



# "Mastering Java Development: From Scratch to Pro"

# IN THIS COURSE, WE COVER THE SYLLABUS.

---

## 1. Basics Java

- Definition
- JVM
- JIT
- Syntax
- Variables
- Operators
- Control flow statement
- Looping statement
- Number systems
- Pattern program

## 2. CORE JAVA

- Code execution process in method
- Class and Interfaces
- Constructor and constructor chaining
- OOPS(Object Oriented Programming Language)
  - Polymorphism
  - Inheritance
  - Encapsulation
  - Abstraction
- Generalization and specialization
- Strings
- Arrays

### 3. Java Library

- Packages
- Wrapper class
- Exceptions and Exception handling
- User define exception
- Collections (List, Queue, Set)
- Map(HashMap, LikedHashMap, TreeMap)
- Threads, Multithreading and threads life cycle
- I/O Streams
- bean / pojo class
- File handling (creating, reading, writing)
- Lambda expression
- Serialization
- Garbage collector

### 4. Advance Java

- JDBC (Java Data Base Connectivity)

### 5. Frame work

- Spring
- Spring data jpa
- Hibernate
- Spring MVC
- Spring Boot
- Microservices basics

## **Required Tools:**

### **• For Basic and Core Java:**

- Edit Plus for beginners
- Eclipse IDE
- JDK 17 or JDK 1.8

### **• For Advanced Java:**

- MySQL or SQL Database

### **• For Frameworks:**

- Gradle or Maven

## Eclipse :

- Latest eclipse download from official website link is given below,  
<https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2024-09/R/eclipse-inst-jre-win64.exe>
- Extract the jar
- Open eclipse
- Go to eclipse installer
- If you are going to develop only stand alone application then click on Eclipse ide for java development.  
If you are going to develop stand alone along with Enterprise addition application then click on Eclipse ide for java development.
- Select the jdk path: make sure jdk 17 or 1.8 is installed
  - go to c drive,
  - then go program file,
  - Then go to java
  - Then select the jdk 17 or 1.8
- Guide Link is shown below  
<https://geekyscript.com/how-to-install-eclipse-ide-in-windows-10-step-by-step-complete-guide/>

## JDK 17 Downloading process:

- **Download:** Visit the official Oracle JDK 17 download page and select the Windows installer.
- **Run Installer:** After downloading the .exe file, run the installer and follow the on-screen instructions.

Set Environment Variables:

### Click on "Environment Variables" :

- Under System variables, click "New" to add.
- JAVA\_HOME = path to your JDK (e.g., C:\Program Files\Java\jdk-17).
- Path = add %JAVA\_HOME%\bin to the existing values.

Verify: Open Command Prompt and type java -version to check if it's installed correctly.

# Course : Java

# Part : Basic Java

## **Java Definition :**

- Java is a high-level programming language used to represent or correlate real-world entities.
- Java was developed by Sun Microsystems, with James Gosling as the lead developer, in 1995.

## **Program :**

Program is a set of instruction which is use to perform specific operation.

## **Programming language:**

The language which is use to create a set of instruction.

## **Type of Language:**

- **Low level Language :** which is understandable by machine. (Exa Binary code)
- **Middle level Language :** Which is minimum understand by human.(Exa byte code)
- **High level Language :** Which is understandable by human.(Exa java, Python, C++, C#, C)

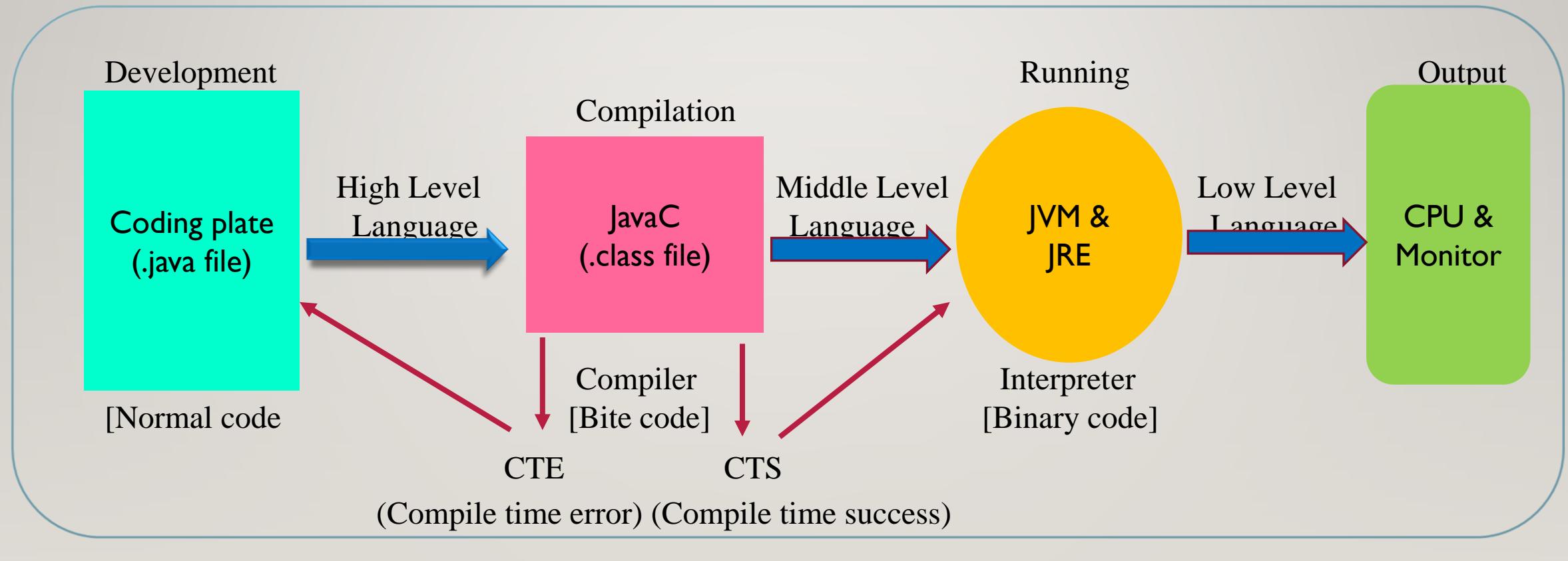
## **Steps :**

- A developer writes the code and saves it as a Java file, which is in a high-level language.
- Then, the file is sent for compilation, where it may result in either a compilation error or success.
- If the compilation is successful, the code is converted into bytecode.
- The bytecode is then executed by the JVM (**Java Virtual Machine**),
- which converts it into binary code. Finally, the result is displayed on the monitor.

## Types of file in java :

- **.java file** : high level programing language which is created by developer as a java file with .java extension.
- **.class file** : middle level programing language which is created by compiler as class file with .class extension.

## Process of Execution in java:

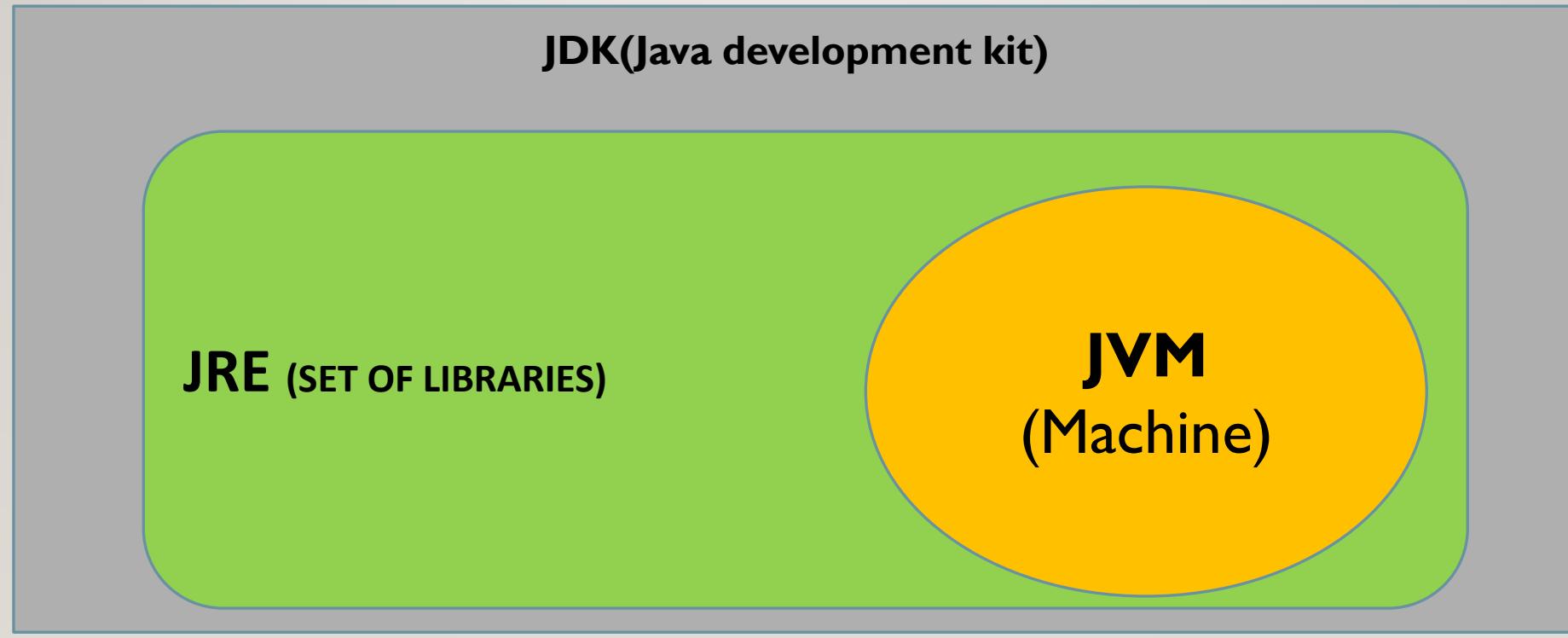


# Java Features:

- 1. Platform Independence:** Write Once, Run Anywhere" (WORA) via JVM.
- 2. Object-Oriented:** Supports OOP principles: Encapsulation, Inheritance, Polymorphism, Abstraction.
- 3. Robust:** Strong memory management, exception handling, and garbage collection.
- 4. Secure:** Built-in security features (e.g., security manager, bytecode verification).
- 5. Multi-threading :** Supports concurrent execution of multiple threads for better performance.
- 6. High Performance:** JIT compiler improves execution speed.
- 7. Dynamic:** Can load classes dynamically and supports reflection.
- 8. Rich Standard Library:** Extensive APIs for data structures, I/O, networking, and GUIs.
- 9. Automatic Memory Management:** Garbage collection eliminates memory management issues.
- 11. High-Level Language:** Abstracts low-level details, making development easier.
- 12. Scalability:** Suitable for both small and large-scale applications.

## **JVM (Java virtual machine) & JRE (Java run time environment):-**

It is responsible for execution, it convert the byte code into a binary code. this is send to a machine to display the output.



### Syntax:-

```
Class ClassName{  
    Public static void main(String[] arge){  
        code  
    }  
}
```

### Camel Case:

A naming convention where the first letter of the first word is lowercase, and the first letter of each subsequent word is uppercase (e.g., myVariableName), method and variable name follow camel case.

### Pascal Case:

A naming convention where the first letter of each word is uppercase (e.g., MyVariableName), Class and Constructor name follow pascal case.

## Data Type :-

Data type is used to create memory, it is also used to specify size of memory.

### Type:-

#### Primitive Datatype :

data type having size is called primitive data type.

1. **Boolean** : we are defining with true and false.

Exa : boolean a=true;

2. **Character** : any special and single alpha beta consider as a character, define by char inside single cod ('').

Exa : char a='x' or '%' or '1';

1. **Number** : byte : it is hole number, declare by byte and it is smallest number in size(1byte)

Exa : byte a=100; this is max size (max 3 digit)

➤ **short** : it is hole number, declare by short and it is greater than byte.(2byte)

Exa : **short** c=10000; (max 5 digit)

➤ **int** : it is hole number, declare by int and it is less than 10 digit.(4byte)

Exa : **int** d=1000000000; (max 10 digit)

➤ **long** : it is hole number, declare by long,L or l should mentioned at the last.(8byte)

Exa : **long** a=1000000000000000000L; (max 19 digit)

#### 4. Float :

- **float**: use to store decimal value, having small size, mention at the F or f. (4byte)  
Exa : `float f=10000000000000000000000000000000.0F;` (39 digit max)
- **double** :use to store decimal value, having large size. (8byte)  
Exa : `double f=10.0F;`

### Non-Primitive Datatype:- data type don't have side.

1. **String** : String is an immutable sequence(collection) of characters used to represent text. Writing in side double code ("").

Exa : `String a="a";`

2. **Array** : array in Java is a collection of elements of the same type stored in a contiguous memory location.

Exa : `int[] numbers = {1, 2, 3, 4, 5};`

2. **Collection** :A collection in Java is a framework that provides an architecture to store and group of objects.

Exa : `List<String> list = new ArrayList<>();  
list.add("Hello");`

3. **Predefine data type** : it is an instance of a class provided by the Java standard library .

## **Variable :**

it is a empty memory where we can store and declare the data.

Exa : **int number = 5, double d=5.0;**

**Rule to declare variable :** Syntax : **datatype variableName=value;**

1. Variable start with alpha bate and space not allowed (follow camel case)
2. Not allowed number and special character at the starting, at the starting accept ( \_ and \$) and alpha bate only.

## **Scanner Class:**

For taking run time input we use Scanner :

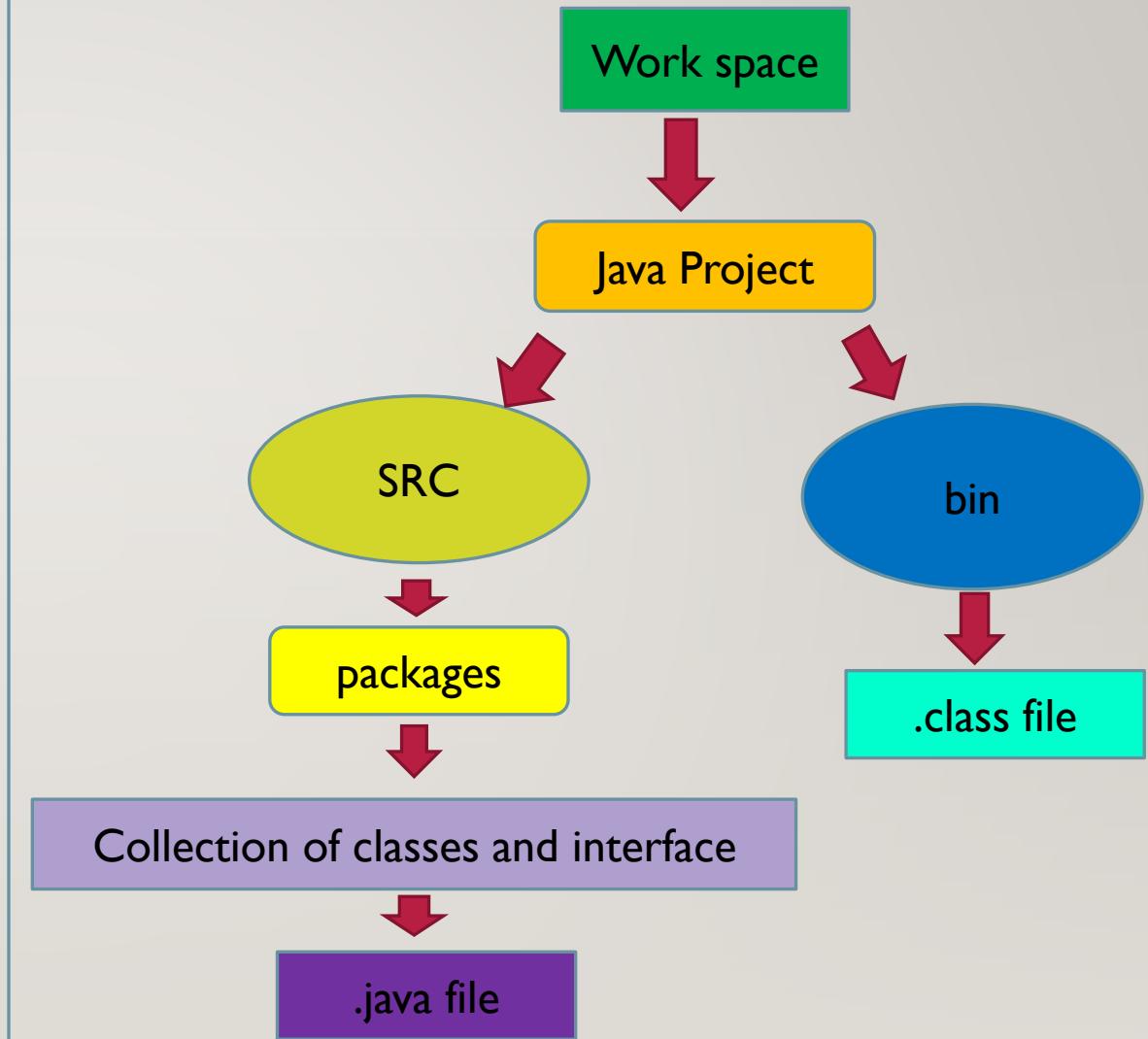
Scanner is use to take runtime input.

```
import java.util.Scanner;
public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        char c=scanner.next().charAt(0);
        scanner.close();
    }
}
```



## Eclipse Hierarchy:

- **Workspace:** A workspace is a folder where we store files and data, One workspace can contain multiple Java projects.
- **Java Project:** A Java project is a container for creating Java applications, It is a directory that organizes source code and other project files.
- **SRC Folder:** The SRC folder within a Java project is where we create Java packages. A single SRC folder can contain multiple packages.
- **Packages:** Packages are used to organize classes and interfaces. Each Java file typically contains one or more classes or interfaces.
- **bin Folder:** The bin folder is used to store the compiled class (.class file) files that are automatically generated by the compiler.



## Key words in java :

In Java, keywords are reserved words that have a predefined meaning, It is always small latter.

- |                      |                     |   |            |
|----------------------|---------------------|---|------------|
| 1. abstract          | 19. float           | 37. short   | 53. record |
| 2. assert            | 20. for             | 38. static  |            |
| 3. boolean           | 21. goto (not used) | 39. strictfp  |            |
| 4. break             | 22. if              | 40. super   |            |
| 5. byte              | 23. implements      | 41. switch  |            |
| 6. case              | 24. import          | 42. synchronized  |            |
| 7. catch             | 25. instanceof      | 43. this  |            |
| 8. char              | 26. int             | 44. throw   |            |
| 9. class             | 27. interface       | 45. throws  |            |
| 10. const (not used) | 28. long            | 46. transient   |            |
| 11. continue         | 29. native          | 47. try   |            |
| 12. default          | 30. new             | 48. void  |            |
| 13. do               | 31. null            | 49. volatile  |            |
| 14. double           | 32. package         | 50. while   |            |
| 15. else             | 33. private         | 51. var (added in Java<br>10 for local<br>variable type<br>inference) |            |
| 16. enum             | 34. protected       |   |            |
| 17. extends          | 35. public          |   |            |
| 18. final            | 36. return          |   |            |

## identifier :

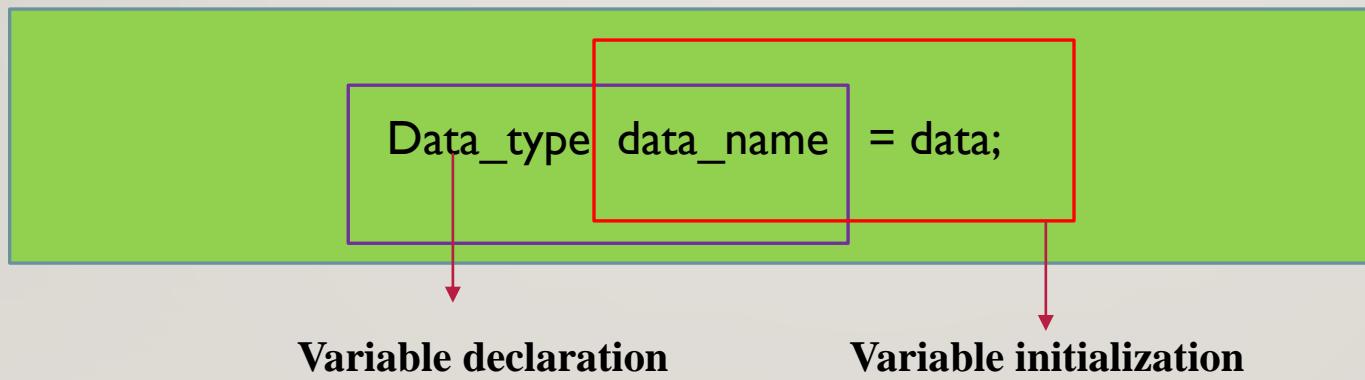
An identifier in Java is a name used to identify variables, methods, classes, or other entities, and it must start with a letter, underscore (\_), or dollar sign (\$), followed by any combination of letters, digits, underscores, or dollar signs. and which is not keyword.

Exa : **int myVariable = 5;**

Variable : variable is a memory block. Which has 4 property.

1. Variable Name
2. Variable Type
3. Variable Size
4. Variable Data

Re-initialization :- **variable\_name= data;**



## Variable initialization

- **byte** by=10;
- **short** so=100;
- **int** num=1;
- **long** lo=1000000L;
- **double** double=1.00;
- **float** float=1.00f;
- **char** char='@';
- **String** string="String";
- **boolean** bool=true;

## Operator :

Operator is used to perform operation.

### 1. Arithmetic Operators

Used to perform basic arithmetic operations.

Type : + : Addition and concatenation > - : Subtraction > \* : Multiplication > / : Division > % : Modulus (remainder)

Exa : int a = 10, b = 5; int sum = a + b; // sum = 15 (add) > String sum = a + b; // sum = 105 (con) >  
int diff = a - b; // diff = 5 > int prod = a \* b; // prod = 50 int div = a / b; // div = 2 > int mod = a % b; // mod = 0

### 2. Relational Operators

Used to compare two values.

Type : == : Equal to / != : Not equal to / > : Greater than / < : Less than / >= : Greater than or equal to / <= : Less than or equal to

Exa : int x = 10, y = 10; > boolean result = x > y; // result = false > boolean result = x < y; // result = false >  
boolean result = x <= y; // result = true > boolean result = x >= y; // result = true >  
boolean result = x == y; // result = true > boolean result = x != y; // result = false >

### 3. Logical Operators

It used to perform logical operations on boolean values.

Type : && : Logical AND > || : Logical OR > ! : Logical NOT

Exa : boolean a = true, b = false; > boolean result = a && b; // result = false > boolean result = a || b; // result = true > boolean result = !b; // result = true

## 4. Assignment Operators

Used to assign values to variables.

Type : = : Simple assignment > += : Add and assign > -= : Subtract and assign > \*= : Multiply and assign > /= : Divide and assign > %= : Modulus and assign

Exa : int x = 5; > x += 3; // x = x + 3; x = 8 > x \*= 2; // x = x \* 2; x = 16

## 5. Unary Operators

Which can be performed using single operator and single operand.

++ : Increment , - - : decrement

Int a=10;

- Pre Increment : first increased by 1 and update it. Exa : ++a +a=20
- Post Increment : first update it and increased by 1. Exa : (a++) +a=21
- Pre decrement : first decrees by 1 and update it. Exa : - -a +a=20
- Post decrement : first update it and decrees by 1. Exa : (a- -) +a=19

## 6. Binary Operators

Which can be performed using single operator and double operand.

Exa : a=10,b=10; a+b=20;

## 6. Ternary Operators

Which can be performed using two or more than two operators and two or more than two portent.

Exa : a=10; String result=( a==10 ) ? "Hello":"Byee";

## 7. Bitwise Operators

Used to perform bit-level operations on integer data.

Type : & : Bitwise AND / | : Bitwise OR / ^ : Bitwise XOR / ~ : Bitwise NOT / << : Left shift  
>> : Right shift

Example : int a = 5, b = 3; int result = a & b;

## 8. instanceof Operator

Checks if an object is an instance of a specific class or interface.

