated);



        // 3. Using PreparedStatement for SELECT
```

```
        String selectSQL = "SELECT col1, col2 FROM yourtable WHERE
condition_column = ?";

        pstmt = con.prepareStatement(selectSQL);

        pstmt.setInt(1, 123); // Set parameter

        ResultSet rs = pstmt.executeQuery();



        // Process the ResultSet



        while (rs.next()) {



           String col1 = rs.getString("col1");

           int col2 = rs.getInt("col2");

           System.out.println("Selected Data: col1 = " + col1 + ", col2 = " + col2);

        }



    } catch (ClassNotFoundException e) {
```

```
            System.err.println("JDBC Driver not found.");
            e.printStackTrace();
        } catch (SQLException e) {
            System.err.println("SQL Exception.");
            e.printStackTrace();
        } finally {


            try {
                if (pstmt != null) pstmt.close();
                if (con != null) con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }}
```

HIBERNATE

1.Hibernate is an ORM(OBJECT RELATIONAL MAPPING) framework it is use to convert java object into database table(relational database)

`@GeneratedValue(strategy = GenerationType.IDENTITY)` annotation in JPA .When you define an entity class in JPA, the primary key field usually needs to be unique for each instance. To handle this, you can configure the primary key to be auto-generated, meaning that the database will automatically generate a unique value for the primary key when a new record is inserted.

```
import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;



@Entity

public class Student {

   @Id

   @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
private Long id;

private String name;

private int age;



// Getters and setters

public Long getId() {

    return id;

}



public void setId(Long id) {

    this.id = id;

}



public String getName() {

    return name;
```

```
    }


    public void setName(String name) {
        this.name = name;
    }


    public int getAge() {
        return age;
    }


    public void setAge(int age) {
        this.age = age;
    }
}
```

Collection framework

1. ArrayList

```
        import java.util.List;
import java.util.ArrayList;



class Main{
   public static void main(String[] args){
      List<Integer>list = new ArrayList<>();


      list.add(1);
      list.add(2);
      list.add(3);
      list.add(1);


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

```
   list.add(3,4);

System.out.println(list);



list.remove(2);




System.out.println(list.get(0));



System.out.println(list.contains(1));



System.out.println(list.size());



for (int i = 0 ;i<list.size() ;i++ ){

    System.out.println(list.get(i));}}}
```

Java LinkedList class

Internally doubly linked list use.

It inheritance abstract class and Deque interface.

```
import java.util.LinkedList;

import java.util.Deque;



public class Main {

    public static void main(String[] args) {



        LinkedList<String> linkedList = new LinkedList<>();




        linkedList.add("Alice");

        linkedList.add("Bob");
```

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




Deque<String> deque = new LinkedList<>();



deque.addFirst("John");

deque.addLast("Jane");

System.out.println( deque);






String popped = deque.pop();

System.out.println(popped);

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

```
deque.add("Mike");

System.out.println(deque);




System.out.println( deque.peek());




System.out.println("First element: " + deque.getFirst());

System.out.println("Last element: " + deque.getLast());




deque.removeFirst();

deque.removeLast();

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

```
        System.out.print("Iterating over deque: ");
        for (String element : deque) {
            System.out.print(element + " ");
        }
        System.out.println();


        deque.clear();
        System.out.println("Deque after clearing: " + deque);
    }
}
```

ArrayList and LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

STACK

(push,pop,peek,remove(data),add,clear,contains,size ,addall ,removeall,etc)

```
import java.util.Stack;

import java.util.List;



class Main {

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

```
Stack<Integer> stack = new Stack<>();




stack.push(1);

stack.push(2);

stack.push(3);




System.out.println(stack);

System.out.println(stack.pop());

System.out.println(stack.peek());

System.out.println(stack.contains(1));

stack.add(6); // this is number

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

```
        stack.remove(2); // it takes index

        System.out.println(stack);

        stack.add(1,15);

        System.out.println(stack);

        stack.clear();



        System.out.println(stack);

    }

}
```

Vector (add,addElement ,
CONTAINS,elementAt(index),removeElement(index))etc

```
import java.util.Vector;

public class VectorExample {

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

```
        Vector<Integer> vector = new Vector<>();

        vector.addElement(10);

        vector.addElement(20);

        vector.addElement(30);

        System.out.println("Vector: " + vector);

       System.out.println("Element at index 1: " + vector.elementAt(1));

       vector.removeElement(20);

        System.out.println("Vector after removing 20: " + vector);

        vector.insertElementAt(25, 1);

        System.out.println("Vector after inserting 25 at index 1: " + vector);

        System.out.println("Does the vector contain 30? " + vector.contains(30));

    }

}
```

Set (not support indexing)

HashSet

Methods = add,addAll , remove(data) , removeAll , contains,isEmpty, size, clear , iterator

set1.addAll(set2); set1.removeAll(set2);

LinkedHashSet

Java LinkedHashSet class contains unique elements only like HashSet.