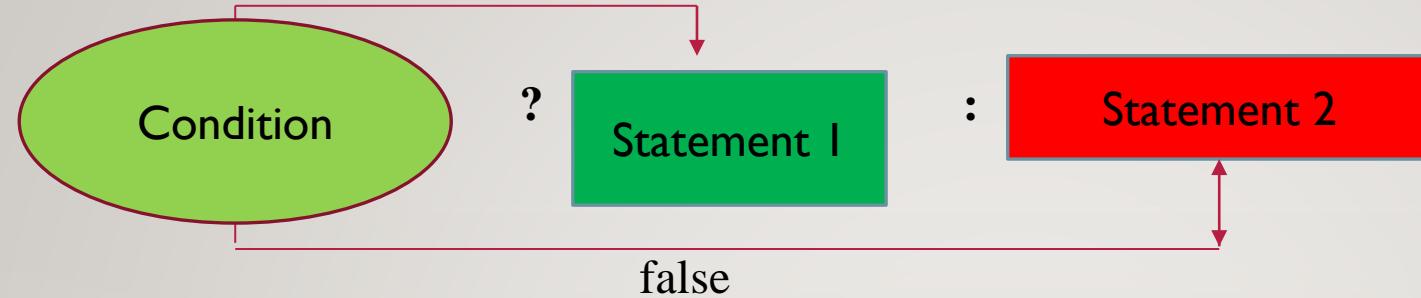
instanceof : Checks type compatibility

Exa : String str = "Hello"; boolean isString = str instanceof String;

## Ternary operator :

It is a one-liner conditional operator that takes three operands. It's used as a reduce the line of code, for if-else statements.  
true

Syntax:

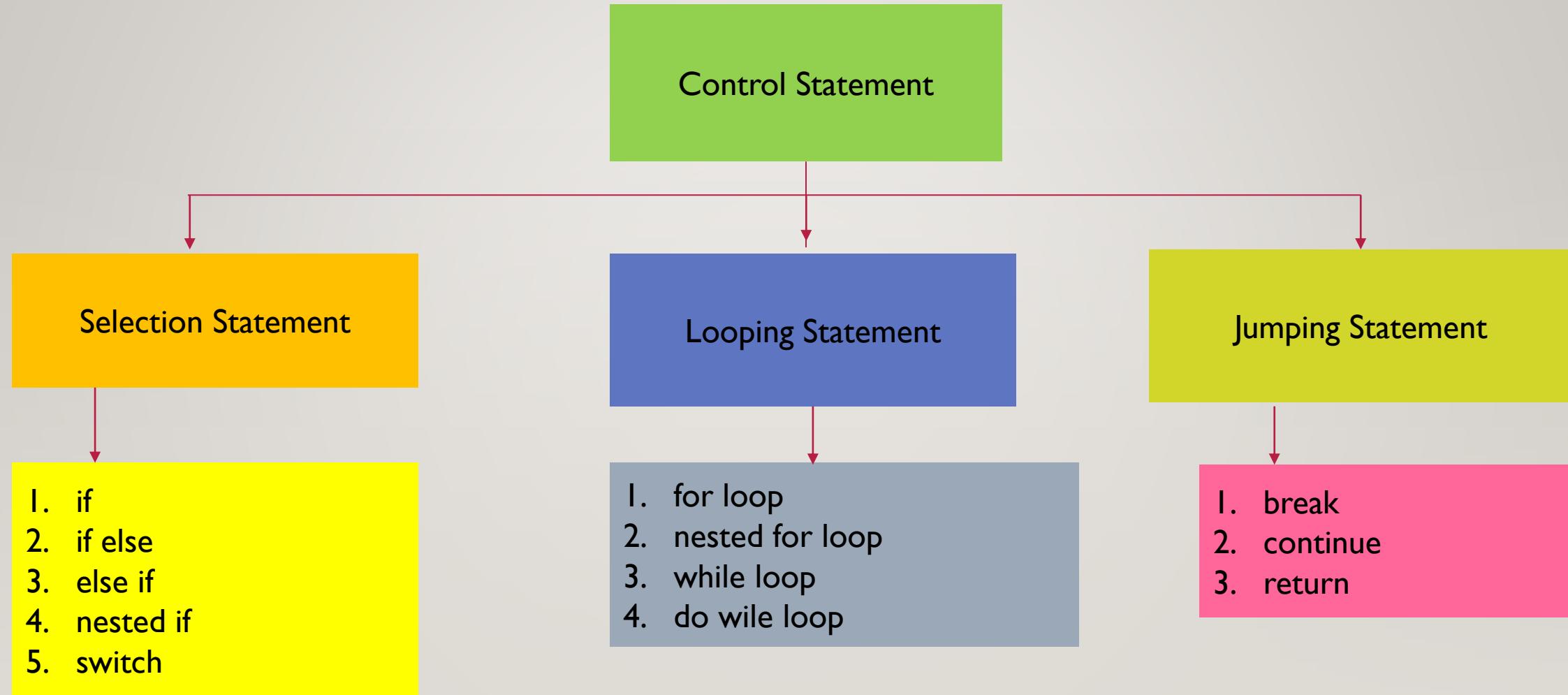


Exa : int a = 5, b = 10;

```
int min = (a < b) ? a : b; // min will be 5
```

## Control Statement:

Which is used to control the flow of execution. JVM always executes top to bottom and left to right.



## ASCII (American Standard Code for Information Interchange) :

ASCII value refers to the numerical representation of a character in the ASCII encoding standard, which maps characters like letters, digits, and symbols to integers (e.g., 'A' is 65, 'a' is 97).

**Exa :** A B C D E F G H I J K L M N O P Q R S T U V W X Y Z  
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90

a b c d e f g h i j k l m n o p q r s t u v w x y  
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121

z=122

INPUT1	INPUT2	OUTPUT	OPERATION
String	String	String	Concatenation
String	Number	String	Concatenation
String	Char	String	Concatenation
String	Bool	String	Concatenation
Bool	String	String	Concatenation
Number	Number	Number	Summation
Number	character	Number	ASCII
Character	Number	Number	ASCII
Character	Character	Number	ASCII

INPUT1	INPUT2	OUTPUT
Boolean	Boolean	Error
Boolean	Number	Error
Boolean	Character	Error
Number	Boolean	Error
Character	Boolean	Error

## If condition :

if condition in Java checks whether a condition is true or false and executes code based on that. if true execute it.

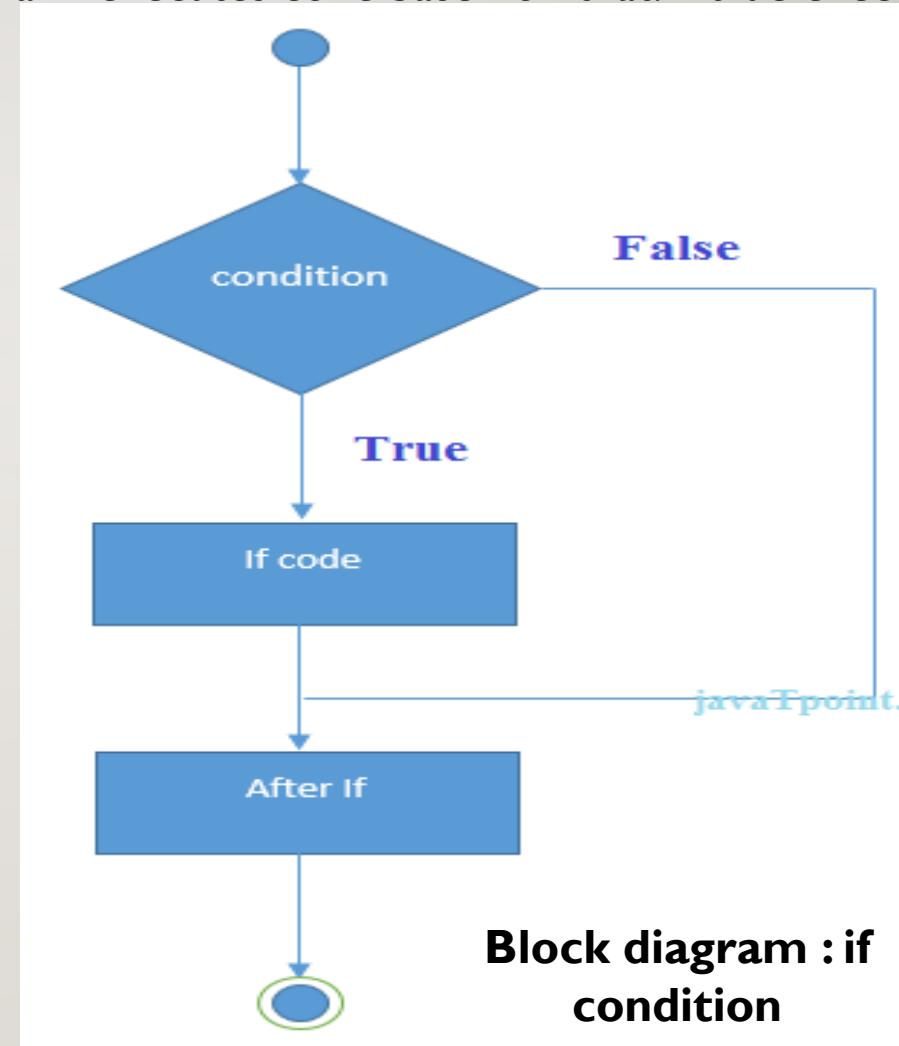
Exa :

```
int number = 10;  
if (number > 5) //true  
    System.out.println("Number is greater than 5");  
}
```

\*\*\*or \*\*\*

```
if (number > 5) //true  
    System.out.println("Number is greater than 5");
```

Note : if we have single line logic then, we no need to use

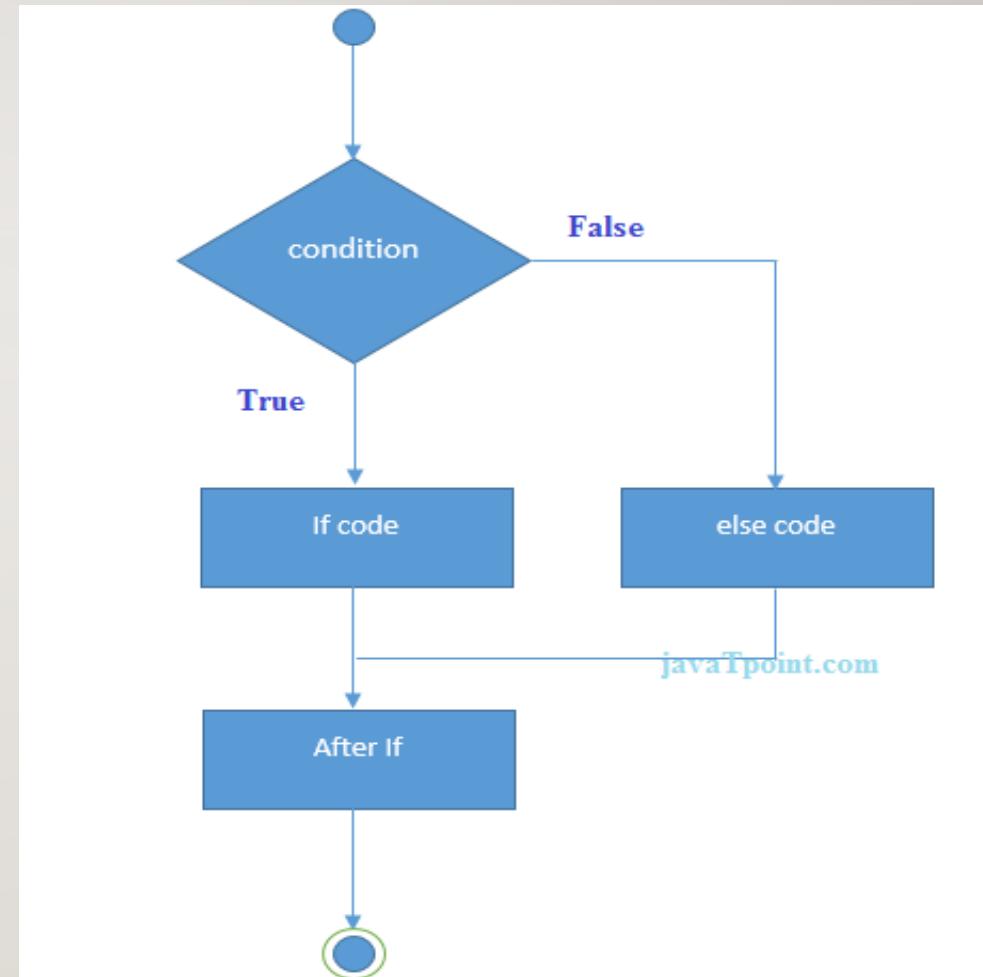


## If else condition :

if-else condition is used to execute one block of code if a condition is true and if it is false then else will execute.

Exa :

```
int num = 5;
if (num > 0) {           //true
    System.out.println("Positive number");
} else {
    System.out.println("Non-positive number");
}
```



block diagram :- if else condition

## **else if condition :**

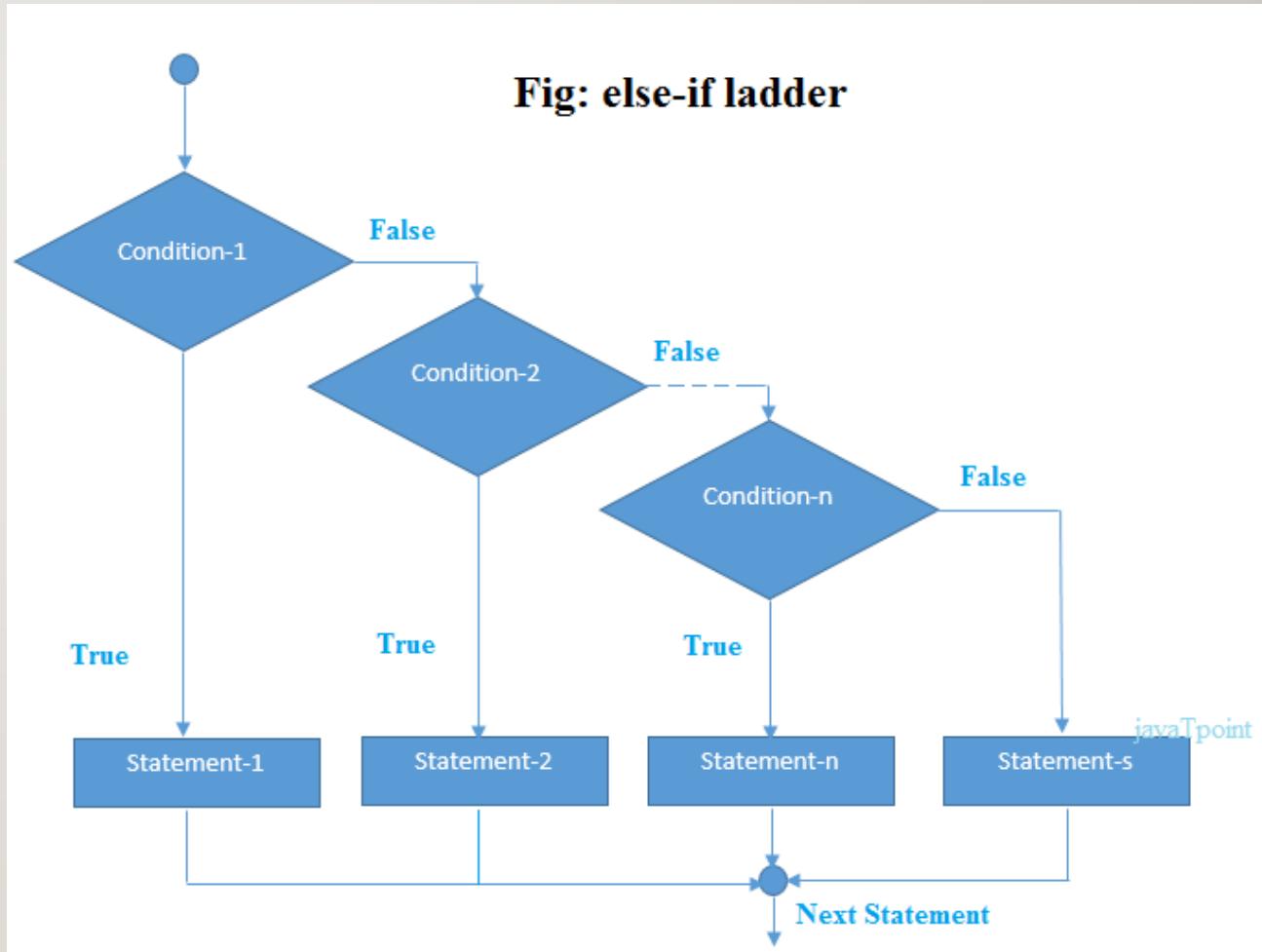
else if condition is used to test multiple conditions in sequence.

If the first if condition is false, the program checks the next else if condition, till condition is matched if all condition unmatched then else block will be executed.

**Exa :**

```
int num = 10;  
if (num > 15) {          //false  
    System.out.println("Greater than 15");  
} else if (num > 5) {    //true  
    System.out.println("Greater than 5");  
} else {  
    System.out.println("5 or less");  
}
```

**Fig: else-if ladder**

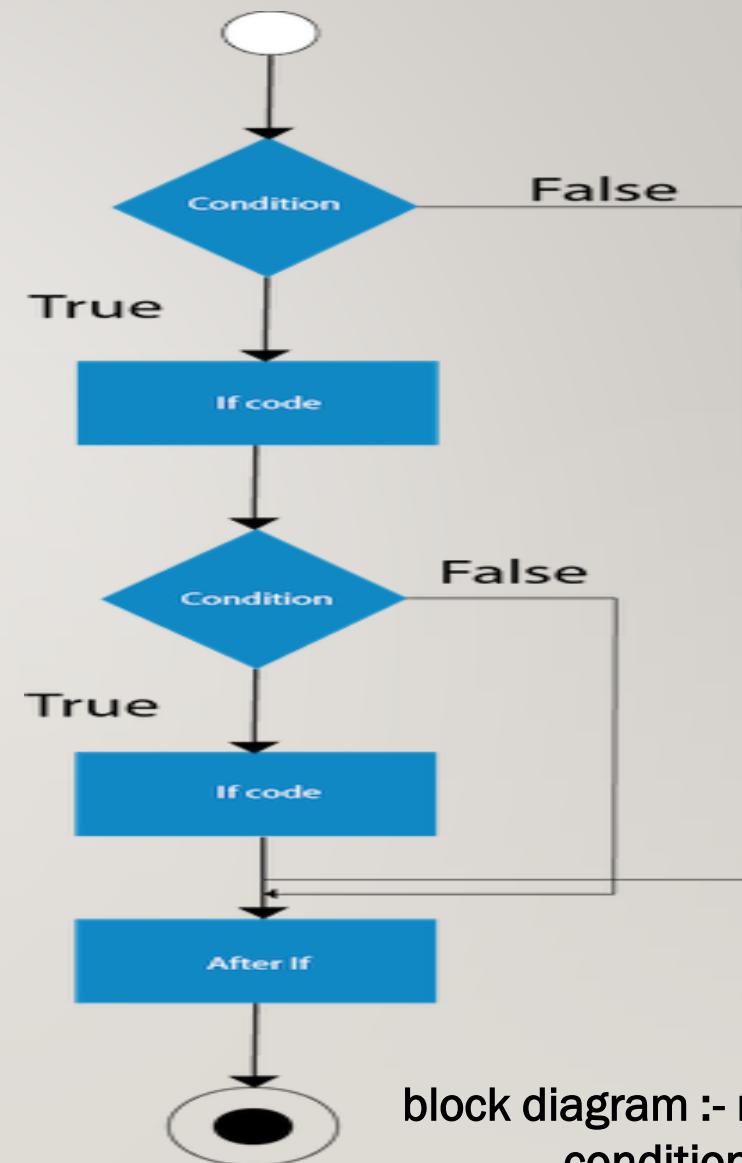


## Nested if condition :

if statement placed inside another if statement. It allows for multi-level decision-making. If outer if will become true then only it will go to inner if condition.

### Exa :

```
int num = 10;  
if (num > 5) {          //true  
    if (num < 15) {      //true  
        System.out.println("Number is between 5 and 15");  
    }  
}
```



block diagram :- nested if condition

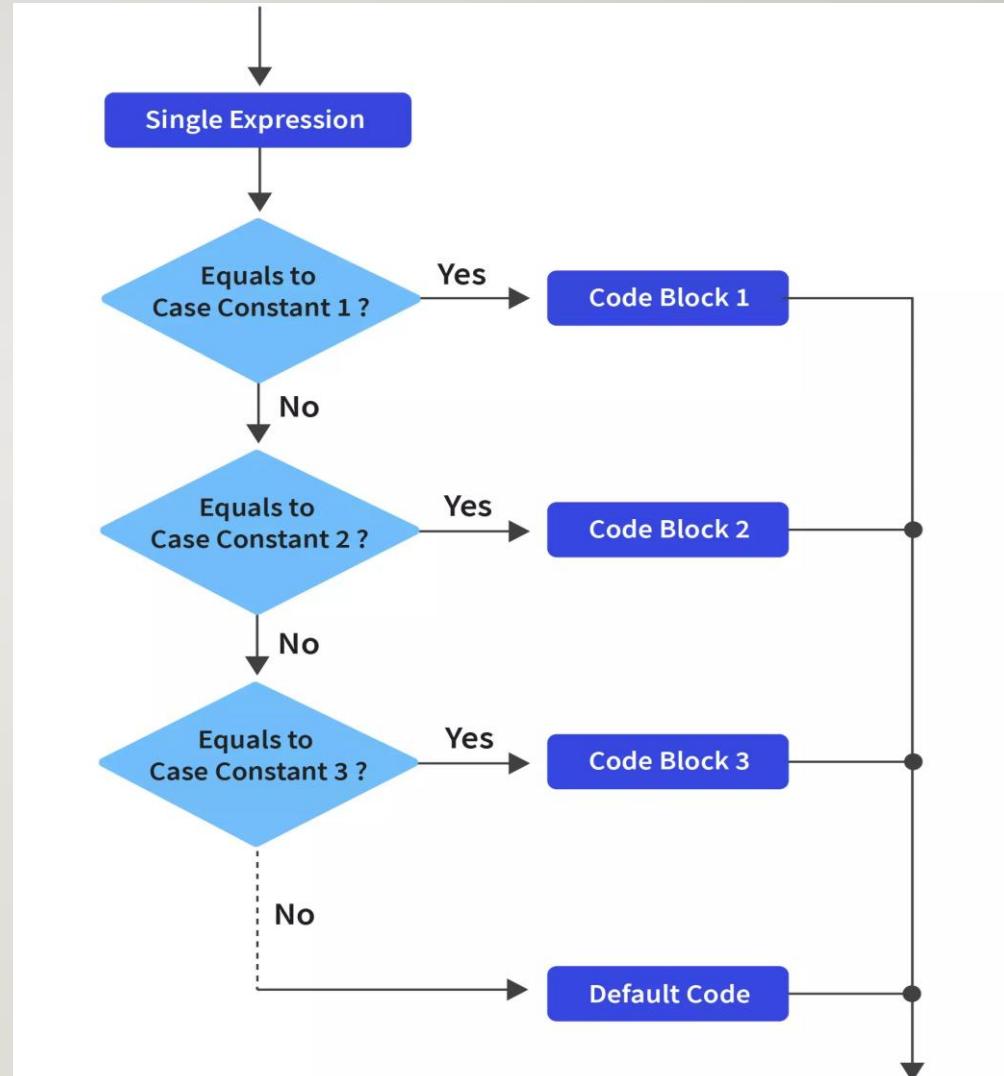
## Switch case :

it is used to select one among multiple options based on the value of an expression.

## Exa :

```
int day = 2;  
switch (day) {  
    case 1:  
        System.out.println("Monday");  
        break;  
    case 2:  
        System.out.println("Tuesday");  
        break;  
    default:  
        System.out.println("Other day");  
}
```

Output :Tuesday



block diagram :- Switch case

## For loop :

it is used to execute the block of code a specific number of times.

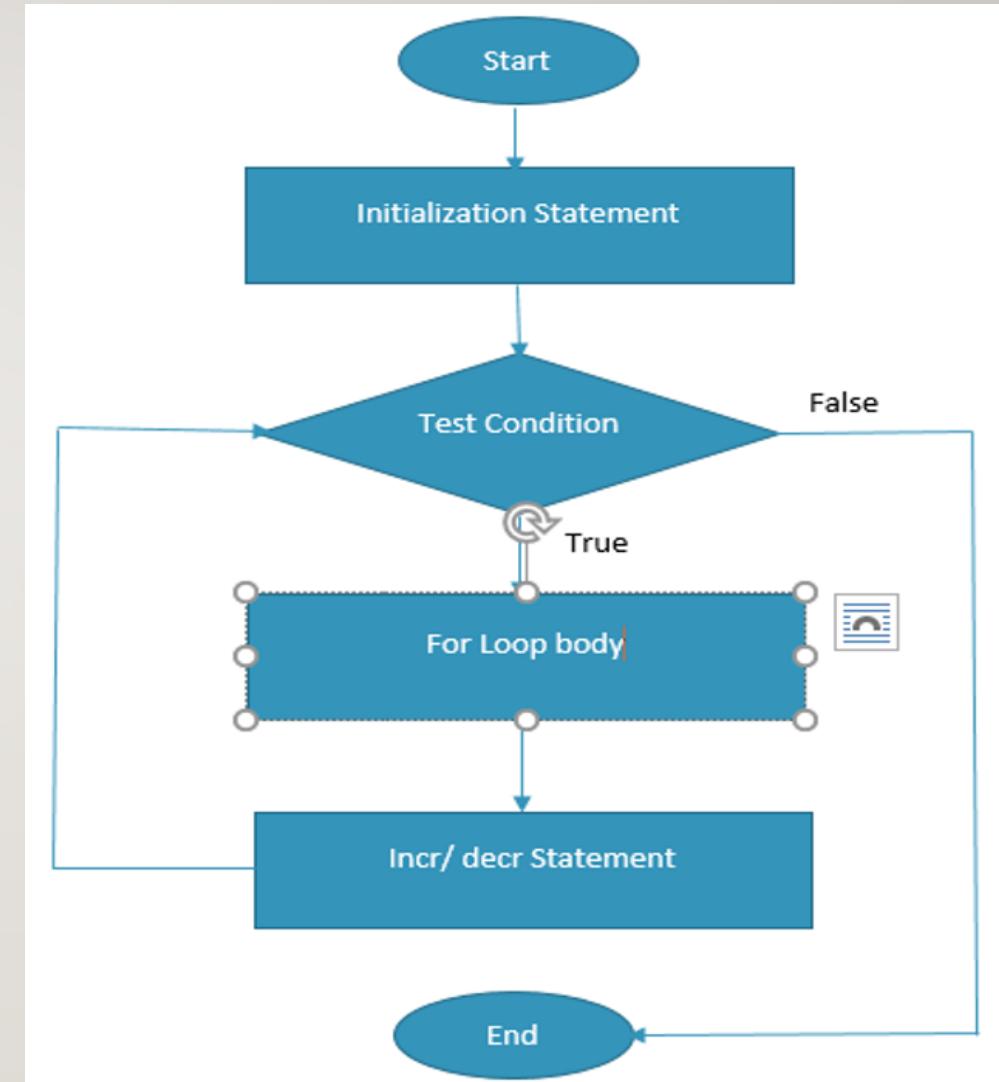
### Exa :

```
for (initialization; condition; update) {  
    // Code  
}  
  
public class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            System.out.print(i);  
            System.out.println(i);  
        }  
    }  
}  
o/p -      1 2 3 4 5
```

12345

Note : System.out.println(i);

this ln will print on next line.



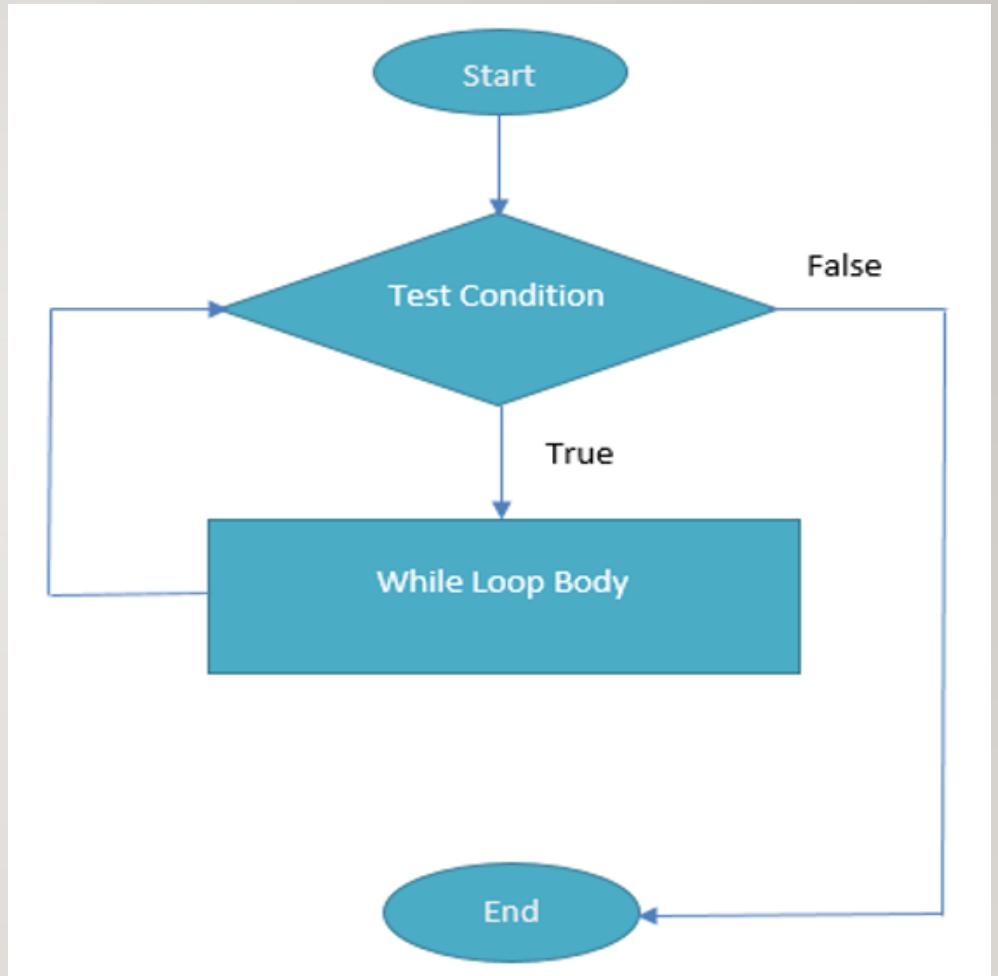
block diagram :- for loop

## While loop :

it repeatedly executes a block of code as long as the specified condition is true.

### Exa :

```
while (condition) { // Code  
Update;  
}  
public class Test {  
public static void main(String[] args) {  
int i = 1;  
while (i <= 5) {  
System.out.println(i);  
i++;  
}  
}  
}  
o/p - 1 2 3 4 5
```



block diagram :- while loop

## Do-while loop :

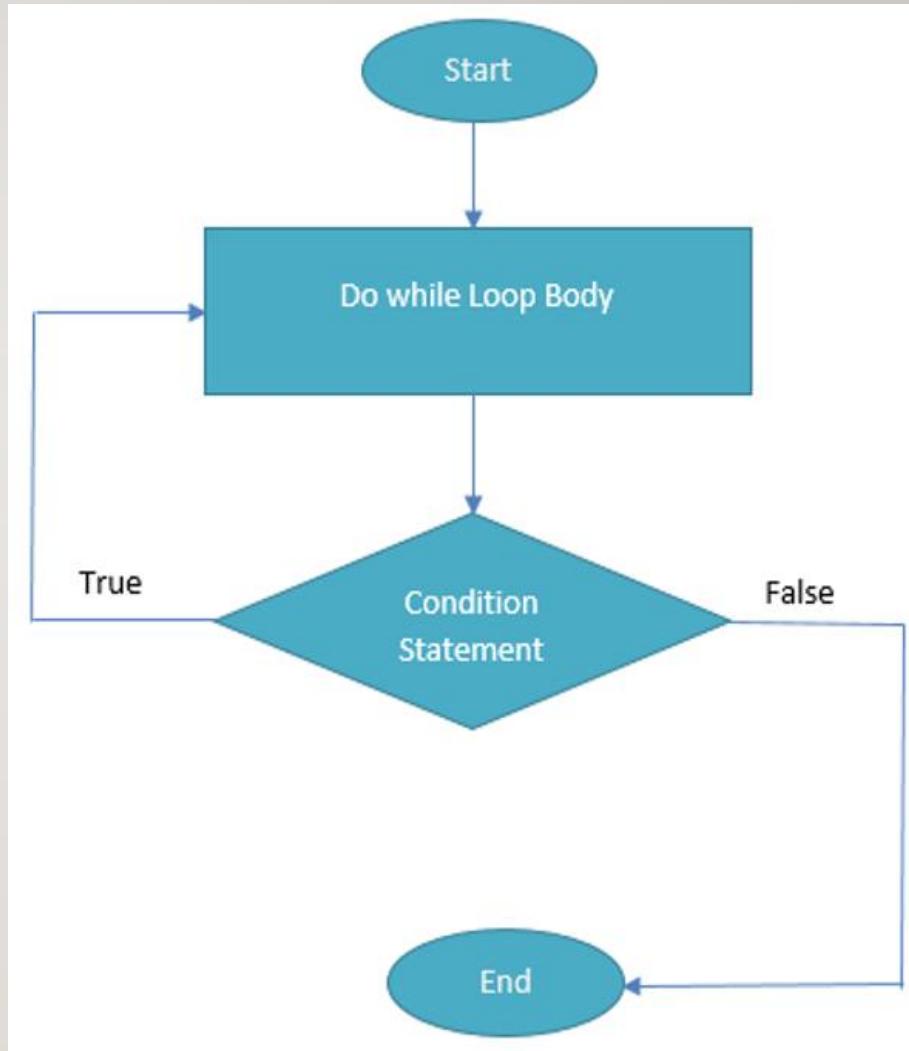
it is used to execute the code block at least once irrespective of checking of condition, and then repeats the loop as long as the condition is true.

Exa :

```
do { // Code
} while (condition);
public class Test {
public static void main(String[] args) {
int i = 1;
do {
System.out.println(i);
i++;
} while (i <= 5);
}
}
```

O/P -

```
1
2
3
4
5
```



block diagram :- Do-while loop

## Nested-for loop :

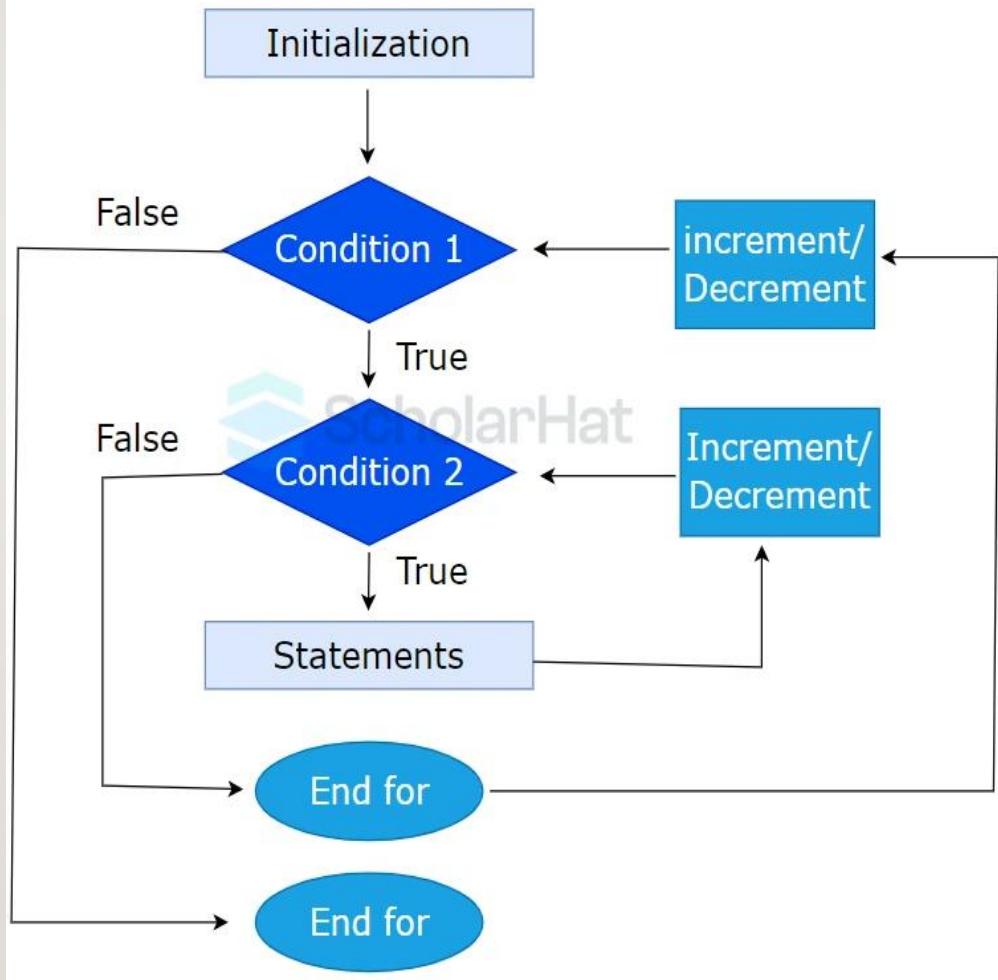
it is a loop inside another loop. The inner loop runs completely for each iteration of the outer loop.

Exa :

```
public class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 3; i++) { // Outer loop  
            for (int j = 1; j <= 2; j++) { // Inner loop  
                System.out.println("i = " + i + ", j = " + j);  
            }  
        }  
    }  
}
```

O/p :-      i = 1, j = 1  
              i = 1, j = 2  
              i = 2, j = 1  
              i = 2, j = 2  
              i = 3, j = 1  
              i = 3, j = 2

## Nested For Loop



block diagram :- Nested-for loop

## Assignment 1

- Write a program to Print Numbers from 1 to 10 by using for lop, while loop and do while loop.

Exa: 1 to 10 = 1      2      3      4      5      6      7      8      9      10

- WAP to Find the Sum of Numbers from 1 to N by using while loop Print Even Numbers from 1 to N

Exa: 1 to 5 = 2      4

- WAP to reverse a Number from given rage to 1 by using scanner class and for loop.

Exa: 1 to 5 = 1      2      3      4      5

- Write a program to print odd number from 1 to 100.

Exa: 1 to 5 = 1      3      5

- WAP to print number which is divisible by 3 or 5 from given range using scanner class.(use mod %)

Exa: 1 to 20 = 3      5      6      9      10      12      15      18      20

- WAP to print number which is divisible by 3 and 5 from given range using scanner class.(use mod %)

Exa: 1 to 30 = 15      30

- WAP to print sum of Nth natural number.

Exa 1 to 5 = 1+2+3+4+5=15

## Jumping Statement :-

**break:** it is use in loop or switch statement to break the condition.

**continue:** Skips the current iteration of a loop and proceeds with the next iteration.

**return:** it is use in method and condition, which is use to returns a value.

**Example:** break, continue, return

```
public class Example {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            if (i == 3) break; //Exit the loop when i==3  
            if (i == 2) continue; //Skip when i == 2  
            System.out.println(i); //Prints 1 and 4  
        }  
        System.out.println("End");  
    }  
}
```

```
public class ReturnExample {  
    public static void main(String[] args) {  
        int result = addNumbers(5, 3);  
        System.out.println("Sum: " + result);  
    }  
    public static int addNumbers(int a, int b) {  
        return a + b; // Return the sum of a and b  
    }  
}
```

## Q) Difference b/w for loop and while loop?

Feature	For Loop	While Loop
Syntax	<code>for(initialization; condition; update ){     //code }</code>	<code>while(condition){     //code     updates}</code>
Initialization	Done once at the start of the loop	Done before the loop starts (outside the loop)
Condition Checking	Checked before each iteration, continues as long as true	Checked before each iteration, continues as long as true
Update	Typically done in the loop header (increment/decrement)	Done inside the loop body or in the condition
Usage	Ideal when the number of iterations.	Ideal for situations where the number of iterations is unknown.
Example	<code>for(int i = 0; i &lt; 5; i++){}</code>	<code>int i = 0; while(i &lt; 5){ i++ }</code>
Termination	Loop terminates when the condition is false or when break is used	Loop terminates when the condition is false or when break is used

## Q) Difference b/w for do-while and while loop?

Feature	while Loop	do-while Loop
Condition Check	Condition is checked before the loop starts.	Condition is checked after the loop executes once.
Syntax	<pre>while (condition) {     // statements     Update; }</pre>	<pre>do {     // statements     Update; } while (condition);</pre>
Use Case	Used when the loop may not need to run at all based on the condition.	Used when the loop must run at least once before checking the condition.
Flow	Executes the statements only if the condition is true initially.	Executes the statements once and then checks the condition.
Example	<pre>while (x &lt; 5) { // code     Update; }</pre>	<pre>do { // code     Update; } while (x &lt; 5);</pre>

**Q1) write a program to check whether the integer is positive or negative?**

```
import java.util.Scanner;
public class Test {
public static void main(String[] args) {

Scanner sc = new Scanner(System.in);
System.out.println("Enter the number : ");
int num = sc.nextInt();
if (num > 0){
System.out.println(num + " is Positive");
} else if (num < 0){
System.out.println(num + " is Negative");
} else{
System.out.println("The number is Zero");
}
sc.close();
}
}
```

**Q2) WAP to check whether a character is uppercase or lowercase without using built-in methods?**

```
public class Test {
public static void main(String[] args) {
char ch = 'A'; // Example input
if (ch >= 'A' && ch <= 'Z')
System.out.println(ch + " is uppercase");
else if (ch >= 'a' && ch <= 'z')
System.out.println(ch + " is lowercase");
else
System.out.println(ch+"is not a letter");
}
}
```

\*\*\*Or down logic\*\*\*

```
if (ch >= 65 && ch <= 90)
System.out.println(ch + " is uppercase");
else if (ch >= 97 && ch <= 122)
System.out.println(ch + " is lowercase");
else
System.out.println(ch+"is not a letter");
```

**Q) WAP to count digit in by using while loop?**

```
import java.util.Scanner;
public class CountDigits {
public static void main(String[] args) {
Int number='12345';
int count = 0;
if {
while (number != 0) {
number /= 10; // Remove the last digit
count++; // Increment the count
}
}
System.out.println("Number of digits: "
+count);
}
```

**o/p :-** Number of digits: 5

**Q) WAP to sum of digit by using while loop using?**

```
import java.util.Scanner;
public class SumOfDigits {
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a number: ");
int number = scanner.nextInt();
int sum = 0;
while (number != 0) {
sum += number % 10; // Get the last digit and add to sum
number /= 10; // Remove the last digit
}
System.out.println("Sum of digits:" +sum);
scanner.close();
}
}
```

**o/p :-** Sum of digits: according to input

### \*\*\*spy number \*\*\*

A spy number is a number where the sum of digits is equal to the product of digits.

**Exa :** 123 (22, 123, 132)

Sum of digits:  $1 + 2 + 3 = 6$

Product of digits:  $1 * 2 * 3 = 6$

```
public class SpyNumberCheck {  
    public static void main(String[] args) {  
        int number = 121, sum = 0, product = 1, temp = number;  
        while (temp > 0) {  
            int digit = temp % 10;  
            sum += digit;  
            product *= digit;  
            temp /= 10;  
        }  
        if (sum == product)  
            System.out.println(number + " is a spy number.");  
        else  
            System.out.println(number + " is not a spy number.");  
        scanner.close();  
    }  
    o/p:- 112 is not a spy number
```

### \*\*\*Strong Number :\*\*\*

A Strong Number is a number for which the sum of the factorials of its digits is equal to the number itself.

**Exa :**  $145 = 1! + 4! + 5! = 1 + 24 + 120 = 145$ .

```
public class StrongNumber {  
    public static void main(String[] args) {  
        int num = 145, sum = 0, temp = num, digit;  
        while (temp != 0) {  
            digit = temp % 10;  
            int fact = 1;  
            for (int i = 1; i <= digit; i++) {  
                fact *= i;  
            }  
            sum += fact;  
            temp /= 10;  
        }  
        if (sum == num) {  
            System.out.println(num + " is a Strong number.");  
        } else {  
            System.out.println(num + " is not a Strong number.");  
        }  
    }  
    o/p:- 145 is a Strong number.
```

## Assignment 2

Q) WAP to print product of even digit by using while loop using?

Exa: 12345 = 8

Q) WAP to find the first digit of a given number using a while loop?

Exa: 12345 = 1

Q) WAP to find the last digit of a given number using a while loop?

Exa: 12345 = 5

Q) WAP an integer number as input and finds the first digit raised to the power of the last digit.

Exa: 12345 =  $1^5 = 1*1*1*1*1 = 1$

Q) WAP to find the largest and smallest digit from a given number?

Exa: 12345 = largest digit is : 5 and smallest digit is : 1

Q) WAP to find repeated digits in a given number?

Exa: 123253 = 2 and 3

Q) WAP for Strong numbers within a given range?

Exa: 12345 =

Q) WAP that reverses the second half of a given number and print it.

Exa: 123456 = 123654

### \*\*\* Palindrome Number\*\*\*

Given number is equal to the reverse of the number is called palindrome number.

**Exa :** 12321, 909, 45654, 1001, 888, 9009, 2332, 1234321, 12321.

```
public class Palindrome {  
    public static void main(String[] args) {  
        int num = 121, originalNum = num, reversedNum = 0;  
        while (num != 0) {  
            int digit = num % 10;  
            reversedNum = reversedNum * 10 + digit;  
            num /= 10;  
        }  
        if (originalNum == reversedNum)  
            System.out.println(originalNum + " is a palindrome.");  
        else  
            System.out.println(originalNum + " is not a palindrome.");  
    }  
}  
o/p:- 121 is a palindrome.
```

### \*\*\*Neon Number:\*\*\*

A number is a Neon number if the sum of the digits of the square of the number is equal to the number itself.

**Exa :**  $9 = 9*9 = 81 = 8 + 1 = 9$  (that means 9 is neon number)

```
public class NeonNumber {  
    public static void main(String[] args) {  
        int num = 9, square = num * num, sum = 0;  
        // Calculate the sum of the digits of the square  
        while (square > 0) {  
            sum += square % 10;  
            square /= 10;  
        }  
        if (sum == num)  
            System.out.println(num + " is a Neon Number.");  
        else  
            System.out.println(num + " is not a Neon Number.");  
    }  
}
```

### \*\*\*Perfect square \*\*\*

Multiply the number by it self will give you perfect square number.

**Exa :**  $2*2=4, 3*3=9, 4*4=16$

```
public class PerfectSquare {  
    public static void main(String[] args) {  
        int num = 16,sqrt = 0;  
        // Find the integer square root using a loop  
        for (int i = 1; i <= num / 2; i++) {  
            if (i * i == num) {  
                sqrt = i;  
                break;  
            }  
        }  
        if (sqrt != 0)  
            System.out.println(num + " is a perfect square. Square root is " + sqrt);  
        else  
            System.out.println(num + " is not a perfect square.");  
    }  
}
```

**O/P:- 16 is a perfect square. Square root is 4**

### \*\*\*Prime number:\*\*\*

prime number can only be divided by 1 and itself without leaving a remainder.

**Exa :** 2,3,5,7,11 etc.

```
public class PrimeNumber {  
    public static void main(String[] args) {  
        int number = 29;  
        boolean isPrime = true;  
        // 0 and 1 are not prime numbers  
        if (number <= 1) {  
            isPrime = false;  
        } else {  
            // Check for factors from 2 to sqrt(number)  
        }  
    }  
}
```

```
for (int i = 2; i <= number / 2; i++) {  
    if (number % i == 0) {  
        isPrime = false;  
        break; // No need to check further  
    }  
}  
if (isPrime)  
    System.out.println(number + " is a prime number.");  
else  
    System.out.println(number + " is not a prime number.");  
}  
}
```

### \*\*\*Armstrong Number:\*\*\*

An Armstrong number is a number which is sumation of each digit rise to the power of total number of digit is equal to the given number.

**Exa :**  $153 = 1^3 + 5^3 + 3^3 = 153$

```
public class Armstrong {  
    public static void main(String[] args) {  
        int num = 153, originalNum = num, sum = 0, digitCount = 0;  
        // Counting the number of digits  
        int tempNum = num;  
        while (tempNum != 0) {  
            digitCount++;  
            tempNum /= 10;  
        }  
    }  
}
```

// Checking if the number is an Armstrong number

```
tempNum = num;  
while (tempNum != 0) {  
    int digit = tempNum % 10; // Get the last digit  
    sum += Math.pow(digit, digitCount); // Raise digit to  
    //the power of the number of digits  
    tempNum /= 10; // Remove the last digit  
}  
if (sum == originalNum)  
    System.out.println(originalNum + " is an Armstrong  
    number.");  
else  
    System.out.println(originalNum + " is not an  
    Armstrong number.");  
}
```

**O/P:- 153 is an Armstrong number**

### \*\*\*Sunny Number:\*\*\*

A sunny number is a number  $n$  for which  $n + 1$  is a perfect square. For example:

**Exa :** If  $n = 3$ ,  $n + 1 = 4$ , which is a perfect square, so 3 is a

```
public class Main {  
    public static void main(String[] args) {  
        int n = 3; // Smallest sunny number candidate  
        // Check if n + 1 is a perfect square  
        int check = n + 1;  
        int squareRoot = 1;  
        while (squareRoot * squareRoot < check) {  
            squareRoot++;  
        }  
        if (squareRoot * squareRoot == check)  
            System.out.println(n + " is a sunny number.");  
        else  
            System.out.println(n + " is not a sunny number.");  
    }  
}  
  
o/p:- 3 is a sunny number.
```

### \*\*\*perfect number:\*\*\*

It is a positive integer that is equal to the sum of its proper divisor value, excluding itself.

**Exa :** 6 its divisors are 1, 2, 3, and  $1 + 2 + 3 = 6$ .

```
public class PerfectNumber {  
    public static void main(String[] args) {  
        int number = 6, sumOfDivisors = 0;  
        // Find divisors of the number and calculate their sum  
        for (int i = 1; i < number; i++) {  
            if (number % i == 0) { // Check if 'i' is a divisor  
                sumOfDivisors += i;  
            }  
        }  
        if (sumOfDivisors == number)  
            System.out.println(number + " is a perfect number.");  
        else  
            System.out.println(number + " is not a perfect number.");  
    }  
}  
  
o/p:- 6 is a perfect number.
```

## Assignment 3

Q) WAP that reverses the first half of a given number and print it?

Exa : 123456 = 321456

Q) WAP to check reverse of the number is perfect square or not?

Exa: 63 = 36 is perfect square or not.

Q) WAP to print Armstrong number between given range.

Exa: 1 to 50 =

Q) WAP to print prime number between given range.

Exa: 1 to 50=

Q) WAP to print  $n^{\text{th}}$  prime number from the number system.

Exa: 3<sup>rd</sup> prime number from hole number = 5

Q) WAP to print prime number between given range.  
Exa: 1 to 50 =

Q) WAP to print given number is prime number or not without using count?

Exa: 11 is prime or not. Or any number

Q) WAP to print perfect number between given range ?

Exa: 1 to 50 =

Q) WAP to print reverse Fibonacci series ?