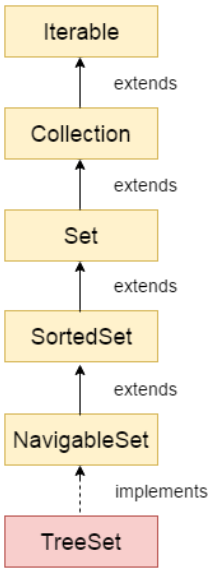
Java LinkedHashSet class provides all optional set operations and permits null elements.

Java LinkedHashSet class is non-synchronized.

Java LinkedHashSet class maintains insertion order.

```
1. Iterator<String> itr=al.iterator();
2. while(itr.hasNext()){
3.  System.out.println(itr.next());
4. }
```

Treeset



1. **Not allow dupli , null , asc order , fast access.**
2. **Iterator i=set.descendingIterator(); => descending val**
3. **pollFirst() , pollLast();**

Queue Interface

1.fifo

2.element add from rare side and remove from front side.

2.offer , poll , add ,remove (head of ele remove and print) , peek , element(1st ele in queue),size,isEmpty,contains,clear etc

(priority queue , Linked list) same method

(Deque , ArrayDeque) => add,addFirst,addLast ,offer , OfferFirst , offerLast

2. Remove Methods => **Remove , removeFirst,removeLast , poll ,pollFirst,pollLast**

3.*peek() = >* peek , peekFirst,peekLast , getFirst , getLast , element

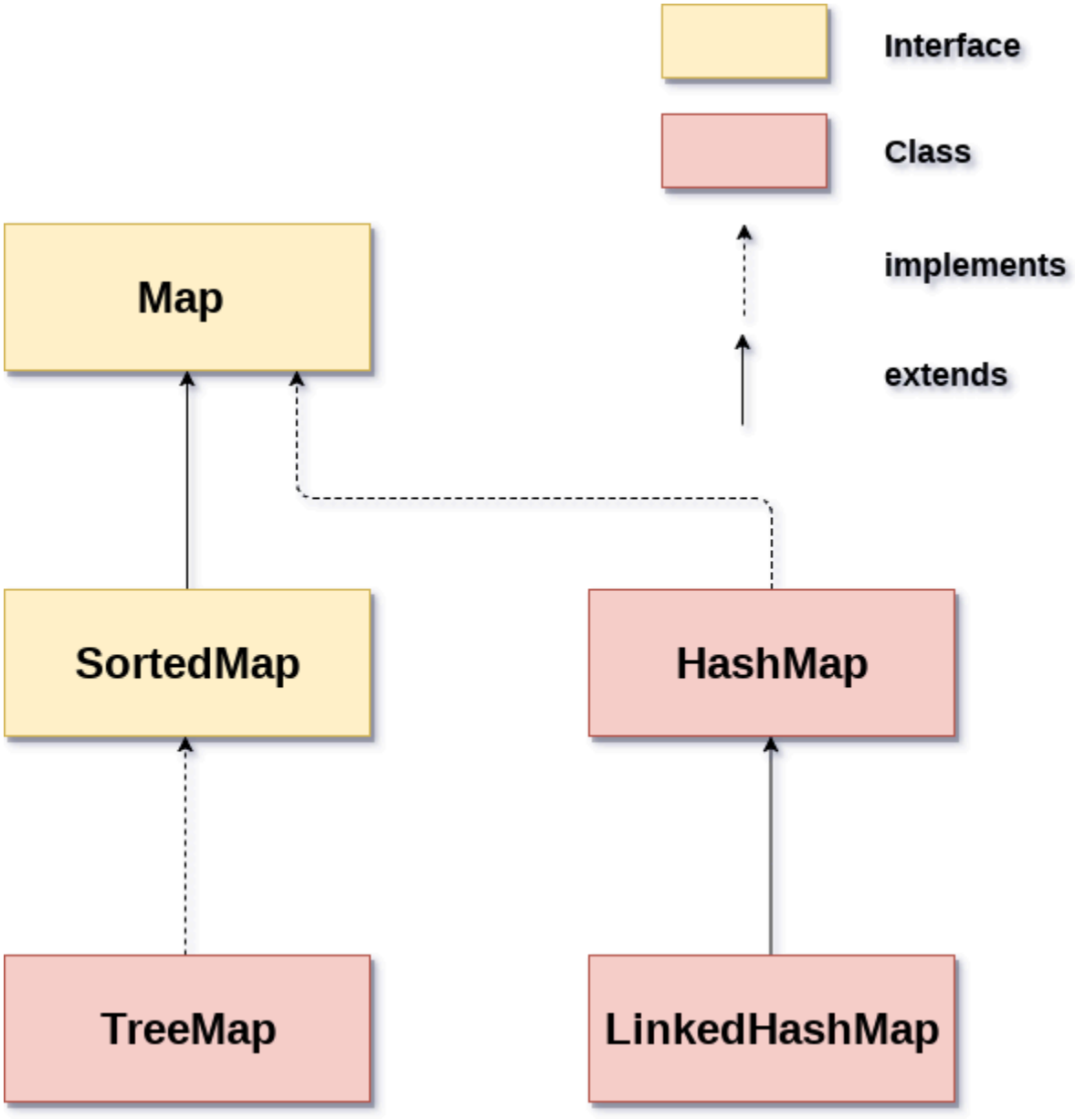
4.clear

You need to use a concrete implementation of the `Queue` interface, such as `LinkedList`, `ArrayDeque`, or `PriorityQueue`.

```
import java.util.Deque;
import java.util.ArrayDeque;
class Main {
   public static void main(String[] args) {
      Deque<Integer> dq = new ArrayDeque<>();   // new LinkedList<>()
       dq.add(1);
      dq.add(2);
      dq.add(3);
      dq.add(1);
      System.out.println("Deque after add operations: " + dq);
      dq.addFirst(5);
      dq.addLast(1);
      System.out.println("Deque after addFirst and addLast operations: " +
dq);
   }
}
```

Java Map Interface

1. Key - value pair.
2. Not part of collection because of this.
3. Keys must be unique.
4. Values duplicate allow.
5. Null value allow



What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

put(key , val) , get(key) ,containsKey(key), containsValue(val), remove(key)

size() , isEmpty() , clear()

keySet() => (Set<String> keys = map.keySet();) , map.values() , entrySet()

map.replace(key , value)

HashMap (order => unpredictable)