Exa: 0 to 10 = 8 5 3 2 1 0

### \*\*\*Co-prime number\*\*\*

Two numbers are said to be co-prime if their greatest common divisor (GCD) is 1.

**Exa :** 4 ,5 => 4%1 && 5%1 = 0

### \*\*\* Automorphic number \*\*\*

automorphic number is a number whose square ends with the same digits as the number itself.

**exa:**  $5 = 5*5 = 25$

$6 = 6*6 = 36$

```
public class AutomorphicNumber {  
    public static void main(String[] args) {  
        int number = 5, square = number * number, temp =  
        number;  
        boolean isAutomorphic = true;  
        while (temp > 0) {  
            if (temp % 10 != square % 10) {  
                isAutomorphic = false;  
                break;  
            }  
            // Move to the next digit  
            temp /= 10;  
            square /= 10;  
        }  
    }  
}
```

if (isAutomorphic)

    System.out.println(number + " is an  
    Automorphic number.");

else

    System.out.println(number + " is not an  
    Automorphic number.");

}

}

**o/p:- 5 is an Automorphic number.**

### \*\*\* Decennium number :\*\*\*

Summation of the digit rise to the power of position of the digit is called decennium number.

**Exa :**  $175 = 1 + 7^2 + 5^3 = 175$

```
public class Dicerium {  
    public static void main(String[] args) {  
        int num = 153, originalNum = num, sum = 0,  
        digitCount = 0;  
        // Counting the number of digits  
        int tempNum = num;  
        while (tempNum != 0) {  
            digitCount++;  
            tempNum /= 10;  
        }  
        tempNum = num;  
        while (tempNum != 0) {  
            int digit = tempNum % 10;  
            sum += Math.pow(digit, digitCount);  
        }  
    }  
}
```

digitCount- -;

```
//the power of the number of digits  
tempNum /= 10; // Remove the last digit  
}  
if (sum == originalNum)  
    System.out.println(originalNum + " is an  
    Armstrong number.");  
else  
    System.out.println(originalNum + " is not an  
    Armstrong number.");  
}  
}
```

**o/p:- 153 is an Armstrong number**

### \*\*\*Fibonacci series\*\*\*

The Fibonacci series is the sum of the two preceding once.  
The sequence typically starts with 0 and 1.

**Exa :**

```
public class Fibonacci {  
    public static void main(String[] args) {  
        int a = 0, b = 1, c = 0;  
        for (int i = 0; c < 10; i++) {  
            System.out.println(c);  
            a = b;  
            b = c;  
            c = a + b;  
        }  
    }  
}
```

**Output:** Fibonacci Series: 0 1 1 2 3 5 8 13 21 34

**WAP to swap two numbers without using a third variable?**

**Exa :**

```
public class SwapNumbers {  
    public static void main(String[] args) {  
        int a = 5, b = 10;  
        System.out.println("Before swap: a = " + a + ", b = " + b);  
        // Swapping without using a third variable  
        a = a + b; // a now becomes 15  
        b = a - b; // b now becomes 5 (original value of a)  
        a = a - b; // a now becomes 10 (original value of b)  
        System.out.println("After swap: a = " + a + ", b = " + b);  
    }  
}  
  
o/p:- Before swap: a = 10, b = 20  
after swap: a = 20, b = 10
```

## WAP to convert Integer to binary?

```
public class Test {  
    public static void main(String[] args) {  
        int n = 10, bin = 1, rem = 0;  
        while (n > 0) {  
            rem = n % 2;  
            bin = bin * 10 + rem;  
            n = n / 2;  
        }  
        int rev = 0;  
        while (bin > 1) {  
            rem = bin % 10;  
            bin = bin / 10;  
            rev = rev * 10 + rem;  
        }  
        System.out.println(rev);  
    }  
}  
o/p:- 1010
```

## WAP to convert binary to integer?

```
public class Test {  
    public static void main(String[] args) {  
        int n = 1010, sum = 0, count = 0, rem = 0;  
        while (n > 0) {  
            rem = n % 2;  
            int p = 1;  
            for (int i = 0; i < count; i++) {  
                p = p * 2;  
            }  
            int mul = p * rem;  
            sum = sum + mul;  
            n = n / 10;  
            count++;  
        }  
        System.out.println(sum);  
    }  
}  
o/p:- 10
```

## Assignment 4

Print pattern :

1)

\* \* \* \*

\* \* \* \*

\* \* \* \*

\* \* \* \*

2)

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

3)

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

4)

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

5)

\* \* \* \* \*

\* \* \* \*

\* \* \* \*

\* \* \*

\*

6)

a b c d e f

g h i j k l

m n o p q r

s t u v w x

7)

1 2 3 4 5

A B C D E

6 7 8 9 10

F G H I J

8)

0

0

0

0

9)

0 0

0 0

0

0 0

0 0

0 0

10)

1

0

1

0

11)

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\*

11)

5 4 3 2 1

4 3 2 1

3 2 1

2 1

12)

0

1 0 1

0 1 0 1 0

1 0 1 0 1 0 1

2)

```
*  
* *  
* * *  
* * * *  
* * * * *  
  
package test;  
public class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= i; j++) {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

3)

```
*  
* *  
* * *  
* * * *  
* * * * *  
  
public class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= 5 - i; j++) {  
                System.out.print(" ");  
            }  
            for (int k = 1; k <= i; k++) {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

3)

a  
bc  
def  
ghij  
klmno

```
public class Test {  
    public static void main(String[] args) {  
        char ch = 'a';  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= i; j++) {  
                System.out.print(ch++);  
            }  
            System.out.println();  
        }  
    }  
}
```

4)

```
*  
***  
*****  
*****  
*****
```

```
public class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= 5 - i; j++) {  
                System.out.print(" ");  
            }  
            for (int k = 1; k <= 2 * i - 1; k++) {  
                System.out.print("*");  
            }  
            System.out.println();  
        }  
    }  
}
```

5)  
1  
2 3  
4 5 6  
7 8 9 10  
11 12 13 14 15

```
public class Test {  
    public static void main(String[] args) {  
        int num = 1;  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= i; j++) {  
                System.out.print(num++ + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

6)  
1  
1 2 1  
1 2 3 2 1  
1 2 3 4 3 2 1  
1 2 3 4 5 4 3 2 1

```
for (int i = 1; i <= 5; i++) {  
    for (int j = 1; j <= 5 - i; j++) {  
        System.out.print(" ");  
    }  
    for (int k = 1; k <= i; k++) {  
        System.out.print(k + " ");  
    }  
    for (int l = i - 1; l >= 1; l--) {  
        System.out.print(l + " ");  
    }  
    System.out.println();  
}
```

7)

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

```
public class Test {  
public static void main(String[] args) {  
for (int i = 1; i <= 5; i++) {  
for (int j = 1; j <= 5 - i; j++) {  
System.out.print(" ");  
}  
for (int k = 1; k <= i; k++) {  
System.out.print(k + " ");  
}  
System.out.println();  
}  
}
```

8)

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

```
public class Test {  
public static void main(String[] args) {  
for (int i = 1; i <= 5; i++) {  
for (int j = 1; j <= 5 - i; j++) {  
System.out.print(" ");  
}  
for (int k = i; k >= 1; k--) {  
System.out.print(k + " ");  
}  
System.out.println();  
}  
}
```

9)

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

```
public class Test {  
    public static void main(String[] args) {  
        for (int i = 5; i >= 1; i--) {  
            for (int j = 1; j <= 5 - i; j++) {  
                System.out.print(" ");  
            }  
            for (int k = i; k >= 1; k--) {  
                System.out.print(k + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

10)

```
1  
1 0  
1 0 1  
1 0 1 0  
1 0 1 0 1
```

```
public class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= 5 - i; j++) {  
                System.out.print(" ");  
            }  
            for (int k = 1; k <= i; k++) {  
                if (k % 2 == 1) {  
                    System.out.print("1 ");  
                } else {  
                    System.out.print("0 ");  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

## Assignment 4

Q) WAP to print reverse Fibonacci series?

Q) WAP to print reverse non-Fibonacci series?

Q) WAP to print non-Fibonacci series?

# Course : Java

## Part : Array, Strings And Object

## Object :

Object is a piece of memory.

Syntax : Object\_name obj\_reference\_variable\_name(ORV)=new Constructor\_name();

Note: new keyword is use to create object in java

Methods : There are 2 type of methods

- 1) Static method
- 2) Non-static method

Syntext to call :

```
MainMethodOrAnyMethod()  
ClassName. staticMethod();  
ClassName orv = new ClassName();  
orv. nonStaticMethod();  
}
```

```
staticMethod(){  
System.out.print("Hii");  
}
```

```
nonStaticMethod(){  
System.out.print("Byee");  
}
```

### #method without argument Syntax

```
Access specifier  access modifier  return type  methodName(){  
//logic or code  
}
```

### # method with argument Syntax

```
Access specifier  access modifier  return type  methodName(datatype argument){  
//logic or cod  
}
```

### # method without argument and return type Syntax

```
Access specifier  access modifier  return type  methodName(){  
//logic or code  
return value;  
}
```

### # method with argument and return type Syntax

```
Access specifier  access modifier  return type  methodName(datatype argument){  
//logic or cod  
return value;  
}
```

## **Static method :**

Static method having single copy nature in java , that's why it is binding during compilation time.

## **Non- satanic method:**

Non-static method having multiple copy nature. That's why that is binding during run time.

### **Non-Static Without Arguments:**

```
public void displayMessage() {  
    System.out.println("This is a non-static method.");  
}
```

### **Non-Static With Arguments:**

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

### **Static Method Without Arguments:**

```
public static void greet() {  
    System.out.println("Hello, world!");  
}
```

### **Static Method With Arguments:**

```
public static void add(int a, int b) {  
    System.out.println(a+b);  
}
```

### **Static Method Without Arguments return type :**

```
public static String greet() {  
    return "Hello, world!";  
}
```

### **Static Method With Arguments return type:**

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

### **Non-Static Without Arguments return type:**

```
public String displayMessage() {  
    return "This is a non-static method";  
}
```

### **Non-Static With Arguments return type:**

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

## Exa : method call

```
public class Test {  
    public static void main(String[] args) {  
        // Calling a static method directly using the class name  
        staticMethod();  
        // Creating an object to call a non-static method  
        Test obj = new Test();  
        obj.nonStaticMethod();  
    }  
    public static void staticMethod() {  
        System.out.println("This is a static method.");  
    }  
    public void nonStaticMethod() {  
        System.out.println("This is a non-static method.");  
    }  
}  
o/p:- This is a static method.  
This is a non-static method.
```

## Exa : with arguments method call

```
public class Test {  
    public static void main(String[] args) {  
        // Calling a static method with arguments  
        int sum = addNumbers(5, 10);  
        System.out.println("Sum: " + sum);  
        // Creating an object to call a non-static method with  
        // arguments  
        Test obj = new Test();  
        int product = obj.multiplyNumbers(2, 3);  
        System.out.println("Product: " + product);  
    }  
    public static int addNumbers(int a, int b) {  
        return a + b;  
    }  
    public int multiplyNumbers(int a, int b) {  
        return a * b;  
    }  
}  
o/p:-Sum: 15  
Product: 6
```

## Exa :

Static and no-static Method Without Arguments return type:

```
public class Test {  
    public static int addNumbers() {  
        return 10+ 10;  
    }  
  
    public int multiplyNumbers(int a, int b) {  
        return a * b;  
    }  
  
    public static void main(String[] args) {  
        // Calling a static method without arguments and getting the return  
        value  
        int sum = addNumbers();  
        System.out.println("Sum: " + sum);  
        // Creating an object to call a non-static method with arguments  
        Test obj = new Test();  
        int product = obj.multiplyNumbers(4, 2);  
        System.out.println("Product: " + product);  
    }  
}
```

Check if given number is perfect square or not?

```
import java.util.Scanner;  
public class Test {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.print("Enter a number: ");  
        int num = scanner.nextInt();  
        boolean isPerfectSquare = isPerfectSquare(num);  
        if (isPerfectSquare)  
            System.out.println(num + " is a perfect square.");  
        else  
            System.out.println(num + " is not a perfect square.");  
    }  
  
    public static boolean isPerfectSquare(int num) {  
        for (int i = 0; i < num; i++) {  
            int sqr=i*i;  
            if (sqr==num)  
                return true;  
        }  
        return false;  
    }  
}
```

## Access specifier :

which is used to specify the access layer. Where to access and where not.

## Type of access specifier :

- 1) **public** : which can be accessed anywhere in the Java .
- 2) **private** : which can be accessed on the class in the Java .
- 3) **protected** : which can be accessed on the same package and subclass in the Java .
- 4) **Default (no keyword)**: Accessible only within the same package.

## Access modifier :

They are used to define the scope of a variable or method.

## Type of access modifier :

- 1) **Static**: which is single copy nature and which is binding during compilation time.
- 2) **Non-Static**: which indicate multiple copy nature and which is binding during run time.

## Return type :

A return type specifies the data type of the value that a method returns to its caller. It is declared before the method name in the method declaration, if we don't return anything then put it as a void.

Access Specifier	Class	Package	Subclass	World
<b>public</b>	Yes	Yes	Yes	Yes
<b>protected</b>	Yes	Yes	Yes	No
<b>default (no-modifier)</b>	Yes	Yes	No	No
<b>private</b>	Yes	No 1	No	No

## \*\*\*Array\*\*\*

An array is a object which is used to store elements of the same data type.

points about arrays:

**Elements:** Each individual value in an array is called an element.

**Index:** Each element is associated with an index, which is used to access it.

**Declaration:** Arrays are declared by specifying the data type and the size of the array.

Syntax :

**1D array :**

```
ArrayType[ ] arrayName=new ArrayType[size];
```

**2D array:**

```
data_type[][] array_name = new data_type[rows][columns];
```

Exa:- int[ ] numbers = new int[5];

Object address

Object creation

0	1	2	3	..5
---	---	---	---	-----

length

## Explanation Of Array Syntax :

- New keyword will create an integer array object.
- Integer array elements consist of integer values, index values start from 0.
- Integer array object consists of length variable.
- length variable consists of the length of the elements.
- Array Name consists of the object address.
- Default value of int array is 0 for each element.

Exa:- **public class** Test {

```
public static void main(String[] args) {  
    int[] a = new int[4];  
    a[0] = 10;  
    a[1] = 20;  
    a[2] = 30;  
    a[3] = 40;  
    System.out.println(a[0]);  
    System.out.println(a[1]);  
    System.out.println(a[2]);  
    System.out.println(a[3]);  
}
```

O/p:- 10 20 30 40 ↓ print line by line

- If we initialize the elements more than the given size, we can get an exception at run time (ArrayIndexOutOfBoundsException).

Exa :By looping :

```
public class Test {  
    public static void main(String[] args) {  
        int[] a = new int[4];  
  
        a[0] = 10;  
        a[1] = 20;  
        a[2] = 30;  
        a[3] = 40;  
  
        for (int i = 0; i < a.length; i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

O/p:- 10 20 30 40 print line by line

2<sup>nd</sup> method to initialize array :

```
class Array {  
    public static void main(String[] args) {  
        int[] a = { 10, 20, 30, 40, 50};  
        for (int i = 0; i < a.length; i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

WAP to merge 2 integer array?

```
class Test {  
    public static void main(String[] args) {  
        int[] a = { 1, 2, 3, 4, 5 }, b = { 6, 7, 8, 9, 10 };  
        int[] c = new int[a.length + b.length];  
        for (int i = 0; i < c.length; i++) {  
            if (i < a.length) {  
                c[i] = a[i];  
            } else {  
                c[i] = b[i - a.length];  
            }  
        }  
        for (int i = 0; i < c.length; i++) {  
            System.out.print(c[i] + " ");  
        }  
    }  
}
```

## Assignment 5

➤ WAP to print odd index array elements.

Exa: { 10,20,30,40,50 } = 10 30 50

➤ WAP to print odd array elements.

Exa: { 1,2,3,4,5 } = 1 3 5

➤ WAP to print the elements which are divisible by 3.

Exa: { 1,2,3,4,5,6,7,8,9,10 } = 3, 6, 9

➤ WAP to print the sum of given array elements.

Exa: { 1,2,3,4,5 } = 15

➤ WAP to print sum of odd index array elements.

Exa: { 1,2,3,4,5 } = 9

➤ WAP to print the sum of first & last element.

Exa: { 1,2,3,4,5 } = 6

➤ WAP to swap 1st and last element.

Exa: {1,2,3,4,5} = 5 2 3 4 1

➤ WAP to print the given array elements in reverse order.

Exa: {1,2,3,4,5} = 5 4 3 2 1

➤ WAP to print palindrome no. in the given integer array (no element).

Exa: {10,11,12,13,22,55} = 11 22

➤ WAP to print perfect square no. from the given integer array element

Exa: {1,2,3,4,5} = 4

Q) `input int[] a={2,4,6,8};`

`o/p:- 3 5 7 9`

```
class Test {  
    public static void main(String[] args) {  
        int[] a = { 2, 4, 6, 8 };  
  
        for (int i = 0; i < a.length; i++) {  
            a[i] = a[i] + 1;  
        }  
  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
    }  
}
```

WAP to print smallest and largest number?

```
class Test {  
    public static void main(String[] args) {  
        int[] a = { -4, -5, -8, -2, -10 };  
        int largest = a[0];  
        int smallest = a[0];  
  
        for (int i = 0; i < a.length; i++) {  
            if (largest < a[i]) {  
                largest = a[i];  
            }  
  
            if (smallest > a[i]) {  
                smallest = a[i];  
            }  
        }  
        System.out.println("Largest:" + largest);  
        System.out.println("Smallest: " + smallest);  
    }  
}
```

### WAP to add 100 at the 1<sup>st</sup> position of array?

```
class AddExtraElement {  
    public static void main(String[] args) {  
        int[] a = {3, 4, 5, 6, 7};  
        int[] b = new int[a.length + 1];  
        for (int i = 0; i < b.length; i++) {  
            if (i == 0) {  
                b[i] = 100;  
            } else if (i == 1) {  
                b[i] = a[i - 1];  
            } else {  
                b[i] = a[i - 2];  
            }  
        }  
        for (int i = 0; i < b.length; i++) {  
            System.out.print(b[i] + " ");  
        }  
    }  
}
```

O/p:- 100 3 4 5 6 7

### Reverse the array element?

```
class Test {  
    public static void main(String[] args) {  
        int[] a = { 10, 20, 30, 40, 50 };  
  
        // Reversing the array in-place  
        for (int i = 0; i < a.length / 2; i++) {  
            int temp = a[i];  
            a[i] = a[a.length - 1 - i];  
            a[a.length - 1 - i] = temp;  
        }  
  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
    }  
}  
O/p:-50 40 30 20 10
```

## Character array :

it is a data structure used to store a sequence of characters. Each character in the array is assigned index, starting from 0.

## Syntax :

```
char[] letters = new char[5];      or      char[] vowels = {'a', 'e', 'i', 'o', 'u'};
```

Note: default value of character is space ( 1 empty space).

## Exa :

```
class Test {  
public static void main(String[] args) {  
char[] a = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' };
```

```
for (int i = a.length - 1; i >= 0; i--) {
```

```
System.out.print(a[i] + " ");
```

```
}
```

```
}
```

```
}
```

```
o/p:-z y x w v u t s r q p o n m l k j i h g f e d c b a
```

**Q) WAP to print only alpha bate ?**

```
class Test {  
    public static void main(String[] args) {  
        char[] a = { 'A', 'B', 'C', '@', 'E', '#', 'G', '*', 'I' };  
  
        for (int i = 0; i < a.length; i++) {  
            if ((a[i] >= 65 && a[i] <= 90) || (a[i] >= 97 && a[i] <= 122)) {  
                System.out.print(a[i] + " ");  
            }  
        }  
    }  
}
```

o/p:-A B C E G I

## Assignment 7

➤ WAP to print given character Alphabets in descending order.

Exa: {'a','d','x','t','c','m'} = x t m d c a

➤ WAP to print even character array element.

Exa: {'a','b','c','d','e','f'} =

➤ WAP to print odd character array element.

Exa: {'a','b','c','d','e','f'} =

➤ WAP to print sum of even index character array element along with their ASCII value.

Exa: {'a','b','c','d','e','f'} =

➤ WAP to merge two character array object elements into a third character array object.

Exa: {'a','b','c','d','e','f'} {"g','h','I'}

- WAP to insert new element to the given character array element at the starting.

Exa: {‘a’,’b’,’c’,’d’,’e’,’f’} = {‘z’,’a’,’b’,’c’,’d’,’e’,’f’}

- WAP to merge two character array object elements into a third character array object without repeating.

Exa: {‘a’,’b’,’c’,’d’,’f’} {‘a’,’d’,’x’,’t’,’c’,’m’}

o/p – a b c d f m t x

- WAP to insert new character array element at the index 3.

Exa: {‘a’,’b’,’c’,’d’,’e’,’f’} = {‘a’,’b’,’c’,’z’,’d’,’e’,’f’}

## Double Array in Java :

it is a data structure used to store multiple double or floating numbers.

Exa: `double[] numbers = {1.23, 4.56, 7.89, 10.11};`    or    `double[] numbers = new double[size];`

```
public class DoubleArrayExample {  
    public static void main(String[] args) {  
        double[] prices = {19.99, 24.99, 15.99};  
        for (int i = 0; i < prices.length; i++) {  
            System.out.println("Price: $" + prices[i]);  
        }  
    }  
}
```

## **String :**

String is an Immutable object in java, as well as it is a predefine class present in side `java.lang` package which is introduced in JDK 1.0, in simple term string is a group of character.

**New Keyword:** The new keyword is used to create a new object in memory.

**Object Reference Variable:** The string variable stores the memory address of the String object.

**String Elements:** The String object contains a sequence of characters. Each character has an index, starting from 0.

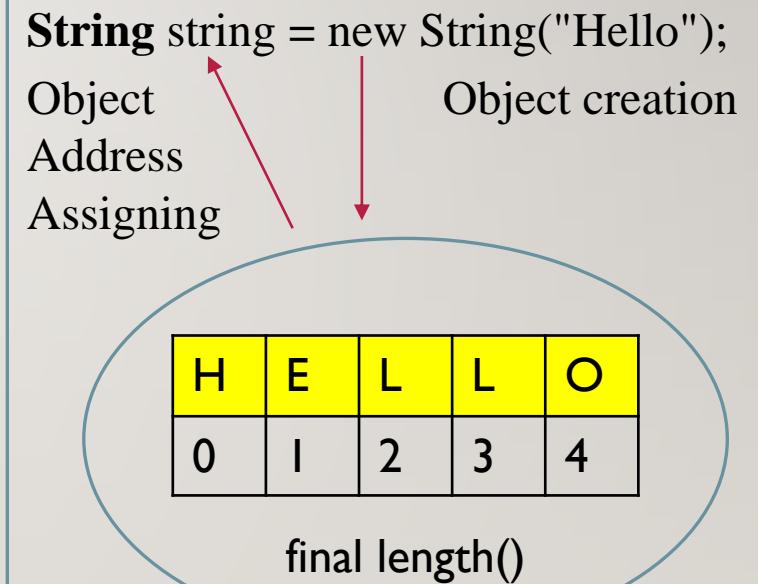
**Default Value:** default value of string is null.

**Length():** The length() method can be used to get the length of a String object.

**Immutable:** Once created, their object cannot be changed. if we try to change then it will create new object and changes will happened with newly created object.

**String Pool:** Java uses a String pool to optimize memory usage for strings.

When you create a String object, the JVM checks the String pool to see if an identical string already exists. If it does, the existing object is returned instead of creating a new one.



## Types of approach to create string object :

### #New Key Word Approach

- When we create an String object by using new keyword an object is created inside non-constant pool area.
- if the data is duplicated, because of new keyword, JVM will create a new object only.
- In non-constant pool area, String Object will be stored.
- We are creating foreach string new object because of new key word & then there is memory wastage.

### #Literals Approach

- When we create String Objects by using literals, objects are created in side constant pool area.
- If the object is duplicated, constant pool area is telling to JVM, instead of creating a new object, share with the existing object.
- This process is known as object sharing in Java.
- In constant pool area there is no new keyword we are using literals (" ") so we are not creating a new object.

**length(): int** :- Returns the length of the string.

**charAt(index): char** :- Returns the character at the specified index.

**concat(str):String** :- Concatenates the specified string to the end of this string.

**indexOf(str): int** :- Returns the index of the first occurrence of the specified substring.

**lastIndexOf(str):int** :- Returns the index of the last occurrence of the specified substring.

**substring(beginIndex):String** :- Returns a substring starting from the specified index.

**substring(beginIndex, endIndex): String** :- Returns a substring starting from the specified begin Index and ending at the specified end Index.

**toLowerCase(): String** :- Converts all characters to lowercase.

**toUpperCase(): String** :- Converts all characters to uppercase.

**trim(): String** :- Removes starting and last whitespace.

**startsWith(prefix):Boolean** :- Checks if the string starts with the specified prefix.

**endsWith(suffix): Boolean** :- Checks if the string ends with the specified suffix.  
1. **replace(old Char, new Char):** Replaces all occurrences of the old character with the new character.

**split(delimiter): String [ ]:-** Splits the string into an array of substrings based on the delimiter.

**equals(str): Boolean** :- Compares two strings object for equality.

**equalsIgnoreCase(str): Boolean** :- Compares two strings object for equality, ignoring case.

## Difference b/w String , String Buffer and StringBuilder ?

Feature	String	StringBuilder	StringBuffer
Definition	string is a Immutable object which contains collection of characters.	String builder is a Mutable sequence of characters.	String buffer Mutable sequence of characters.
Inbuilt Class	Yes, present in java.lang package	Yes, present in java.lang package	Yes, present in java.lang package
Introduced	Version 1.0	Version 1.0	Version 1.5
Object	All String objects are immutable	StringBuilder objects are mutable	StringBuffer objects are mutable
Methods	All methods present in String class are non-synchronized	All methods present in StringBuilder class are non-synchronized	All methods present in StringBuffer class are synchronized
Thread Safety	Because of non-synchronized nature, String objects are not thread-safe	Because of non-synchronized nature, StringBuilder objects are not thread-safe	Because of synchronized nature, StringBuffer objects are thread-safe
Performance	Because of non-thread-safe nature and multiple thread access, performance is low	Because of single-threaded access, performance is high	Because of multiple thread access, performance is high
Concatenation	concat(String)	append(String)	append(String)
Methods	equals(), hashCode(), toString()	toString()	toString()

## String Array in Java

it is a data structure that stores multiple strings. It's similar to a regular array, but each element is a reference to a String object.

**Exa :** `String[] fruits = {"apple", "banana".....};`    or    `String[] args =new String[size];`

**Exa :**

```
class Test {  
    public static void main(String[] args) {  
        String[] colors = { "red", "green", "blue" };  
        for (int i = 0; i < colors.length; i++) {  
            System.out.print(colors[i] + " ");  
        }  
    }  
}  
  
o/p:- red green blue
```

**WAP to convert given string to character and print it?**

```
class Test {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        char ch;  
        for (int i = 0; i < s1.length(); i++) {  
            ch = s1.charAt(i);  
            System.out.print(ch + " ");  
        }  
    }  
}  
  
o/p:- H e l l o
```

## Note Points:

1. When two users have created two string objects, to compare both strings, we make use of the inbuilt method equals().
2. We are not supposed to use comparison operator (==) as we are not comparing primitive values. We are comparing objects.
3. When the user has just initialized the value for the strings, JVM will create one object, at that time

## Exa:

```
class Palindrom {  
    public static void main(String[] args) {  
        String s1 = "mom";  
        String s2 = "MoM";  
        if (s1.equalsIgnoreCase(s2))  
            System.out.println("Equal");  
        else  
            System.out.println("Not equal");  
    }  
}  
o/p:- Equal
```

## Assignment 8

Question from string :

1. WAP to check given string is palindrome or not.
2. WAP to reverse a given string.
3. WAP to remove a given substring.
4. I/p : s1=J%A\$V@A  
o/p : s2=JAVA

WAP to check given string is palindrome or not.

```
class Test {  
    public static void main(String[] args) {  
        String s = "mom";  
        String rev = "";  
        char[] c = s.toCharArray();  
        for (int i = s.length() - 1; i >= 0; i--) {  
            rev += c[i];  
        }  
  
        if (s.equalsIgnoreCase(rev)) {  
            System.out.println("Given String is palindrome");  
        } else {  
            System.out.println("Given String is not palindrome");  
        }  
    }  
}  
o/p:- Given String is palindrome
```

## \*\*\*Anagram\*\*\*

An anagram is a word formed by rearranging the letters of another word . For example, "listen" and "silent" are anagrams.

**Note:** both string length should be equal.

```
class Test {  
    public static void main(String[] args) {  
        String str1 = "listen";  
        String str2 = "silent";  
        // Convert both strings to character arrays  
        char[] charArray1 = str1.toCharArray();  
        char[] charArray2 = str2.toCharArray()  
        // Sort both character arrays  
        Arrays.sort(charArray1);  
        Arrays.sort(charArray2);  
        if (Arrays.equals(charArray1, charArray2))  
            System.out.println("The two strings are anagrams.");  
        else  
            System.out.println("The two strings are not anagrams.");  
    }  
}
```

WAP to sort array via bubble sort?

```
class Test {  
    public static void main(String[] args) {  
        int[] a = { 42, 1, 7, 56, 98, 33 };  
  
        for (int i = 0; i < a.length - 1; i++) {  
            for (int j = 0; j < a.length - 1; j++) {  
                if (a[j] > a[j + 1]) {  
                    int temp = a[j];  
                    a[j] = a[j + 1];  
                    a[j + 1] = temp;  
                }  
            }  
        }  
  
        for (int i = 0; i < a.length; i++) {  
            System.out.print(a[i] + " ");  
        }  
    }  
}  
o/p:-1 7 33 42 56 98
```

**Q) WAP to print lexicographical series from given string?**

```
class Test {  
    public static void main(String[] args) {  
        String originalString =  
            "Aakash,Baby,Ajay,Raju,Bedprakash,Krishnamurthy";  
        String[] sortedString = originalString.split(",");  
        for (int i = 0; i < sortedString.length; i++) {  
            for (int j = i + 1; j < sortedString.length; j++) {  
                if (sortedString[i].compareTo(sortedString[j]) > 0) {  
                    String temp = sortedString[i];  
                    sortedString[i] = sortedString[j];  
                    sortedString[j] = temp;  
                }  
            }  
        }  
        for (int i = 0; i < sortedString.length; i++) {  
            System.out.println(sortedString[i]);  
        }  
    }  
}
```

**O/p:-**Aakash Ajay Baby Bedprakash Krishnamurthy  
Raju

**WAP to print nth max or min from given array?**

```
class Test {  
    public static void main(String[] args) {  
        int[] a = { 12, 35, 1, 10, 34, 1 };  
        int n = 2;  
        for (int i = 0; i < a.length; i++) {  
            int countL = 0, countS = 0;  
            for (int j = 0; j < a.length; j++) {  
                if (a[i] < a[j]) {  
                    countL++;  
                }  
                if (a[i] > a[j]) {  
                    countS++;  
                }  
            }  
            if (countL == (n - 1)) {  
                System.out.println(n+"th largest no from given array is "+a[i]);  
            }  
            if (countS == (n )) {  
                System.out.println(n+"th smallest no from given array is "+a[i]);  
            }  
        }  
    }  
}
```

**O/p:-**2th smallest no from given array is 10  
2th largest no from given array is 34

## Foreach loop:

A foreach loop is a control flow statement, that iterate over the object. like arrays, lists, maps and predefine object. Introduce in JDK 1.5.

### Syntax:

```
for (dataType variableName : collectionName) {  
    // Code to be executed for each element  
}
```

### Example:

```
int[] numbers = {1, 2, 3, 4, 5};  
for (int number : numbers) {  
    System.out.println(number);  
}
```

Feature	for Loop	foreach Loop
Syntax	for(initialization; condition; increment/decrement){ }	for(dataType variableName : collectionName) { }
Flexibility	Highly flexible, allows for precise control over the iteration process.	Less flexible, cannot modify the collection elements during iteration.
Efficiency	Less efficient as compare to foreach loop for complete iteration.	Generally more efficient for simple iterations.
Readability	Can be less readable for complex iterations.	More readable for simple iterations.

# Course : Java

## Part : Core Java



## **Class and Object (Imp) :**

As soon as class is created new data type will be created this data type is called user define data type or non-primitive datatype.

### **User define data type**

It is created by the user not predefine.

### **Derived data type**

It is predefine data type in java.

### **Primitive data type**

Already define in java, which having length is called primitive data type.

Exa : int, char, float, double, byte, short, long

### **Non-primitive data type**

It is not introduced in java, we need to define.

Exa : String, Array, Collection and predefine class.

### **Key Point :**

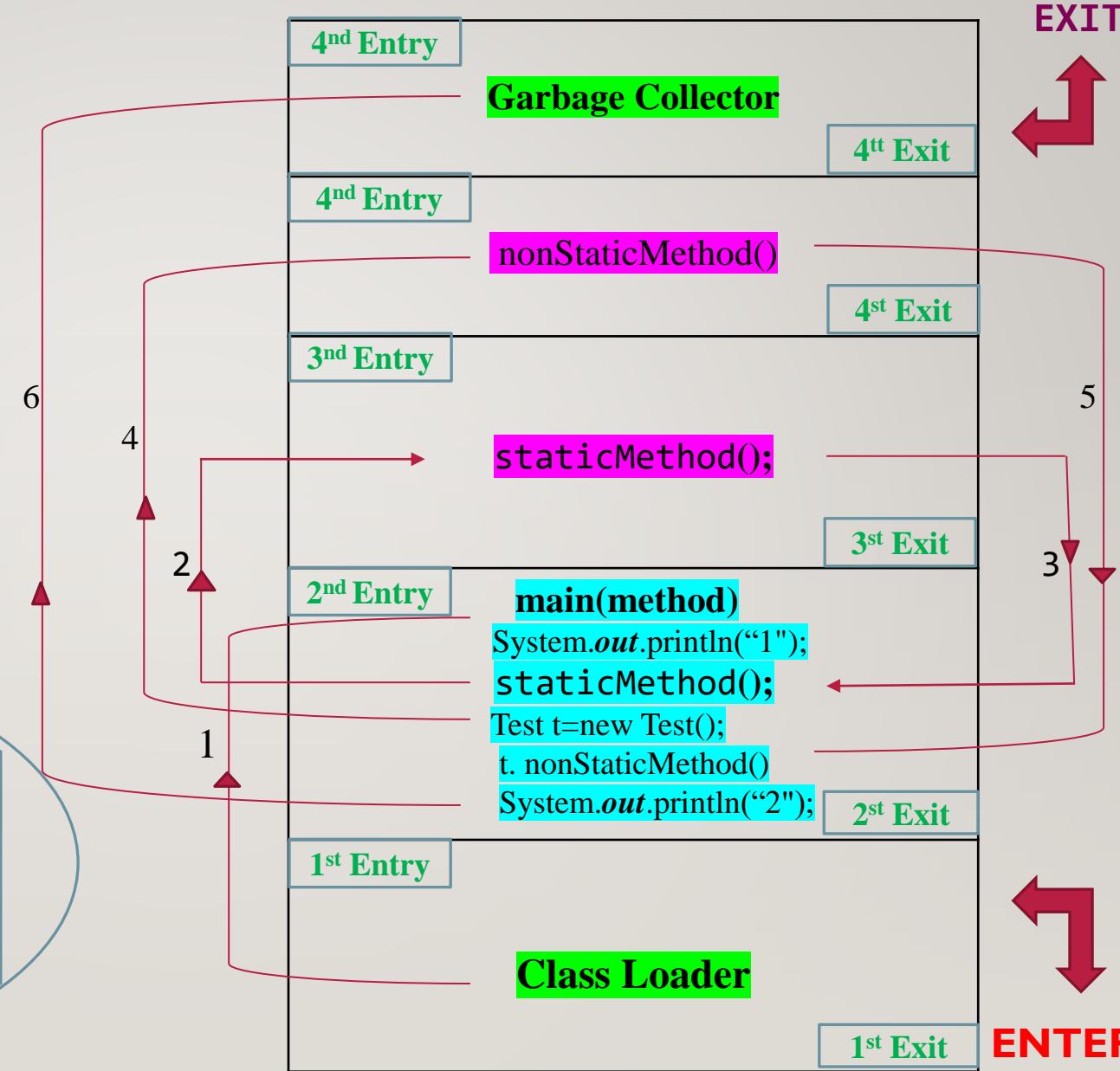
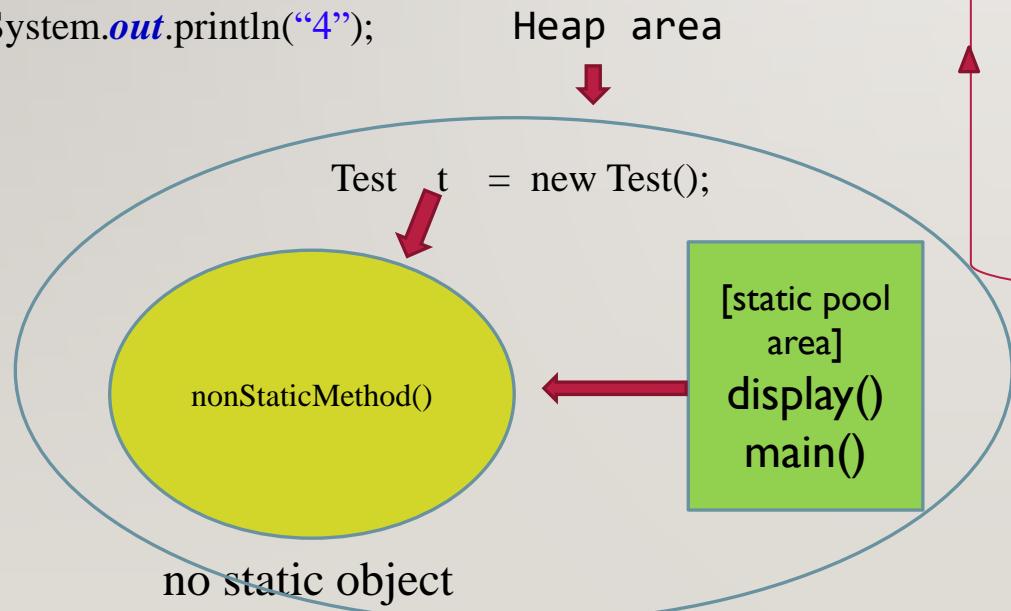
- As soon as class is created, new datatype is created
- Java language represent everything in the form of object.
- An object can be leaving or non-leaving entity.

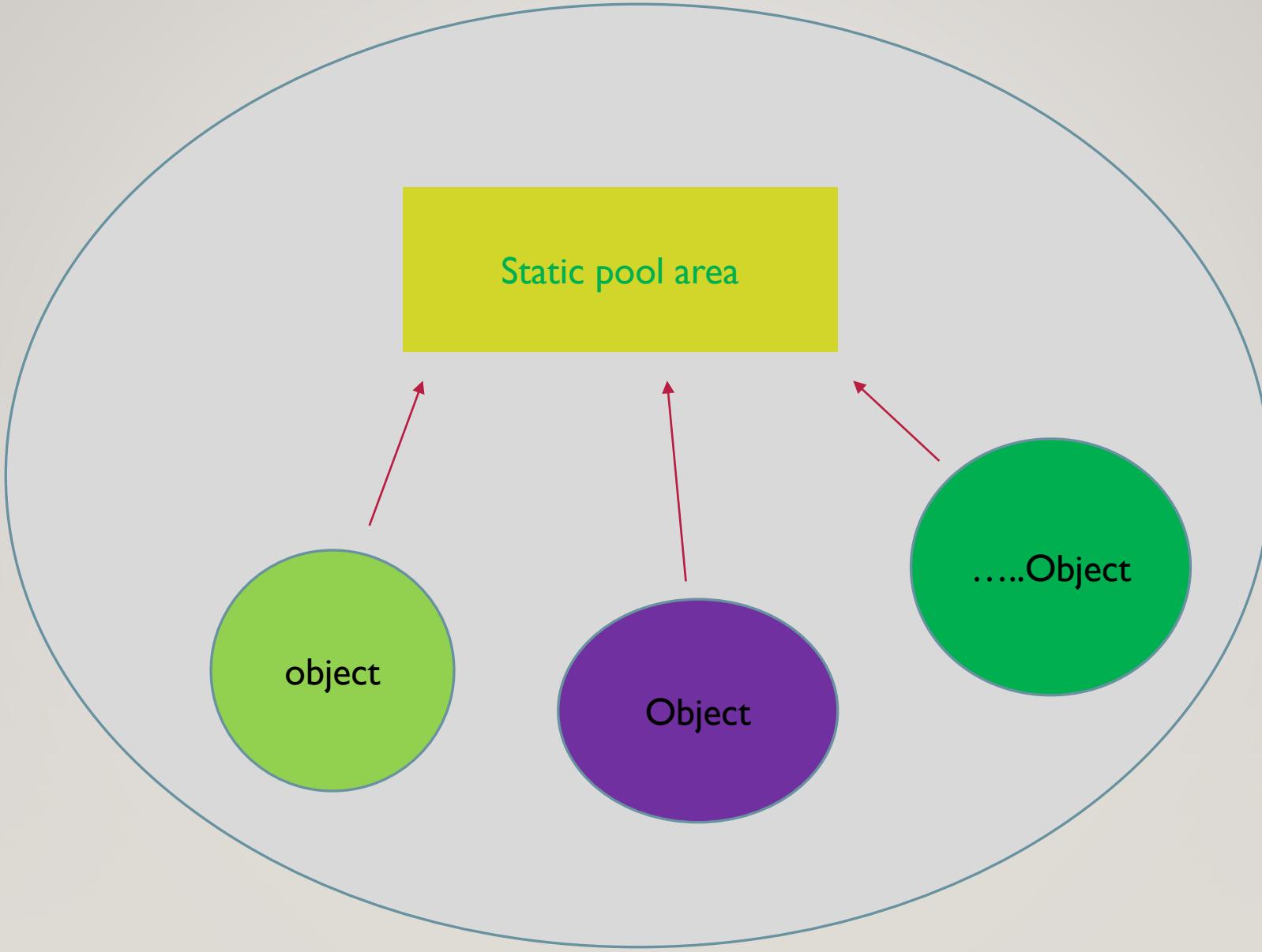
## #Execution process:

```
class Test {
    public static void main(String[] args) {
        System.out.println("1");
        staticMethod();
        Test t=new Test();
        t.nonStaticMethod();
        System.out.println("2");
    }

    public static void staticMethod() {
        System.out.println("3");
    }

    public void nonStaticMethod() {
        System.out.println("4");
    }
}
```





## Conclusion for execution flow

- When a Java program is executed, two memory areas are created:
  - **Stack Area**
  - **Heap Area**
- Stack Area is used for execution purposes.
- Heap Area is used for storage purposes.
- JVM makes use to call resources:
  - **Class loader**
  - **Main method**
  - **Garbage collector**
- Every operation needs to be executed in the stack area.
- Class loader is responsible to create the static pool area and load all the static methods and variable into the static pool area.
- For each class, class loader will come and load the classes once.
- The new operator is responsible to create objects and load all the non static method and variable to the heap area.
- Object address is assign to the object reference variable.
- After execution all the heap area clean by the garbage collector.

Calling method Or variable From I method to other :

Caller Method	Called Method	With In Same Class	With In The Different Class
static	static	Direct	By class name
static	No-static	By object reference variable Name	By object reference variable
No-static	static	direct	By class name
No-static	No-static	direct	By object reference variable

**Variables are classified into three types:**

**1. Global Variable( Instance Variable / class variable )**

- Static Variable:
- Non-Static Variable:

**2. Local Variable**

- Local Variable:

**Static Variable:**

- Declared with the static keyword and class level not inside method.
- It can be accessed directly with in the class and variable initialization is not mandatory we have default value.

**Non-Static Variable:**

- Declared without the static keyword and class level not inside method.
- Each object has its own copy of the non-static variable and variable initialization is not mandatory we have default value

**Syntax** for declaring static and non-static variables:

AccessSpecifier AccessModifier dataType variableName = data;

Exa : `public static long ifsc=SBI12345678;`

`public int accountNumber=12345678912;`

### Local Variable:

- it is declare with in the method and it is not access out of the method.
- Local variable don't have default value and it must initialize by developer.

### Syntax Local variable :

`dataType variableName = data;`

Exa : `int num=10;`

### Static and non static variable default value :

Data Type	Size (bits)	Default Value
byte	8	0
short	16	0
int	32	0
long	64	0L
float	32	0.0f
double	64	0.0d
char	16	( ) empty space
boolean	1	FALSE

default value  
for non  
primitive data  
type :



Data Type	Default Value
String	null
Array	null
Class	null
Interface	null

## **Method Overloading (Imp) :**

Creating multiple methods with the same **method name** but **different parameters** is called method overloading, when we want to perform same operation in multiple why then we are going with method overloading.

- Exa: real time example:-**
- 1) mobile will open with password, pattern, fingerprint.
  - 2) Multiple collages affiliated with single university.

**Benefits:- Code Reusability, Improved Code Organization, Enhanced Flexibility.**

```
public class MethodOverloadingExample {  
    public static void sum(int a, int b) {  
        System.out.println("Sum of two integers: " + (a + b));  
    }  
    public static void sum(double a, double b) {  
        System.out.println("Sum of two doubles: " + (a + b));  
    }  
    public static void sum(int a, int b, int c) {  
        System.out.println("Sum of three integers: " + (a + b + c));  
    }  
    public static void main(String[] args) {  
        sum(10, 20); // Calls the first sum method  
        sum(10.5, 20.5); // Calls the second sum method  
        sum(10, 20, 30); // Calls the third sum method  
    }  
}
```

## **Encapsulation(Imp) :**

binding data members (variables and methods) into a single unit (class) is called encapsulation.

### **Steps of Encapsulation:**

1. Declare variables as private.
2. Provide access to the variables through public methods.
3. Provide validation conditions inside the methods.

**Exa :** Real Time Exm:- Remote control, mobile.

**Benifites:-** Improved Code Organization, data security

```
class EncapsulationExample {  
    private int age; // Private variable  
    public int getAge() { // Public getter method  
        return age;  
    }  
}
```

```
public void setAge(int age) { // Public setter method  
    if (age >= 0) {  
        this.age = age;  
    } else {  
        System.out.println("Invalid age");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        EncapsulationExample person = new  
        EncapsulationExample();  
        person.setAge(25); // Set the age through the setter  
        int age = person.getAge();  
        System.out.println("Age: " + age);  
    }  
}
```

## **Constructor (Imp) :**

- A constructor is a special method that is executed during object creation.
- The constructor's name must be the same as the class name.
- Constructors don't have return type.
- Constructors are by default non-static. Developers should not use the static keyword with constructors.

## **Syntax of a constructor:**

```
accessSpecifier ConstructorName(arguments/non- arguments) {
```

```
    // Constructor body
```

```
}
```

- If a developer doesn't create any constructor, then the compiler will create a default constructor.
- A constructor without arguments and with an empty body is called a default constructor.

### **➤ Constructors are classified into two types:**

- **Constructor with arguments**
- **Constructor without arguments**

- Constructors are used for performing critical operations.
- One of the common critical operations in every program is the initialization of non-static variables.
- Creating multiple constructors with different argument data types is called constructor overloading.
- Calling one constructor from another constructor is called constructor chaining.
- A constructor cannot call another constructor by using its name.

- To call another constructor, we must use the this() statement.
- To call another constructor on other class we use super() statement.
- The this() statement informs the JVM to call the appropriate constructor based on the argument data type.
- Syntax of the this() statement:

**this(arg1, arg2, ...);**

- A constructor can chain with only one (single) constructor.
- To avoid breaking the above rule, the this() statement must be the first statement in the constructor.

**Exa :**

```
class Box {  
    int width, height, depth;  
    // Default constructor  
    Box() {  
        width = height = depth = 5;  
    }  
    // Parameterized constructor  
    Box(int w, int h, int d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```



## **Exa: Argumenta, non argumenta constructor and constructor overloading.**

```
class Car {  
    private String model;  
    private int year;  
    // Non-argumented constructor (default constructor)  
    public Car() {  
        model = "TATA";  
        year = 2024;  
    }  
    // Argumented constructor  
    public Car(String model, int year) {  
        this.model = model;  
        this.year = year;  
    }  
    public void displayInfo() {  
        System.out.println("Model: " + model);  
        System.out.println("Year: " + year);  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        // Creating a car object using the default  
        // constructor  
        Car car1 = new Car();  
        car1.displayInfo();  
        // Creating a car object using the  
        // parameterized constructor  
        Car car2 = new Car("Toyota Camry", 2023);  
        car2.displayInfo();  
    }  
}
```

## Exa: Constructor chaining by using super() and this()

```
class Vehicle {  
    protected String brand;  
    protected int year;  
    public Vehicle(String brand, int year) {  
        this.brand = brand;  
        this.year = year;  
    }  
}  
  
class Car extends Vehicle {  
    private String model;  
    public Car(String brand, int year, String model) {  
        super(brand, year); // Call the parent class's constructor  
        this.model = model;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car("Toyota", 2023, "Camry");  
    }  
}
```

## Inheritance(Imp) :

- It is a concept which is used to create child class based on the parent class or existing class. by using extend keyword.
- all the parent class separate copy of property will be shared with each sub class.
- Static component can not have separate copy, it will be shared with each sub class.