HashMap allows one null key

Example

```
import java.util.Map;

import java.util.HashMap;

import java.util.Set;

import java.util.Collection;

public class Main{

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

```
        HashMap<Integer , String> map = new HashMap<>(

         map.put(1, "Apple");

map.put(2, "Banana");

map.put(3, "Cherry");

map.put(4, "Date");

map.put(5, "Elderberry");

    System.out.println(map);

map.replace(5,"orange");

System.out.println(map);

System.out.println(map.size());

System.out.println(map.isEmpty());

System.out.println(map.get(1));

System.out.println(map.containsValue("Apple"));

 Set<Integer> keys = map.keySet();
```

```
Collection<String> values = map.values();

System.out.println("Keys: " + keys);

System.out.println("Values: " + values);

for(Map.Entry<Integer,String> obj : map.entrySet()){

   System.out.println(" " + obj.getKey() + " " + obj.getValue());

}
```

LinkedHashmap => internally doubly linked list . insertion order follow,

(order - insertion order)

TreeMap is class implement sortedmap . not null keys , keys unique , multiple null val allow , asc order. Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.(order => depend upon keys)

- **Java TreeMap is non synchronized.**

Java Hashtable class

- **A Hashtable is an array of a list. Each list is known as a bucket.**
- **The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.**

- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.

```
import java.util.*;
class Hashtable4{
 public static void main(String args[]){
   Hashtable<Integer,String> map=new Hashtable<Integer,String>();
    map.put(100,"Amit");
    map.put(102,"Ravi");
    map.put(101,"Vijay");
    map.put(103,"Rahul");
    System.out.println("Initial Map: "+map);
    //Inserts, as the specified pair is unique
    map.putIfAbsent(104,"Gaurav");
    System.out.println("Updated Map: "+map);
    //Returns the current value, as the specified pair already exist
    map.putIfAbsent(101,"Vijay");
    System.out.println("Updated Map: "+map);
 }
}
```

Output:

```
Initial Map: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
```

HashMap	Hashtable
1) HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is synchronized. It is thread-safe and can be shared with many threads.
2) HashMap allows one null key and multiple null values.	Hashtable doesn't allow any null key or value.
3) HashMap is a new class introduced in JDK 1.2.	Hashtable is a legacy class.
4) HashMap is fast.	Hashtable is slow.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is traversed by Iterator.	Hashtable is traversed by Enumerator and Iterator.
7) Iterator in HashMap is fail-fast.	Enumerator in Hashtable is not fail-fast.
8) HashMap inherits AbstractMap class.	Hashtable inherits Dictionary class.

Java Collections class

Import java.util.Collection;

METHODS : -
Collections.addAll(list, "Servlet","JSP");
Collections.max(list)
Collections.min(list)
int index = Collections.binarySearch(arrlist, "D");
Collections.fill(arrlist,"JavaTpoint"); fill all arrlist element la value Java Tpoint
Collections.reverse(mylist);
Collections.sort(list, Collections.reverseOrder()); etc

Java Features

Lambda Expression

```
package com.java.lambda;

interface Shape{
   void draw();
}

class reactangle implements Shape{
   public void draw(){
        System.out.println("in rectangle");
   }

}

class circle implements Shape{
   public void draw(){
        System.out.println("in circle");
   }

}

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

```
        Shape rectangle = () -> {System.out.println("i'm in recto");};  // for
you need to comment this for last print method wala
        //rectangle.draw();

        Shape circle = () -> {System.out.println("i'm in circle");
                                                };
        //circle.draw();

        print(rectangle);
        //or print(() -> {System.out.println("i'm in recto"));
        print(circle);
        //or print(() -> {System.out.println("i'm in circle"));
    }
    private static void print(Shape shape){
        shape.draw();
    }
}
```

Output **i'm in recto**
i'm in circle

Pass ParameterList :=

```
package com.java.lambda;

interface addable{
    int addition(int a , int b);
}

class addableimp implements addable{


    public int addition(int a, int b) {
        return (a+b);
    }
}
public class LambdaParameters {
    public static void main(String[] args) {
       addable add = (int a, int b) /* or (a,b) */ -> {
            return (a+b);  // or (a+b);
        };
      int res = add.addition(1,2);
        System.out.println(res);
    }
}
```

o/p = 3

Multiple Statement :=

```
addable add1 = (int a, int b) /* or (a,b) */ -> {
   int c = (a+b);  // or (a+b);
   return c;
};

int res1 = add.addition(5,5);
System.out.println(res1);
```

Runnable Interface implemented by using lambda exp

```
package com.java.lambda;

class Threaddemo implements Runnable{
   public void run(){
        System.out.println("run");
   }
}
public class RunnableLambda {
   public static void main(String[] args) {
        Thread thread = new Thread(new Threaddemo());
        thread.start();

        Runnable runnable = () -> {
            System.out.println("run in lambda");
        };
        Thread threadlambda = new Thread(runnable);
        threadlambda.start();
   }
}
```

FUNCTIONAL INTERFACE

```
package com.java.lambda;
@FunctionalInterface
public interface MyfunctionalInterface {
  void abstractMethod(String message);

  default void nonAbstractMethod() {
     System.out.println("This is a non-abstract method.");
  }
```

```
}
//new class
package com.java.lambda;
public class LamdaExpFunInter {
  public static void main(String[] args) {
     MyfunctionalInterface obj =(String msg) -> {
        System.out.println("func interface");
     };
     obj.abstractMethod("abstract method");

     obj.nonAbstractMethod();
  }
}
```

o/p => func interface
This is a non-abstract method.

Inbuilt method

1.Supplier<T> (GET() USED)

```
package com.java.lambda;

import java.time.LocalDateTime;
import java.util.function.Supplier;

class supp implements Supplier<LocalDateTime>{

   public LocalDateTime get(){
       return LocalDateTime.now();
   }
}
public class SupplierDemo {
   public static void main(String[] args) {
       Supplier<LocalDateTime>sup = () -> {
           return LocalDateTime.now();
       };
       System.out.println(sup.get());
   }
}
```

2 . Consumer<Type> (IT ACCEPT AND PRINT)

```java
package com.java.lambda;

import java.util.function.Consumer;

class consume implements Consumer<String>{

   public void accept(String ip){
       System.out.println(ip);
   }
}

public class Consumerdemo {
   public static void main(String[] args) {
       Consumer<String> con = (String ip) -> {
           System.out.println(ip);
       };
       con.accept("hello");
   }
}
```