**Use:** for code reusability, Code organization, fast performance

**Real Time exm:** employee hierarchy, Vehicle hierarchy, Animal hierarchy.

**Exa:**

```
class Animal {  
    public void eat() {  
        System.out.println("Animal is eating");  
    }  
}
```

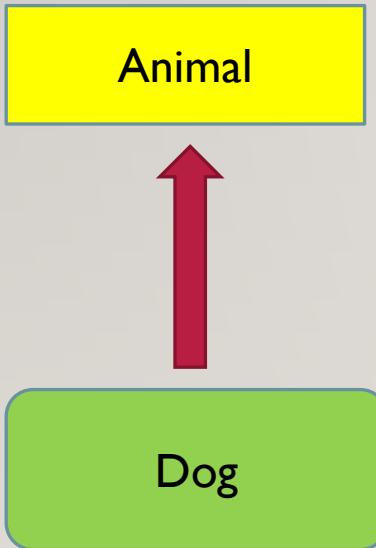
```
class Dog extends Animal {  
    public void bark() {  
        System.out.println("Dog is barking");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Dog dog = new Dog();  
        dog.eat(); // Inherited from Animal  
        dog.bark();  
    }  
}
```

## Types of inheritance :

1. Single level inheritance
2. Multi level inheritance
3. Hierarchical inheritance
4. Multiple inheritance

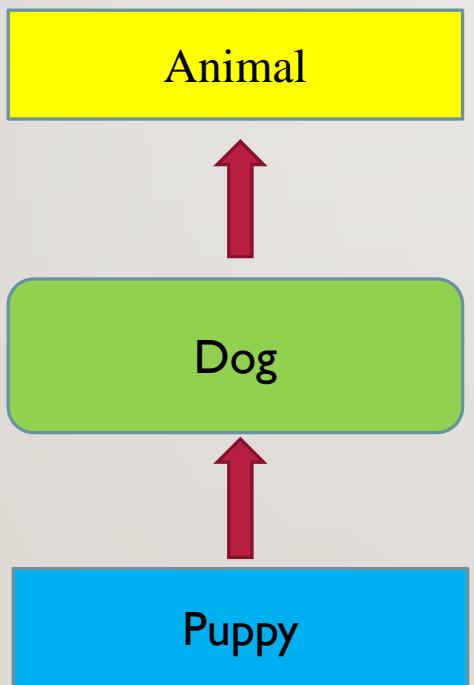
## 1) Single inheritance

```
class Animal {}  
class Dog extends  
Animal {}
```



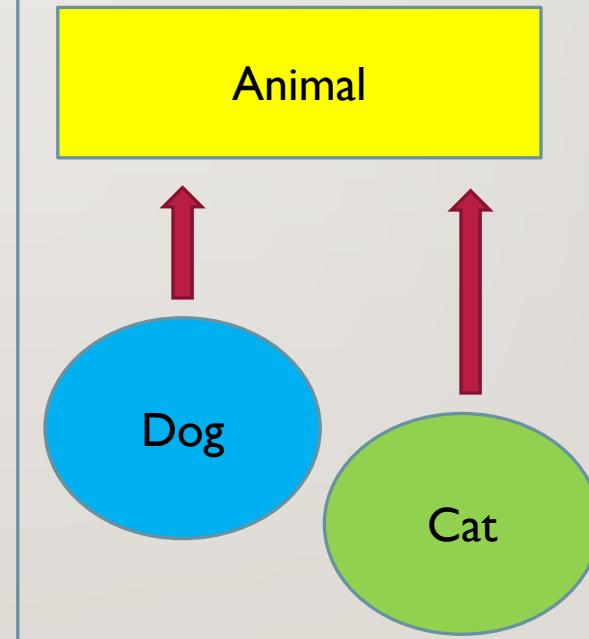
## 2) Multilevel inheritance

```
class Animal {}  
class Dog extends Animal {}  
class Puppy extends Dog {}
```



## 3) Hierarchical inheritance

```
class Animal {}  
class Dog extends Animal {}  
class Cat extends Animal {}
```



## 4) Multiple inheritance

single subclass having multiple upper class is called Multiple inheritance

Multiple inheritance is not possible in java class.  
Because of Dimond and ambiguity problem.

## Ambiguity problem :

- it is a constructor chaining problem, if we have 1 subclass and multiple super class then we will try to call super class default or same argument constructor then it will give ambiguity error.

```
Class A{  
    public A(){  
        //code  
    }  
}
```

```
Class B{  
    public B(){  
        //code  
    }  
}
```

```
Class C extends A, B{  
    public A(){  
        super(); //ambiguity  
    }  
}
```

## Diamond problem :

If we have 1 sub class and multiple super class and super class having same method and we try to call from sub class to the super class method then we will get ambiguity problem.

```
Class A{  
    public static walk(){  
        //code  
    }  
}
```

```
Class B{  
    public static walk(){  
        //code  
    }  
}
```

```
Class C extends A, B{  
    supper.walk();  
}
```

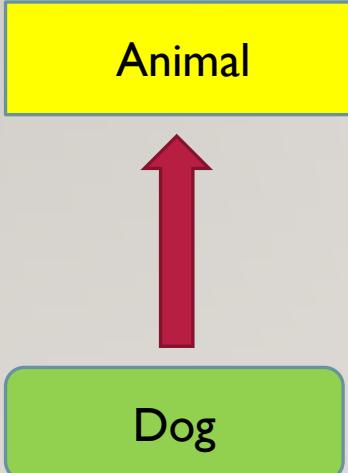
## **Interface (Imp) :**

- As soon as interface is created, new data type is created. This data type is called a user-defined data type or derived data type non-primitive data types.
- As soon as interface are created, new blueprints are created.
- In interface all the methods are by default public and abstract.
- In interface all the variables are by default final and static.
- Class extends class
- Class extends class and implements interface
- Class implements multiple interfaces
- Interface extends interface and Interface extends multiple interfaces
- Multiple inheritance is achievable using interface because there is no diamond problem because:
  - Interface does not allow constructors.
  - Have no ambiguity in constructor chaining.

## Types Of Inheritance by using interface :

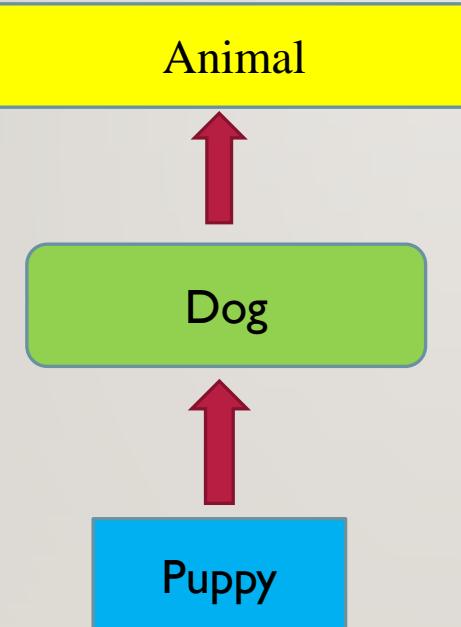
### 1) Single inheritance

```
interface Animal {}  
class Dog extends Animal {}
```



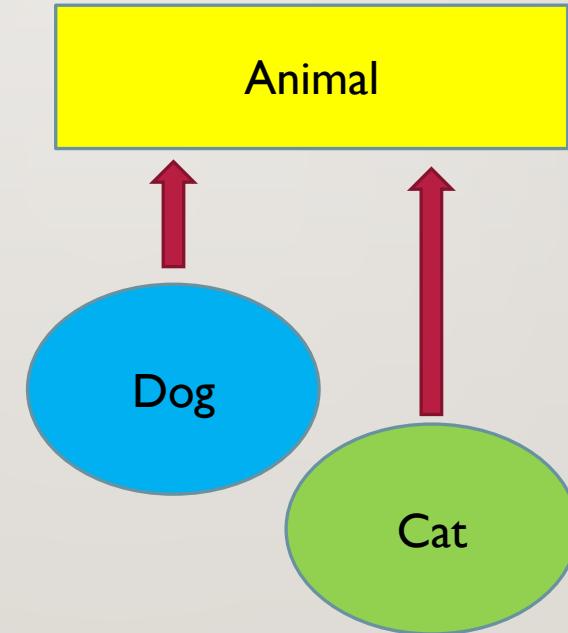
### 2) Multilevel inheritance

```
interface Animal {}  
interface Dog extends Animal {}  
interface Puppy extends Dog {}
```



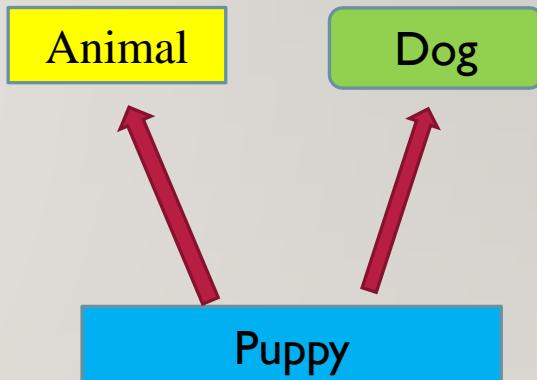
### 3) Hierarchical inheritance

```
interface Animal {}  
interface Dog extends Animal {}  
interface Cat extends Animal {}
```



### 4) Multiple inheritance

```
interface Animal {}  
interface Dog extends Animal {}  
interface Puppy extends Dog, Animal {}
```



### Ambiguity problem :

- In interface does not allow constructor so there is no ambiguity problem.

### Diamond problem :

In interface only method signature is there , there is no method implementation so we don't have diamond problem.

```
interface A{  
    public static walk();  
}
```

```
interface B{  
    public static walk();  
}
```

```
interface C extends A, B{  
    public static walk(){  
        //code  
    }  
}
```

## I) Single-Level Inheritance

```
interface Animal {  
    void eat();  
}  
  
class Dog implements Animal {  
    public void eat() {  
        System.out.println("Dog is eating");  
    }  
}
```

## 2) Multiple-Level Inheritance:

```
interface Pet {  
    void pet();  
}  
  
interface Animal extends Pet {  
    void eat();  
}  
  
class Cat implements Animal {  
    public void eat() {  
        System.out.println("Cat is eating");  
    }  
  
    public void pet() {  
        System.out.println("Petting the cat");  
    }  
}
```

### 3) Multiple Inheritance:

```
interface Flyable {  
    void fly();  
}  
  
interface Swimmable {  
    void swim();  
}  
  
class Duck implements Flyable, Swimmable {  
    public void fly() {  
        System.out.println("Duck 1 is flying");  
    }  
  
    public void swim() {  
        System.out.println("Duck is swimming");  
    }  
}
```

### 4) Hierarchical Inheritance:

```
interface Vehicle {  
    void move();  
}  
  
interface Car extends Vehicle {  
    void drive();  
}  
  
interface Bike extends Vehicle {  
    void ride();  
}  
  
class Sedan implements Car {  
    public void move() {  
        System.out.println("Sedan is moving");  
    }  
  
    public void drive() {  
        System.out.println("Driving a sedan");  
    }  
}
```

Diff b/w class and interface ?

Feature	Interface	Class
<b>Definition</b>	A blueprint of a class. It defines a set of methods that a class must implement.	A blueprint of an object. It defines the properties and behaviors of an object.
<b>Keyword</b>	interface	class
<b>Inheritance</b>	Multiple inheritance is possible.	Single inheritance is possible.
<b>Methods</b>	All methods are abstract.	Can have both abstract and concrete methods.
<b>Variables</b>	All variables are public, static, and final (constants).	Can have variables of different types.
<b>Instantiation</b>	Cannot be instantiated directly.	Can be instantiated to create objects.

## **Is-A Relationship (Inheritance) (Imp) :**

This relationship represents a hierarchical relationship between classes. A subclass "is a" type of its superclass.

**example:** Dog is an Animal.

```
class Animal {  
}  
class Dog extends Animal {  
}
```

## **Has-A Relationship (Composition/Aggregation) (Imp) :**

This relationship represents a "whole-part" relationship between classes. A class "has a" reference to another class.

**example: car has a engine**

```
class Car {  
    private Engine engine=new Engine();  
}  
class Engine {  
}
```

Feature	Is-A Relationship (Inheritance)	Has-A Relationship (Composition/Aggregation)
<b>Keyword</b>	extends	No specific keyword
<b>Relationship</b>	Hierarchical	Whole-part
<b>Access</b>	Inherited members can be accessed directly	Members of the composed object need to be accessed through its reference
<b>Lifetime</b>	Lifetime of the child object is tied to the parent object	Lifetime of the composed object is independent of the container object

## #Type Casting(Imp) :

type casting means converting one object type to look like another object type. Is called object type casting.

Object type casting is classified into two types:

### 1. Upcasting

- Converting a subclass object to look like a superclass object by hiding subclass properties and showing superclass properties. This process is called upcasting.
- Upcasting is possible because of subclass object is having superclass properties.

### 2. Down casting

- The process of converting an upcasted object to look like a subclass object by making subclass properties visible. This process is called down casting.
- Down casting is possible only after upcasting.
- Direct down casting is not possible. That is, we cannot directly convert a superclass object to look like a subclass object because a superclass object will not have subclass properties.

### Note :

Upcasting will have zero (0) effect on method overriding because an overridden method is a superclass property, and upcasting's job is only to hide subclass properties.

```

Exa :1 class Test {
    public static void main(String[] args) {
        // Upcasting
        Animal animal = new Dog(); // Dog object is assigned
        to Animal reference
        animal.eat(); // Output: Animal is eating
        // Downcasting (with type checking)
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal; // Safe downcasting
            dog.bark(); // Output: Dog is barking
        } else {
            System.out.println("Cannot downcast to Dog");
        }
    }

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

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

```

```

Exa: 2
class Shape {
    public void draw() {
        System.out.println("Drawing a generic shape");
    }
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle(); // Upcasting: Circle object to
        Shape reference
        shape.draw(); // Output: Drawing a circle
    }
}

```

## Method Overriding

Multiple method with same signature but implementation can be different or same is called method Overriding.

Note point :

- Inheritance is mandatory.
- A subclass can change the method implementation of inherited methods.
- Changing the method implementation of inherited methods is called method overriding.
- In order to override an inherited method, the subclass should maintain the same method signature given by the superclass.
- Overriding is not a mandatory process rather, it depends on project requirements.
- Only non-static methods can be overridden because of multiple copy nature.
- Static methods cannot be overridden because they do not support inheritance.
- Method overriding will only affect the current subclass because overriding is happening on the separate copy of the method given by the superclass.

### Exa :1

```
class Animal {  
    public void makeSound() {  
        System.out.println("Generic animal sound");  
    }  
  
    class Dog extends Animal {  
        @Override  
        public void makeSound() {  
            System.out.println("Woof!");  
        }  
  
    public class Main {  
        public static void main(String[] args) {  
            Animal animal = new Dog();  
            animal.makeSound(); // Output: Woof!  
        }  
    }
```

### Exm :2

```
class Shape {  
    public void draw() {  
        System.out.println("Drawing a generic shape");  
    }  
    class Circle extends Shape {  
        @Override  
        public void draw() {  
            System.out.println("Drawing a circle");  
        }  
        class Square extends  
        Shape {  
            @Override  
            public void draw() {  
                System.out.println("Drawing a square");  
            }  
            public class Main {  
                public static void main(String[] args) {  
                    Shape shape1 = new Circle();  
                    Shape shape2 = new Square();  
                    shape1.draw(); // Output: Drawing a circle  
                    shape2.draw(); // Output: Drawing a square  
                }  
            }
```

## Difference b/w method overloading and method overriding?

Feature	Method Overloading	Method Overriding
Definition	Declaring multiple methods with the same name but different parameters of argument	Declaring a method with the same method signature and implementation can be different.
Parameter List	Different	Same
Return Type	Can be different	Must be the same
Access Modifiers	Can be different	Must be the same
Compiler/Runtime Resolution	Compile-time	Runtime
Inheritance	Not necessary	necessary
Purpose	To perform same operation based on different input.	To provide a specific operation with different implementation.,
Read time example	Making drink, making rise, mobile lock open in multiple way.	Every car has a engine.

## The final Keyword in Java

The final keyword in Java is used to declare java component that cannot be modified once they are initialized. It can be applied to variables, methods, and classes.

### 1. Final Variables:

#### ➤ Constant Variable

```
final double PI = 3.14159;
```

#### ➤ Final Classes:

#### Preventing Inheritance:

```
final class ImmutableClass { // code  
}
```

//Error: Cannot inherit from a final class

```
class DerivedClass extends ImmutableClass {  
//code  
}
```

#### ➤ Final Methods:

#### ➤ Preventing Overriding:

```
class Parent {  
    public final void method() {  
        // code  
    }  
}
```

```
class Child extends Parent {  
    // Cannot override the final method  
    public void method() {  
        // code  
    }  
}
```

#### ➤ Class-Level Constants:

```
public class MyClass {  
    public static final int MAX_VALUE = 100;  
}
```

## # Method Binding

- Method is divided into two parts:
  - 1) **Method signature**
  - 2) **Method implementation**
- Process of connecting method signature with method implementation is called method binding.
- During coding stage, developer is responsible only for creating method signature and method implementation.
- Methods are not bound during coding stage, rather methods are created during coding stage.
- Methods are bound either during compilation stage or execution stage.
- If methods are bound during compilation stage, then this binding is called compile-time binding.
- If methods are bound during execution stage, then this binding is called run-time binding.
- Compile-time binding is also known as static binding, because binding between method signature and method implementation can not be changed.
- **Run-time Binding** is also known as **dynamic binding or late binding** because the binding can be changed from one implementation to another implementation based on object creation.
- **Compile-time Binding** is also known as **early binding**, because methods are already bound before execution starts and methods are ready for execution at any point of time.

Syntext:

Access Specifier	Access Modifier	Return Type	MethodName() //method signature
{ //block of code }			//Method implementation

Exa:

```
public static void methodName(arg/non-arg) //method signature
{
    //block of code
}
```

//Method implementation

**Note: Main Method is static for two reasons:**

1. Static methods are loaded first before non-static methods during compilation time.
2. Static methods are bound first (early binding) before non-static methods are bound.

**Access Level According to access modifier :**

Access Specifier	Within the Class	Within the Package	Subclasses (Same Package)	Subclasses (Different Package)	Outside the Package
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default (no modifier)	Yes	Yes	No	No	No
private	Yes	No	No	No	No

## Abstract (Imp) :

- Methods are classified into two types:
  - **Concrete methods or complete method**
  - **In-complete methods**
- Method which has method signature as well as method implementation is called concrete method.
- Method which has only method signature is called incomplete method or abstract method.

## Steps to create abstract method:

- Step 1: Create only method signature.
- Step 2: end with semi-colon.
- Step 3: Declare method as abstract.
- Step 4: Declare class as abstract.

Exa : abstract class Shape {  
    public abstract void draw();  
}

## Steps to use abstract method:

- Step 1: Create a sub-class to inherit in-complete method.
- Step 2: Complete the method or implement the method or override the method.

- If class contain even single abstract method, declare class as abstract.
- If subclass does not complete all the super class abstract methods, subclass is also abstract class.
- Declare super class as abstract class because to block super class object creation.
- Classes are declared abstract in 3 cases:

- Case 1: If class contain incomplete method, class is declared as abstract.
- Case 2: To block object creation.
- Case 3: If class contain all static members.

### Exa :1

```
abstract class Shape {  
    public abstract void draw();  
    public void erase() {  
        System.out.println("Erasing shape");  
    }  
}  
  
abstract class Circle extends Shape {  
    // Abstract method to be implemented by concrete subclasses  
    public abstract void calculateArea();  
    @Override  
    public void draw() {  
        System.out.println("Drawing a circle");  
    }  
}
```

```
class ColoredCircle extends Circle {  
    @Override  
    public void calculateArea() {  
        // Implementation for calculating area of a circle  
    }  
}
```

## # Specialization

Process of creating a method which can take single objects. This process is called specialization and the method is called specialized method.

### → Steps to create Specialized method:

- Create specialized method.
- Create Static or non-static methods with arguments.
- Argument should be user define data type followed by object reference variable.
- Define repetitive code within specialized method.
- Call the method and pass object as an input to specialized method.

### Note:

- User define data type and object type should be same.

## # Generalization

Process of creating a method which can take multiple objects of different types and have a common superclass. This process is called Generalization and the method is called generalized method.

### → Steps to create Generalized method:

- Create static or non-static method with arguments.
- Argument should be user-defined data type followed by object reference variable.
- Define repetitive code within generalized method.
- Call the method and pass an object as an input to generalized method.
- The passed object should be upcasted.
- User-defined data type should be superclass data type, whereas passed object should be subclass type.
- Finally we concluded, by using specialization or generalization, we can minimize code repetition and implement code reusability.

## Exa : Generalization :

```
class Shape {  
    public void draw() {  
        System.out.println("Drawing a shape");  
    } }  
  
class Circle extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Drawing a circle");  
    } }  
  
class Square extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Drawing a square");  
    } }
```

```
public class Main {  
  
    public static void drawShape(Shape shape) {  
        shape.draw();  
    }  
  
    public static void main(String[] args) {  
        Circle circle = new Circle();  
        Square square = new Square();  
        drawShape(circle);  
        drawShape(square);  
    }  
}
```

## Exa : Specialization:

```
class Animal {  
    public void makeSound() {  
        System.out.println("Generic animal sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof!");  
    }  
}  
  
public class Main {  
    public static void callDog(Dog dog) {  
        System.out.println("Meow!");  
    }  
}
```

```
public static void main(String[] args) {  
    Dog dog1 = new Dog();  
    Dog dog2 = new Dog();  
    dog1.makeSound();  
    dog2.makeSound();  
}
```

## # Abstraction(imp) :

- In simple words, abstraction means hiding unnecessary detail and providing access only necessary detail.
- In technical terms, abstraction means hiding method implementation detail and providing access only method signature.

**Abstraction is achieved in 2 ways: By using abstract class and by using interface.**

## Steps to design Abstraction:

- Step 1:** Create an interface / abstract class.
- Step 2:** Create a method in interface / abstract class.
- Step 3:** Create implementation class / sub class.
- Step 4:** Implement the method / Override the method.
- Step 5:** Create a helper class.
- Step 6:** Create helper methods.
- Step 7:** Helper method will create object of implementation class / sub class.

**Step 8:** Implementation class object is upcasted to interface / superclass.

**Step 9:** Upcasted object is returned.

**Step 10:** Call the helper method.

- If the helper method is static, call the helper method by using helper class name.
- If the helper method is non-static, create helper class object, call the helper method by using helper class object.

**Step 11:** Helper method will return implementation class object in upcasted manner.

**Step 12:** Receive the object by using interface datatype followed by object reference variable.

**Step 13:** Use interface object reference variable to call interface methods.

### **Exm : 1**

```
interface Shape {  
    void draw();  
}  
  
class Circle implements Shape {  
    @Override  
    public void draw() {  
        System.out.println("Drawing a circle");  
    }  
}  
  
class HelperClass {  
    public static Shape getShapeObject() {  
        Shape shape=new Circle();  
        return shape;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Shape shape = HelperClass.getShapeObject();  
        shape.draw(); // Output: Drawing a circle  
    }  
}
```

### **Use Case :**

1. Security
2. Code Reusability
3. Making Code organized

### **Real-World Examples of Abstraction:**

1. Remote Control
2. Car Driving
3. Software Libraries

## # Polymorphism

- Polymorphism means single entity having multiple forms.
- Polymorphism classified into two types:
  1. **Compile-time polymorphism**
  2. **Run-time polymorphism**

### ➤ Run-time polymorphism

- Call to overridden method is decided during runtime based on the object creation. This is called runtime polymorphism.
- To achieve runtime polymorphism, we need to use:
  1. **Interface/Superclass/abstract class**
  2. **Implementation class/subclass**
  3. **Method overriding.**
  4. **Generalization**
  5. **Upcasting**

### ➤ Compile-time polymorphism

- Call to overloaded method is decided during compilation time based on the argument list. This is called compile-time polymorphism.
- In order to achieve compile-time polymorphism, we need to use:
  1. **Static methods**
  2. **Method overloading**

### Real time Exa:

Shape Interface, Animal Sounds, Remote Control.

### Use Cases:

Method Overriding, Method Overloading, Interface-Based Programming

### Benefits :

Code Reusability, Flexibility, Maintainability

## Exa : Run time polymorphism

```
class Animal {  
    public void makeSound() {  
        System.out.println("Generic animal sound");  
    }  
}  
class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof!");  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Animal animal1 = new Dog();  
        animal1.makeSound(); // Output: Woof!  
    }  
}
```

## Exa : Compile time polymorphism

```
class Calculator {  
    public static int add(int a, int b) {  
        return a + b;  
    }  
    public static double add(double a, double b) {  
        return a + b;  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        int sum1 = Calculator.add(2, 3);  
        double sum2 = Calculator.add(2.5, 3.5);  
    }  
}
```

## instanceof keyword :

The instanceof keyword in Java is used to check if an object is belong to particular class or interface or not, ensuring type safety during down casting.

### Syntax :

**boolean result = object instanceof ClassName;**

### Exa : 1

```
class Animal {  
    public void makeSound() {  
        System.out.println("Generic animal sound");  
    }  
}  
  
class Dog extends Animal {  
    public void bark() {  
        System.out.println("Woof!");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Animal animal = new Dog();  
  
        if (animal instanceof Dog) {  
            Dog dog = (Dog) animal;  
            dog.bark(); // This will print "Woof!"  
        } else {  
            System.out.println("Animal is not a Dog");  
        }  
    }  
}
```

### Exa : 2

Boolean **isString="Hii" instanceof String;**

# Course : Java

## Part : Java Library

## #Lambda

- **Lambda expression** is an anonymous function that doesn't have a name, modifiers, or return type.
- Lambda expressions work only with functional interfaces, which are interfaces that contain only one abstract method.
- By using lambda expressions, we can convert Java from an object-oriented language to a functional programming language. Functional programming means any operation doesn't need to be written in an elaborate way. By using lambda expressions, we can simplify the code. This simplification process is known as functional programming.
- Lambda expressions were introduced in the year 2014 and in the JDK version of 1.8.

### Syntax:

(parameter list) -> lambda body

## Exa :1

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class LambdaExample {
    public static void main(String[] args)
    {
        List<String> names = Arrays.asList("Alice", "Bob",
"Charlie",
"David");
        // Sort the list using a lambda expression
        Collections.sort(names, (a, b) ->
a.compareToIgnoreCase(b));
        // Print the sorted list
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

## #Object Class

- **Object class** is an inbuilt class present in the `java.lang` package.
- **Object class** is the super most class present in Java.
- **Object class** contains common methods which will be applicable in any type of object, so that the name they have given it is Object.
- In between our class and Object class, there is an implicit inheritance.
- When a developer needs to perform inheritance, the compiler itself will be performing the inheritance.
- Object class contains **11 common methods** which will be applicable in all the classes present in Java.
- All the object class methods are **non-static methods**.
- Object class properties follow **1st method** as:

- While performing inheritance, we have to create an object of the subclass. By using the subclass object reference variable, we are going to access object class properties.

### Object class methods:

1. `toString(): String`
2. `equals(Object obj): boolean`
3. `hashCode(): int`
4. `getClass(): Class<?>`
5. `clone(): Object`
6. `wait(): void`
7. `wait(long timeout): void`
8. `wait(long timeout, int nanos): void`
9. `notify(): void`
10. `notifyAll(): void`
11. `finalize(): void`

## **toString(): String**

- It is non static method, in entire java `toString()` will be executed without calling, that's why we are calling this special method.
- Example: `System.out.println(new Object().toString());`

## **equals(Object obj): boolean**

- Compares this object with the specified object for equality.
- Example: `String s1 = "hello"; String s2 = "hello"; System.out.println(s1.equals(s2));`

## **hashCode(): int**

- Returns a hash code value for the object.
- Example: `Integer i = new Integer(10); System.out.println(i.hashCode());`

## **getClass(): Class<?>**

- Returns the Class object associated with this object.
- Example: `String s = "hello"; System.out.println(s.getClass());`

## **clone(): Object**

- Creates and returns a copy of this object.
- Example: `Object obj = new Object(); Object cloneObj = obj.clone();`

## **wait(): void**

- Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
- Example: Used in inter-thread communication.

## **wait(long timeout): void**

- Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified timeout occurs.
- Example: Used in inter-thread communication with a timeout.

## **wait(long timeout, int nanos): void**

- Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified timeout occurs, with nanosecond precision.
- Example: Used in inter-thread communication with a precise timeout.

### **notify(): void**

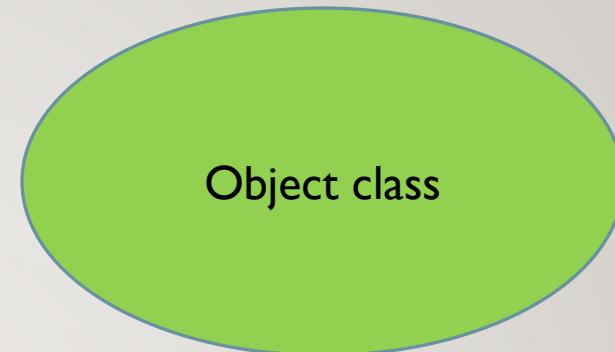
- Wakes up a single thread that is waiting on this object's monitor.
- **Example:** Used in inter-thread communication to signal a waiting thread.

### **notifyAll(): void**

- Wakes up all threads that are waiting on this object's monitor.
- **Example:** Used in inter-thread communication to signal all waiting threads.

### **finalize(): void**

- Called by the garbage collector on an object before it is garbage collected.
- **Example:** Can be overridden to perform clean-up tasks, but it's generally discouraged.



Predefine classes

Inbuild classes

## Why Strings are Immutable in Java

Strings are immutable, meaning their values cannot be changed once they are declared. This is called immutability, it offers several advantages.

### **□ Security:**

Prevents unintended modification of sensitive data (e.g., passwords) stored as Strings.

### **□ Efficiency:**

Enables String pooling, where identical String literals share the same memory location, reducing memory footprint.

### **□ Thread Safety:**

Eliminates the need for explicit synchronization when multiple threads access the same String, as its value is guaranteed not to change.

### **□ Hashing:**

Ensures consistent hash codes, crucial for efficient use in hash-based collections (HashMap, HashSet).

•

## Example:

```
String str1 = "Hello";
```

```
String str2 = str1.concat(" World");
```

### **Length :**

- String implicitly implemented character array and char array length is final.

## #Wrapper Class

### #What is wrapper class?

- The class which will wrap the primitive data. Such classes we can call as wrapper class.

### #Before wrapper class

- We can not conclude Java as 100% object-oriented language because we cannot create the object of predefine data.

### #After wrapper class

- We can conclude Java as 100% object-oriented language. Because not only for class we can create object even for predefine data also we can create the objects, by using wrapper classes.
- Each and every predefine datatype it having its own corresponding wrapper class.

## Primitive data type corresponding wrapper class

Data Type	Wrapper Classes
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

## #Boxing

- The process of converting predefined data into Object is known as Boxing.
- Boxing can be done in 2 ways:
  - Explicit Boxing
  - Implicit Boxing

### Explicit Boxing:

By using the new operator and constructors calling, if we do this type of boxing, such type is known as Explicit boxing.

```
package wrapper_class;  
public class Explicit_Boxing_UnBoxing {  
    public static void main(String[] args) {  
        int intData = 10;  
        Integer intObject = new Integer(intData);  
    }  
}
```

## Implicit Boxing and Auto Boxing

- From JDK versions of 1.0 to 1.4, developers can perform only Explicit Boxing and unboxing but from JDK 1.5 developers can perform explicit and at the same time, he can also perform implicit boxing and unboxing.
- Implicit boxing is also known as auto boxing.

### Exa :

```
Integer intObject = 10; // Implicit Boxing
```

### Unboxing

- The process of converting again back from object to predefined data. Is known as unboxing.
- Unboxing can be done in 2 ways:
  - Explicit Unboxing
  - Implicit Unboxing

## Implicit Unboxing/Auto Unboxing

- From JDK 1.0 to 1.4, developers can perform only explicit unboxing.
- From JDK 1.5, developers can perform unboxing in both explicit and implicit ways.
- Implicit unboxing is also known as auto unboxing.

## #Explicit Unboxing

- To perform explicit unboxing, developer has to call value() method.
- This method will return corresponding primitive data.

```
int intValue = IntegerObject.intValue(); // Explicit Unboxing  
byte byteVal = ByteObject.byteValue(); // Explicit Unboxing  
  
package wrapper_class;  
public class Wrapper_Class_Explicit_Boxing_Unboxing {  
    public static void main(String[] args) {  
        int intData = 10;  
        Integer intObject = new Integer(intData); // Explicit Boxing  
        int intValue = intObject.intValue(); // Explicit Unboxing  
    }  
}
```

```
public class BoxingUnboxingExample {  
    public static void main(String[] args) {  
        // Implicit Boxing  
        int num = 10;  
        Integer integerObject = num; // Implicitly boxed to Integer  
        // Explicit Boxing  
        float f = 2.5f;  
        Float floatObject = new Float(f);  
        // Implicit Unboxing  
        int num1 = integerObject; // Implicitly unboxed to int  
        // Explicit Unboxing  
        char charValue = charObject.charValue();  
    }  
}
```

## Converting Strings to Primitive Data Types in Java

To convert a string to a primitive data type in Java, you typically use the parseXXX() methods provided by the respective wrapper classes. Here are some common examples:

```
public class StringToPrimitiveConversion {  
    public static void main(String[] args) {  
        // Conversion to Integer  
        String intStr = "123";  
        int intNum = Integer.parseInt(intStr);  
  
        // Conversion to Double  
        String doubleStr = "3.14";  
        double doubleNum = Double.parseDouble(doubleStr);  
  
        // Conversion to Boolean  
        String boolStr = "true";  
        boolean boolValue = Boolean.parseBoolean(boolStr);  
    }  
}
```

### // Conversion to Long

```
String longStr = "1234567890";  
long longNum = Long.parseLong(longStr);
```

### // Conversion to Float

```
String floatStr = "3.14f";  
float floatNum = Float.parseFloat(floatStr);
```

### // Conversion to Character

```
String charStr = "A";  
char charValue = charStr.charAt(0);  
}  
}
```

## Converting Primitive Data Types to Strings in Java

### 1. Using String.valueOf()

The `String.valueOf()` method is a way to convert various data types, including primitive data types, to the string.

```
int num = 42;  
String str = String.valueOf(num);
```

```
double pi = 3.14159;  
String piStr = String.valueOf(pi);
```

```
boolean isTrue = true;  
String boolStr = String.valueOf(isTrue);
```

```
char ch = 'A';  
String charStr = String.valueOf(ch);
```

### 2. Using `toString()` Method of Wrapper Classes

Each primitive data type has a corresponding wrapper class. These wrapper classes provide a `toString()` method to convert the primitive value to its string representation.

Exa :

```
int num = 42;  
String str = Integer.toString(num);
```

```
double pi = 3.14159;  
String piStr = Double.toString(pi);
```

```
boolean isTrue = true;  
String boolStr = Boolean.toString(isTrue);
```

```
char ch = 'A';  
String charStr = Character.toString(ch);
```

## # Third way of performing Boxing

- Boxing can also be done by using valueOf().
- Which is presenting corresponding wrapper classes and we have to access this method by using class name.

**Ex:**

1. Character charObject = Character.valueOf('#');
2. Integer intObject = Integer.valueOf(10);
3. Byte byteObject = Byte.valueOf((byte) 10);
4. Short shortObject = Short.valueOf((short) 10);
5. Long longObject = Long.valueOf(1234567890L);
6. Float floatObject = Float.valueOf(10.5F);
7. Double doubleObject = Double.valueOf(10.5);
8. String stringObject = String.valueOf("text");

## #Exception(imp):

- Exception is an unexpected event faced by JVM inside Stack area.
- Once an exception occurred in our program, JVM will be immediately terminated.

## #Exception Handling

The process of handling the Exception is exception handling.

- If developer has given exception handling code, exception will be handled.
- If developer does not give exception handling code, JVM will be creating an exception object and with the property of:
  - Name, Description, Location
- With this property, JVM will be calling print stack trace method implicitly.
- print stack trace method will be using this exception property and this method will trace the stack area and what all happening in stack area will be printed.
- Exception can be handled by try and catch block

## ➤ try block

## ➤ catch block

- All the dangerous statements should be written inside **try block**.
- All the recovery statements should be written inside **catch block**.

## Syntax :

```
try {  
    // Code that might throw an exception  
} catch (ExceptionType1 exceptionObject1) {  
    // Handle ExceptionType1  
} catch (ExceptionType2 exceptionObject2) {  
    // Handle ExceptionType2  
} finally {  
    // Code that always executes, regardless of exceptions  
}
```

Exm :

```
public class ExceptionExample {  
    public static void main(String[] args) {  
        int[] numbers = {10, 20, 30};  
        try {  
            // This line might throw an  
            ArrayIndexOutOfBoundsException  
            int value = numbers[5]; // Index 5 is out of  
            bounds  
            System.out.println("Value: " + value);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array index out of bounds:  
            " + e.getMessage());  
        }  
    }  
}
```

a

## #Exception Propagation

- The process of moving on or shifting or propagating Exception from called method to caller method.
- In our program if we use propagate Exception object from called method to caller method, this process is known as Exception propagation.

## Exception propagation will done in two ways

1. Explicit propagation.
2. Implicit propagation.

## There are two types of exception

1. Checked Exception
2. Unchecked Exception

## Checked Exception

- If the statement is checked by compiler - those statements are known as checked statements.
- Checked exception will be getting from checked statements.

- What all the exception classes are inheriting exception super class, those exceptions are checked exception.
- Our program contains checked exception, but have to handle inside caller method.
- Examples of Checked Exception:
  - **IOException:** Occurs when an I/O operation fails, such as reading from or writing to a file.
  - **SQLException:** Occurs when a database operation fails.
  - **ClassNotFoundException:** Occurs when a class cannot be found.

## Unchecked Exception

- If the statement is not checked by compiler, those statements are known as unchecked statements.
- Unchecked statement will be getting from unchecked statements.

- what all the exception classes are inheriting run time exception super class, that will become unchecked statement exception.
- If our program contain unchecked statement we have to perform implicit propagation, implicit propagation can be done by using without throws keyword

### Note :

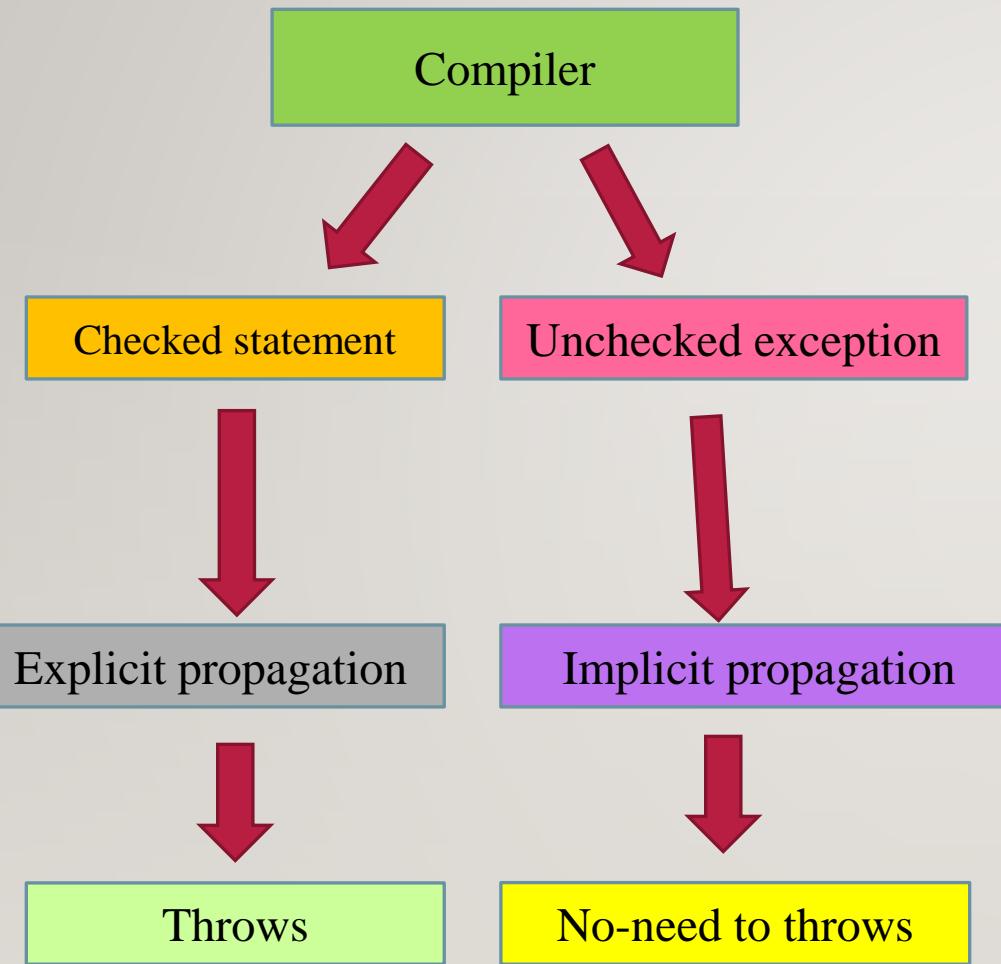
- If we are performing explicit propagation we have to make use of throws keyword followed by the exception type.

### Exa:

- **NullPointerException:** Occurs when trying to use an object reference that has not been initialized.
- **ArrayIndexOutOfBoundsException:** Occurs when trying to access an array element with an invalid index.
- **ArithmaticException:** Occurs when an arithmetic operation, such as division by zero, fails.

- **ClassCastException:** Occurs when trying to cast an object to an incompatible type.
- **IllegalArgumentException:** Occurs when a method is called with an illegal or invalid argument.
- **IllegalStateException:** Occurs when a method is called at an illegal or inappropriate time.
- **IndexOutOfBoundsException:** Occurs when trying to access an index that is out of bounds of a list or array.

## Exception Propagation



## #User Defined Exception

- An exception created from user side this is known as user defined exception.
- To create user defined exception, a developer can create a class and class should inherit exception super class or RuntimeException super classes.
- According to the exception which you want to throw the exception, by using the try-catch block we have to catch.

### How to Create User Defined Exception

#### 1. Create a Class

That class should inherit exception super class or RuntimeException super classes.

#### 1. Creating the exception

We can create the exception by using the throw keyword.

1. If in case our exception wants to become unchecked exception, we need to make use of RuntimeException.

Exa :

```
public class InsufficientFundsException extends Exception {  
    public InsufficientFundsException(String message) {  
        super(message);  
    }  
  
    public class Account {  
        private double balance;  
        public void withdraw(double amount) throws  
            InsufficientFundsException {  
            if (balance < amount) {  
                throw new InsufficientFundsException("Insufficient  
                balance");  
            }  
            balance -= amount;  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Account account = new Account();  
        account.balance = 100;  
        try {  
            account.withdraw(200);  
        } catch (InsufficientFundsException e) {  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

Diff b/w checked exception and unchecked exception?

Feature	Checked Exceptions	Unchecked Exceptions
Checked by Compiler	Yes	No
Subclass of	Exception	RuntimeException
Handling	Must be either caught or declared in the method signature using throws	Not required to be handled, but can be
Common Examples	IOException, SQLException, ClassNotFoundException	NullPointerException, ArrayIndexOutOfBoundsException, ArithmaticException
Origin	External factors like file I/O, network issues	Programming errors or unexpected conditions

## Diff b/w error and exception?

Feature	Errors	Exceptions
Definition	Severe issues that indicate system problems beyond the program's control.	Unexpected events that occur during program execution due to issues within the code.
Recoverability	Generally not recoverable	Can often be recovered from using try-catch blocks.
Checked/Unchecked	All errors are unchecked.	Can be both checked and unchecked.
Responsibility	System or environment	Program code
Package	java.lang.Error	java.lang.Exception
Examples	OutOfMemoryError, StackOverflowError, VirtualMachineError	NullPointerException, ArrayIndexOutOfBoundsException, IOException, SQLException

Diff b/w throe and throws keyword?

Feature	throw Keyword	throws Keyword
Purpose	Used to create an exception	Used to throw the exception from called method to caller method and it can be one or more exceptions
Exception Type	It can create only one exception at a time	It Can throws multiple exceptions
Usage	Primarily used for unchecked exceptions	Primarily used for checked exceptions
Placement	Can be used inside methods, constructors, or conditional blocks	Must be declared in the method signature
Syntax	throw new ExceptionObject();	methodName(param1, param2) throws Exception1, Exception2 { // ... }

## #Finally Block

- All the program input statement should be given inside the finally block.
- Finally block will be executed irrespective of exception occur or not.
  - 1) If exception occurs catch and finally will be executed.
  - 2) If exception does not occur try and finally block will be executed.