Etc

METHOD REFERNCES

1.TO STATIC METHOD (class::staticmethodname)

```java
package com.java.lambda.MethodRefernce;

import java.lang.Math;
import java.util.function.BiFunction;
import java.util.function.Function;

public class MethodRefrencesDemo {
   public static int addition(int a ,int b){
       return (a+b);
   }
   // Method to demonstrate method references
   public static void main(String[] args) {


       // Method reference to the static method Math.sqrt
       Function<Integer, Double> fun = Math::sqrt;
```

```
        System.out.println(fun.apply(4)); // Output: 2.0

        //lamda
        BiFunction<Integer,Integer,Integer> biFunction = ( a , b) -> {
            return MethodRefrencesDemo.addition(a,b);
        };
        System.out.println(biFunction.apply(10,20));

        //static for method
        BiFunction<Integer,Integer,Integer> biFunction1 =
MethodRefrencesDemo::addition;
        System.out.println(biFunction1.apply(10,50));



    }
}
```

2. Instance method of particular obj (obj::instacnemethodname)

```
package com.java.lambda.MethodRefernce;
interface Abc{
    void print(String msg);
}
public class MethodRefrencesDemo {
    public  void display(String msg){
        msg = msg.toUpperCase();
        System.out.println(msg);
    }
    public static void main(String[] args) {
        MethodRefrencesDemo metho = new MethodRefrencesDemo();
        //instance method
        Abc obj = metho::display;
        obj.print("hello world");
    }
}
```

3.Refrence to an instance method of an arbitrary object specific type

Syntax = Class::instance method name

```
package com.java.lambda.MethodRefernce;

import java.lang.Math;
import java.util.function.BiFunction;
import java.util.function.Function;
```

```
public class MethodRefrencesDemo {
   // Method to demonstrate method references
   public static void main(String[] args) {

        Function<String,String> funco = (String input) -> input.toLowerCase();
        System.out.println(funco.apply("arbitary obj"));

        Function<String,String>funco1 = String::toLowerCase;
        System.out.println(funco1.apply("HEY BROTHER"));

   }
}
```

4.REFERNCE TO AN CONSTRUCTOR

CLASSNAME::NEW

```
package com.java.lambda.MethodRefernce;

import java.lang.Math;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.HashSet;

public class MethodRefrencesDemo {
   public static void main(String[] args) {
        //classname::new
        List<String> list = new ArrayList<>();
        list.add("apple");
        list.add("banana");
        list.add("apple");
        Function<List<String>, Set<String>> setfun = (fruitlist) -> new
HashSet<>(fruitlist);
        System.out.println(setfun.apply(list));


        Function<List<String>, Set<String>> setfun1 = HashSet::new ;
        System.out.println(setfun1.apply(list));

   }
}
```

OVERALL CODE

```
package com.java.lambda.MethodRefernce;

import java.lang.Math;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.HashSet;
import java.util.function.BiFunction;
import java.util.function.Function;

interface Abc{
   void print(String msg);
}
public class MethodRefrencesDemo {

   public  void display(String msg){
       msg = msg.toUpperCase();
       System.out.println(msg);
   }

   public static int addition(int a ,int b){
       return (a+b);
   }
   // Method to demonstrate method references
   public static void main(String[] args) {


       // Method reference to the static method Math.sqrt
       Function<Integer, Double> fun = Math::sqrt;
       System.out.println(fun.apply(4)); // Output: 2.0

       //lamda
       BiFunction<Integer,Integer,Integer> biFunction = ( a , b) -> {
           return MethodRefrencesDemo.addition(a,b);
       };
       System.out.println(biFunction.apply(10,20));

       //static for method
       BiFunction<Integer,Integer,Integer> biFunction1 =
MethodRefrencesDemo::addition;
       System.out.println(biFunction1.apply(10,50));

  MethodRefrencesDemo metho = new MethodRefrencesDemo();
       //instance method

       Abc obj = metho::display;
       obj.print("hello world");
```

```
        // refernce to an instacne method of an arbitary object specific type
        //class::instacne method name
        // input. => arbitary obj , string input = > method args of lambda
expression
        // tolowercase = > it's method

        Function<String,String> funco = (String input) -> input.toLowerCase();
        System.out.println(funco.apply("arbitary obj"));

        Function<String,String>funco1 = String::toLowerCase;
        System.out.println(funco1.apply("HEY BROTHER"));


        //REFRENCE TO CONSTRUCTOR
        //classname::new
        List<String> list = new ArrayList<>();
        list.add("apple");
        list.add("banana");
        list.add("apple");

        Function<List<String>, Set<String>> setfun = (fruitlist) -> new
HashSet<>(fruitlist);
        System.out.println(setfun.apply(list));


        Function<List<String>, Set<String>> setfun1 = HashSet::new ;
        System.out.println(setfun1.apply(list));

    }
}
```

O/P

2.0
30
60
HELLO WORLD
arbitary obj
hey brother
[banana, apple]
[banana, apple]