```
Exa : public class FinallyBlockExample {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // throwing ArithmeticException  
        } catch (ArithmaticException e) {  
            System.out.println("Arithmatic Exception: "+e.getMessage());  
        } finally {  
            System.out.println("This will always execute.");  
        }  
    }  
}
```

**Exception Information will be printed by using following 3 methods.**

**1. printStackTrace(); void**

**2. toString(); String**

**3. getMessage(); String**

**1. printStackTrace(): void**

- Prints a additional information about the exception like exception name, description and location of the exception.

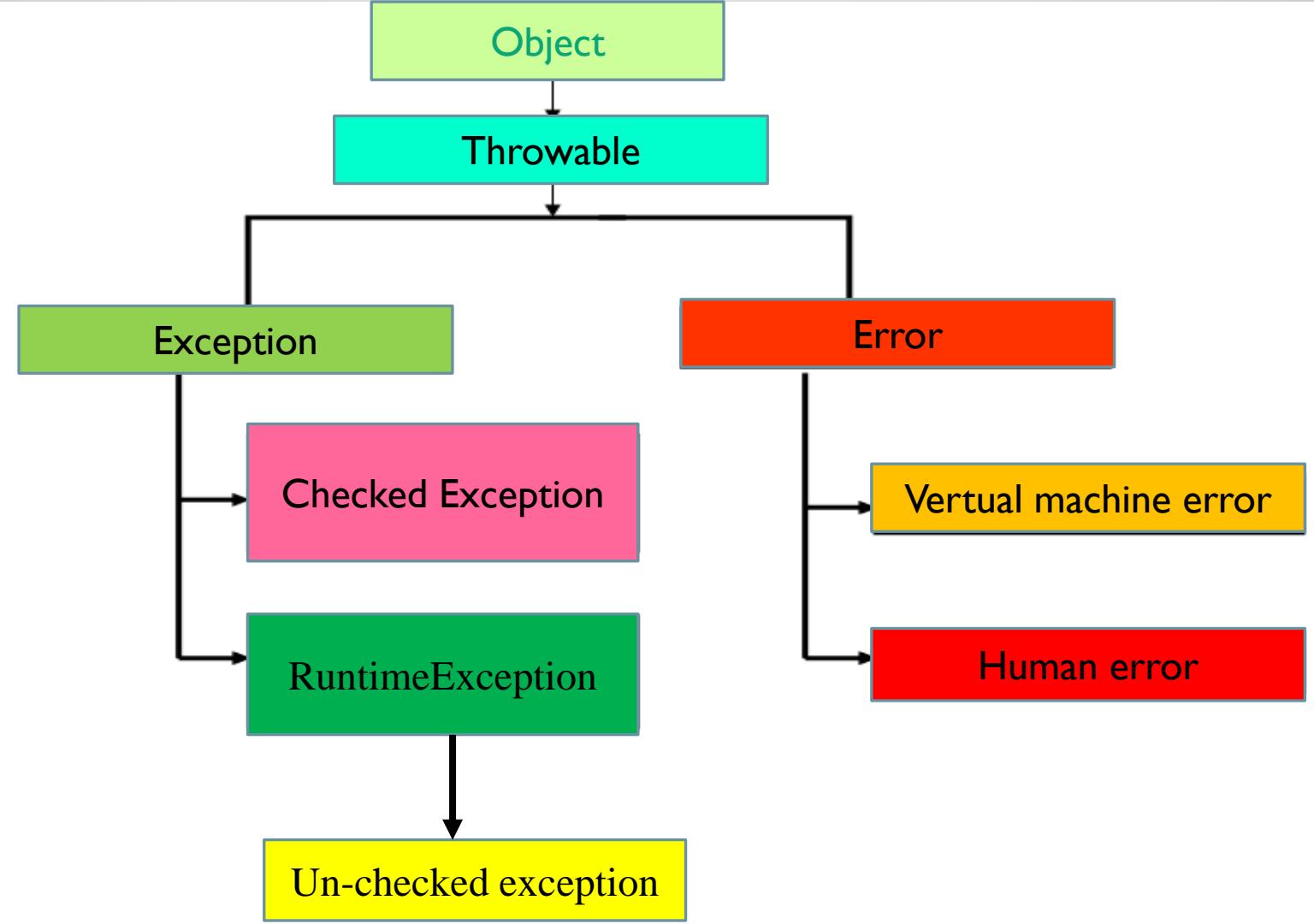
**2. toString(): String**

- Returns a string representation of the exception, typically including the exception's class name and a brief message.

**3. getMessage(): String**

- Returns the detailed message associated with the exception. This message provides additional information about the cause of the exception.

## Exception hierarchy :



## Collection :

A collection is a data structure that organizes and stores multiple elements, providing efficient ways to manage . It can be dynamic, store various data types, and offer operations like adding, removing, searching, and sorting.

## Diff b/w Array and collection ?

Feature	Array	Collection
Type	Object	Interface
Data Type	Homogeneous (same data type)	Heterogeneous (different data types)
Primitive Data Types	Can store	Cannot store
Fixed Size	Yes	No
Methods	Limited built-in methods	Rich set of methods for various operations

## **What is Collection (Interface)?**

- Collection is a predefined interface in Java.
- It was introduced in version JDK 1.2.
- It is present in the `java.util` package.
- In Java, the collection framework provides a standard architecture to store groups of objects.

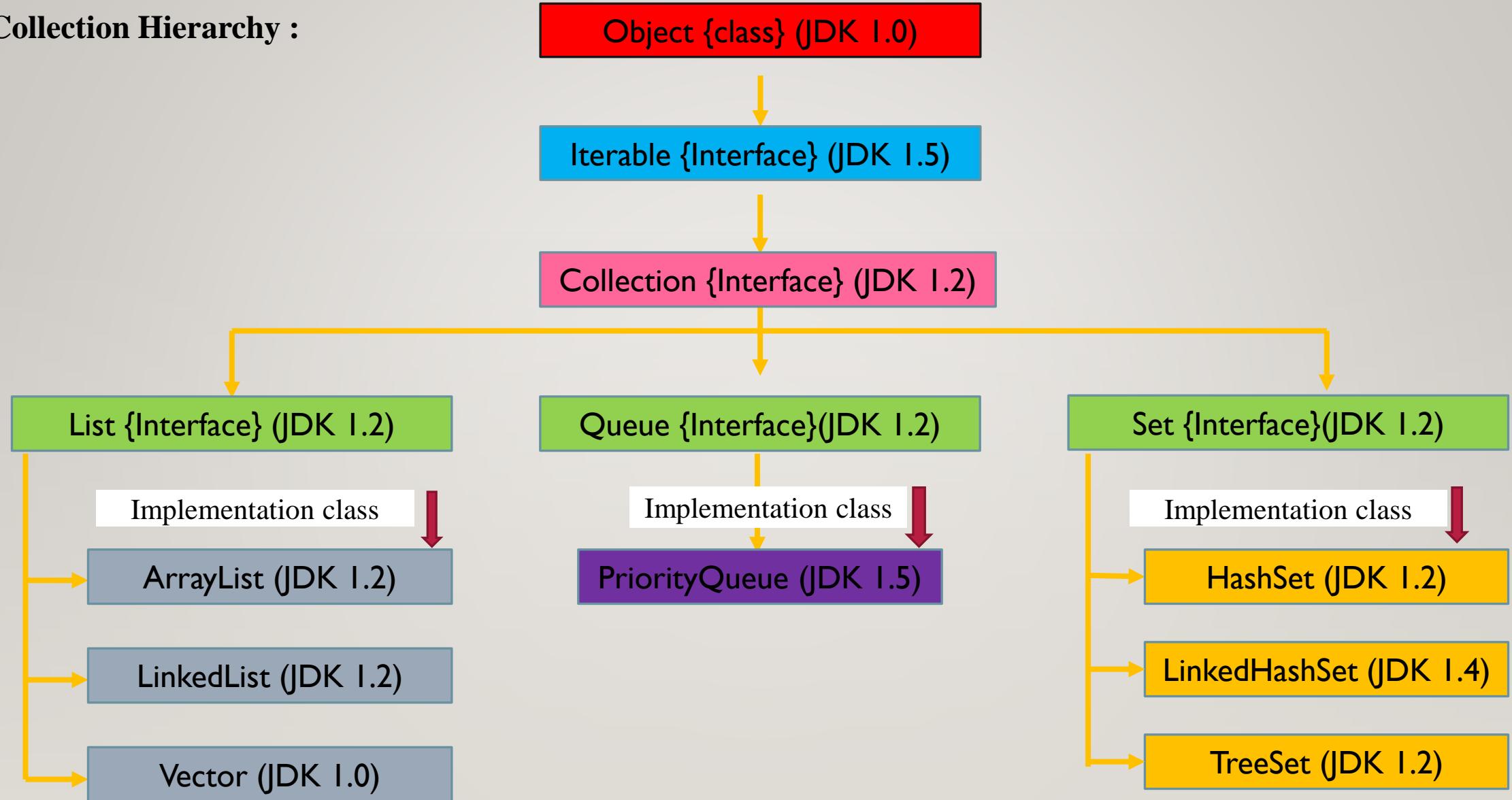
### **Note Point:**

- Collection is nothing but a collection/group of objects.

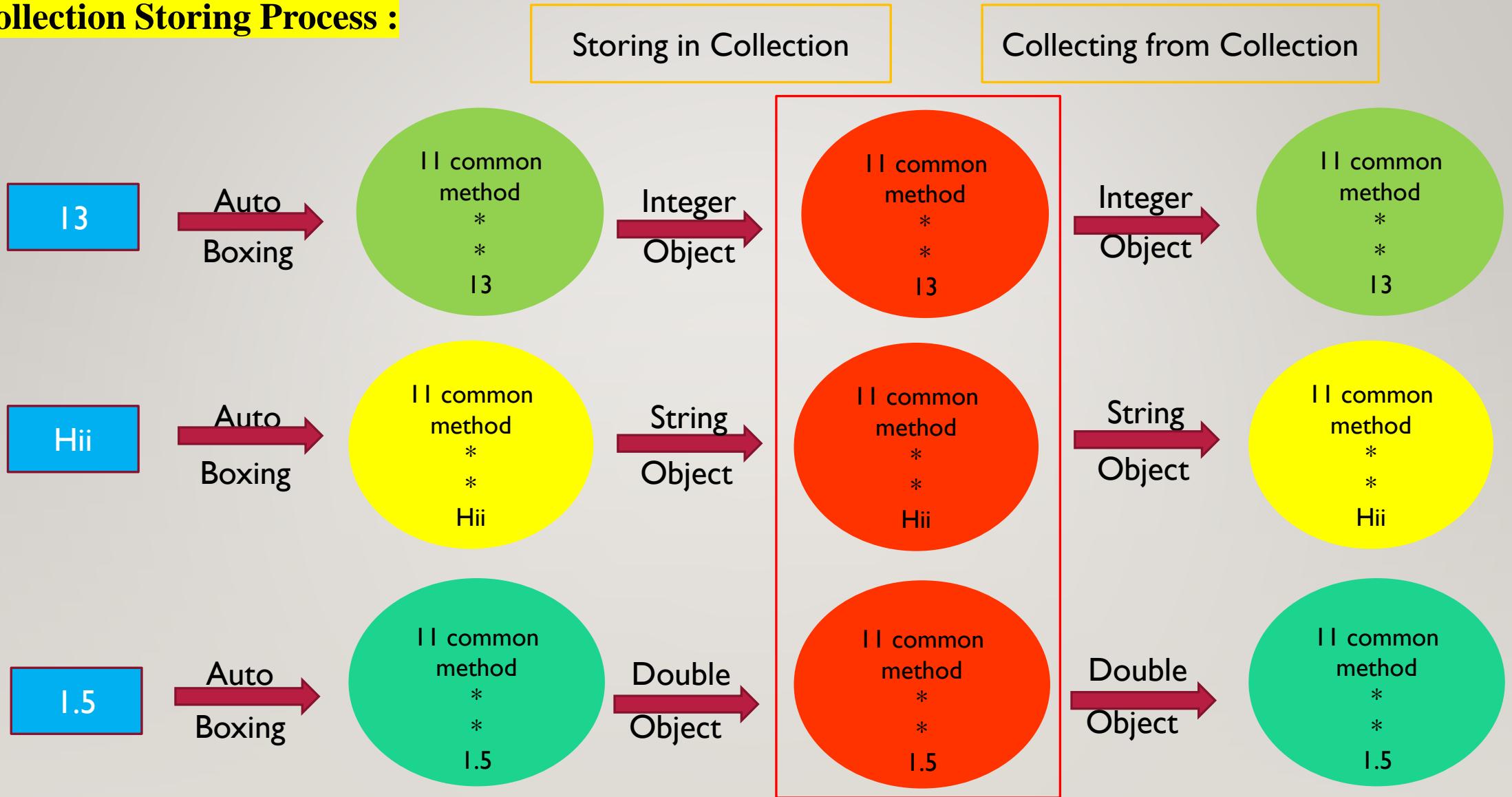
## **What is Framework?**

- Framework represents a group of classes and interfaces.

## Collection Hierarchy :



## Collection Storing Process :



## Diff b/w Generic Type Collection and Raw Type Collection?

Feature	Raw Type Collection	Generic Type Collection
Syntax	<code>ArrayList list = new ArrayList();</code>	<code>ArrayList&lt;ReferenceType&gt; list = new ArrayList&lt;ReferenceType&gt;();</code>
Type Safety	Not type-safe. Can hold any type of data.	Type-safe. Can only hold the defined type of data.
Compile-Time Checking	No compile-time type checking for elements.	Checked for type safety at compile-time.
Typecasting	Individual type casting needed at every retrieval.	No typecasting needed.
Generics Support	No generics support.	Full generics support.
Warnings	Generates unchecked warnings at compile time.	No unchecked warnings.
Example	<code>list.add("hello");</code> <code>list.add(123);</code>	<code>list.add("hello");</code> <code>list.add(123); // Compile-time error</code>

## List

- List is a predefined subinterface of the Collection interface.
- It was introduced in the version of JDK 1.2.
- It is present in the java.util package.
- List interface having 4 implementation classes:
  1. ArrayList
  2. LinkedList
  3. Vector

### Common qualities of List type of collection:

- It is index-based.
- It allows duplicate values.
- It allows null values.

## ArrayList

- ArrayList is a predefined implementation class of the List interface.
- It was introduced in the version of JDK 1.2.
- It is present in the java.util package.
- ArrayList is index-based.
- It allows duplicate values.
- It allows null values.
- ArrayList is both homogeneous as well as heterogeneous.
- ArrayList follows insertion order.
- The initial or default capacity of ArrayList is 10 continuous memory blocks.
- The auto-growable capacity of ArrayList is half of its previous capacity + 1.

### Syntax :

```
ArrayList<Character> arraylist = new ArrayList<>();
```

## Basic Operations:

- **add(E e):** Adds the specified element to the end of the list.
- **add(int index, E element):** Inserts the specified element at the specified position in the list.
- **clear():** Removes all of the elements from this list.
- **contains(Object o):** Returns true if this list contains the specified element.
- **get(int index):** Returns the element at the specified position in this list.
- **indexOf(Object o):** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
- **isEmpty():** Returns true if this list contains no elements.
- **lastIndexOf(Object o):** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

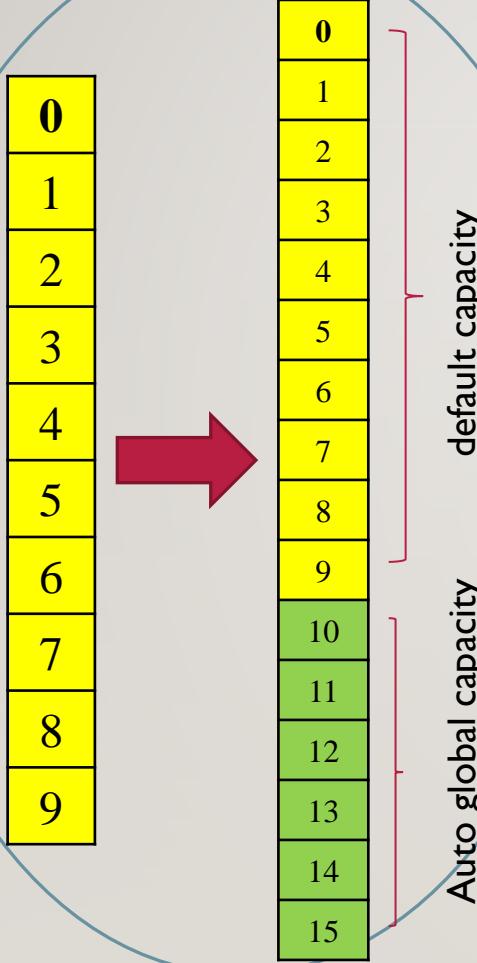
- **remove(int index):** Removes the element at the specified position in this list.
- **remove(Object o):** Removes the first occurrence of the specified element from this list, if it is present.
- **set(int index, E element):** Replaces the element at the specified position in this list with the specified element.
- **size():** Returns the number of elements in this list.

## Search and Sort:

- **containsAll(Collection<?> c):** Returns true if this list contains all of the elements of the specified collection.
- **indexOfAll(Object o):** Returns a list containing all indices of the specified element in this list.
- **lastIndexOfAll(Object o):** Returns a list containing all indices of the last occurrence of the specified element in this list.
- **sort(Comparator<? super E> c):** Sorts this list according to the order induced by the specified comparator.

```
ArrayList<Integer> arraylist = new ArrayList<>();
```

Default capacity = 10 , auto global capacity=6  
AGC=Half of the previous +1



Exa :

```
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Collections;  
public class ArrayListExample {  
    public static void main(String[] args) {  
        // Create an ArrayList to store integers  
        ArrayList<Integer> numbers = new ArrayList<>();  
        // Adding elements  
        numbers.add(10);  
        numbers.add(20);  
        numbers.add(5);  
        numbers.add(15);  
        // Printing the ArrayList  
        System.out.println("Original ArrayList: " + numbers);  
    }  
}
```

```
// Getting the size  
  
int size = numbers.size();  
  
System.out.println("Size of the ArrayList: " + size);  
  
// Checking if an element exists  
  
boolean contains20 = numbers.contains(20);  
  
System.out.println("Does the ArrayList contain 20? " +  
contains20);  
  
// Getting an element by index  
  
int thirdElement = numbers.get(2);  
  
System.out.println("Third element: " + thirdElement);  
  
// Removing an element by index  
  
numbers.remove(1);  
  
System.out.println("ArrayList after removing the second  
element: " + numbers);  
  
// Removing an element by value  
  
numbers.remove(Integer.valueOf(15));  
  
System.out.println("ArrayList after removing 15:" + numbers);
```

```
// Clearing the ArrayList  
  
numbers.clear();  
  
System.out.println("ArrayList after clearing: " + numbers);  
  
// Adding multiple elements using the addAll method  
  
numbers.addAll(Arrays.asList(1, 2, 3, 4, 5));  
  
System.out.println("ArrayList after adding multiple  
elements: " + numbers);  
  
// Sorting the ArrayList  
  
Collections.sort(numbers);  
  
System.out.println("Sorted ArrayList: " + numbers);  
  
// Reversing the ArrayList  
  
Collections.reverse(numbers);  
  
System.out.println("Reversed ArrayList: " + numbers);  
  
// Checking if the ArrayList is empty  
  
boolean isEmpty = numbers.isEmpty();  
  
System.out.println("Is the ArrayList empty? " + isEmpty);  
} }
```

## Exa : 2

```
// Generic ArrayList to store Strings
```

```
ArrayList<String> names = new ArrayList<>();  
names.add("Alice");  
names.add("Bob");  
names.add("Charlie");
```

```
// Generic HashMap to store key-value pairs of Integer  
and String
```

```
HashMap<Integer, String> map = new HashMap<>();  
map.put(1, "Apple");  
map.put(2, "Banana");  
map.put(3, "Cherry");
```

```
// Raw ArrayList
```

```
ArrayList list = new ArrayList();  
list.add("Apple");  
list.add(123);  
list.add(true); // This is not type-safe and can lead to  
runtime errors
```

```
// Raw HashMap
```

```
HashMap map2 = new HashMap();  
map2.put("name", "Alice");  
map2.put("age", 30);  
map2.put("city", "New York"); // Again, not type-safe
```

### **Exa : 3 retrieve element from ArrayList 1 by 1?**

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Create an ArrayList to store strings
        ArrayList<String> names = new ArrayList<>();

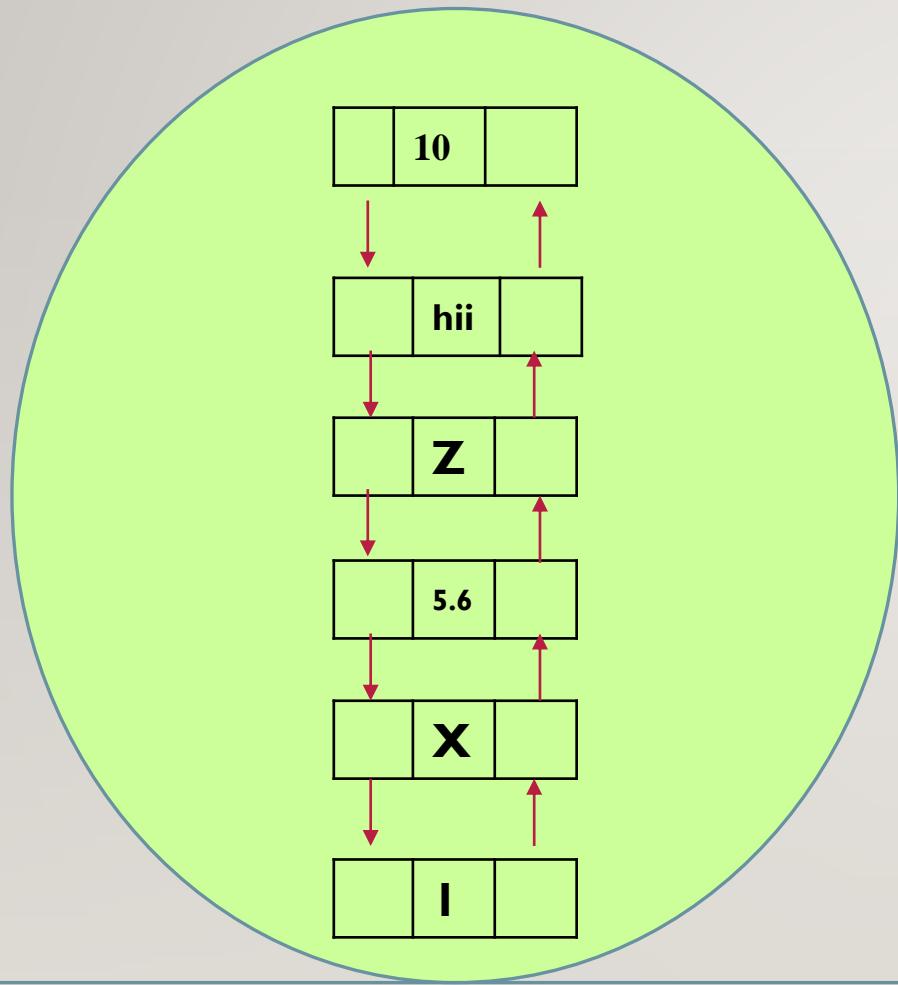
        // Add elements to the ArrayList
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Iterate through the ArrayList using a for-each
        loop
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

### **LinkedList: (Double Linked List)**

- LinkedList is a predefined implementation class of the List interface.
- It was introduced in the version of JDK 1.2.
- It is present in the java.util package.
- LinkedList is index-based.
- It allows duplicate values.
- It allows null values.
- LinkedList follows insertion order(Same inserting order).
- LinkedList is both homogeneous and heterogeneous.
- In LinkedList, there is no initial capacity.
- Whenever we are adding elements into the collection, that time instantly one memory block will get created.
- Each memory block knows where my next object is present and next object knows where my previous object is present.

## Work flow and architecture :

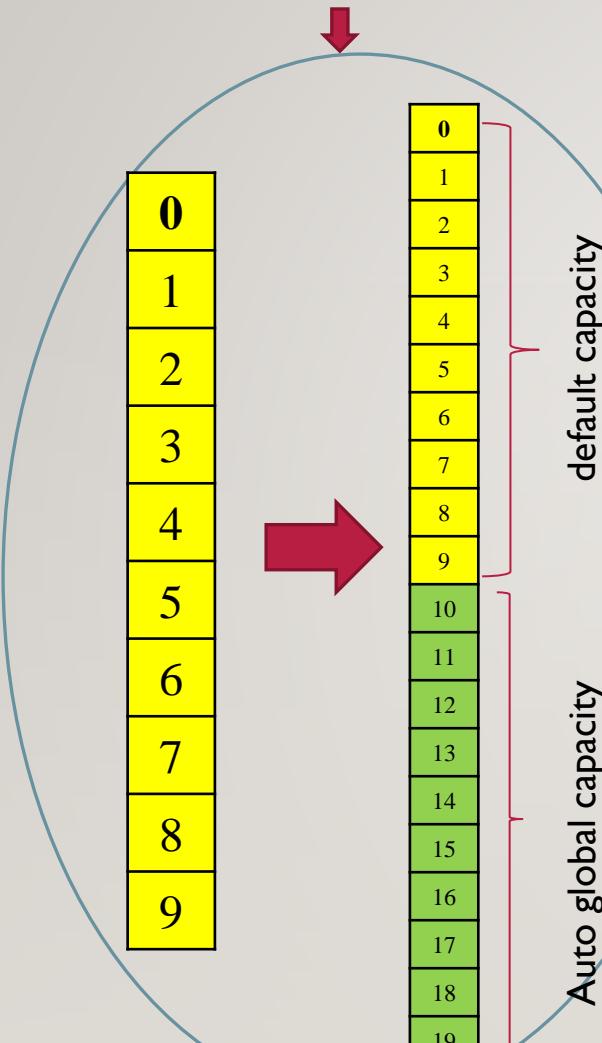


## Exa :

```
import java.util.LinkedList;  
public class LinkedListExample {  
    public static void main(String[] args) {  
        // Create a LinkedList to store strings  
        LinkedList<String> names = new LinkedList<>();  
        // Add elements to the LinkedList  
        names.add("Alice");  
        names.add("Bob");  
        names.add("Charlie");  
        // Iterating through the LinkedList using a for-each loop  
        for (String name : names) {  
            System.out.println(name);  
        }  
    }  
}
```

AGC=Half of the previous \*2  
Default capacity =10 , auto global capacity=10

```
Vector arraylist = new Vector();
```



### Vector/Legacy Class:

- Vector is a predefined implementation class of the List interface.
- It was introduced in the version of JDK 1.0.
- It is present in the `java.util` package.
- Vector class is also called as Legacy class because vector class was introduced before introducing collection.
- Vector class is thread-safe.
- It allows duplicate and null values.
- Vector follows insertion order.
- Vector is both homogeneous and heterogeneous.
- The initial capacity of vector is 10 continuous memory blocks.
- The auto-growable capacity of vector is half of its previous capacity + 1.

Syntax:

```
Vector vector = new Vector();
```

### Exa :

```
import java.util.Vector;  
  
public class VectorExample {  
  
    public static void main(String[] args) {  
  
        // Create a Vector to store strings  
  
        Vector<String> names = new Vector<>();  
  
        // Add elements to the Vector  
  
        names.add("Alice");  
  
        names.add("Bob");  
  
        names.add("Charlie");  
  
        // Iterating through the Vector using a for-each loop  
  
        for (String name : names) {  
  
            System.out.println(name);  
  
        }  
  
    }  
}
```

### Queue

- Queue is a sub-interface of the Collection interface.
- It was introduced in the version of JDK 1.2.
- It is present in the java.util package.
- Queue interface having one predefined implementation class.

### Common qualities of Queue interface:

- Queue is not index-based.
- Queue allows duplicate values.
- Queue does not allow null values.

## PriorityQueue

- PriorityQueue is a predefined implementation class of the Queue interface.
- It was introduced in the version of JDK 1.5.
- PriorityQueue is not index-based.
- It is present in the java.util package.
- It allows duplicate values.
- It does not allow null values.
- If the programmer tries to store a null value, that time null value will get a NullPointerException.
- PriorityQueue is only homogeneous. If we try to store different data types, we will get a ClassCastException.

## Priority queue having two predefined own methods:

### 1. peek(): Object

- it is non-static method present in PriorityQueue, peek() is used to return/retrieve/get the head element from the collection.

### 2. poll(): Object

- it is non-static method present in PriorityQueue, poll() first it will retrieve the head object and permanently remove it from the collection.

Exa :

```
import java.util.PriorityQueue;  
public class PriorityQueueExample {  
    public static void main(String[] args) {  
        // Create a PriorityQueue 1 to store integers  
        PriorityQueue<Integer> pq = new PriorityQueue<>();  
        // Add elements to the PriorityQueue  
        pq.add(10);  
        pq.add(5);  
        pq.add(15);  
        pq.add(2);  
        pq.add(8);
```

```

// Printing the PriorityQueue

System.out.println("PriorityQueue: " + pq);

// Retrieving the minimum element without removing it

int minElement = pq.peek();

System.out.println("Minimum element: " + minElement);

// Removing and returning the minimum element

int removedElement = pq.poll();

System.out.println("Removed minimum element: " +
removedElement);

// Printing the PriorityQueue after removal

System.out.println("PriorityQueue after removal: " + pq);

}

```

**Is it possible to keep other statements between the try and catch block and finally block?**

- No, we cannot able to write code between the try-catch and finally blocks.

**Can we keep other statements between the try and catch block and finally block?**

- No, we cannot able to write code between the try-catch and finally blocks. They are mutually coupled.

**Why do you use the throws keyword in Java?**

- To propagate exceptions from a called method to a caller method, we use the throws keyword in Java.

### **Set:**

- Set is a predefined subinterface of the Collection interface.
- It was introduced in the version of JDK 1.2.
- It is present in the java.util package.
- Set having four implementation classes:
  1. HashSet
  2. LinkedHashSet
  3. TreeSet
  4. ConcurrentHashMap.keySet()

### **Common qualities of Set interface:**

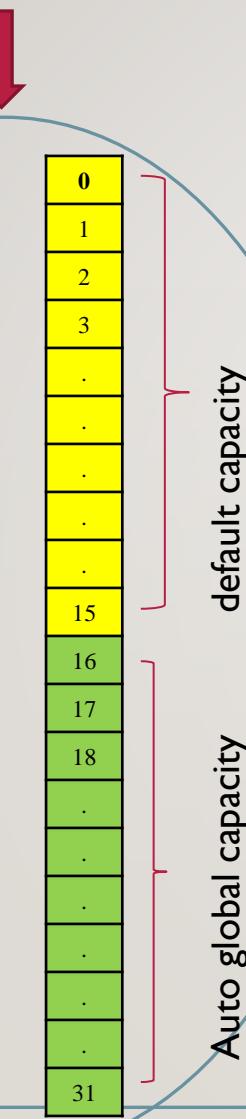
- Set is not index-based.
- Set does not allow duplicate values.

### **HashSet**

- HashSet is a predefined implementation class of the Set interface.
- It was introduced in the version of JDK 1.2.
- It is present in the java.util package.
- HashSet is not index-based.
- HashSet does not allow duplicate values.
- HashSet allows null value.
- HashSet is both homogeneous and heterogeneous.
- HashSet does not follow insertion order, but internally it follows hashing technique order.
- Initial capacity of HashSet is 16 continuous memory blocks.
- The fill ratio threshold value is 75%, which means whenever 12 memory blocks are filled, it will auto-grow half of its previous capacity.
- It will not wait until all memory blocks are filled.

AGC=Half of the previous \*2  
Default capacity =16 , auto global capacity=16

HashSet arraylist = new HashSet();



Exa :

```
import java.util.HashSet;  
  
public class HashSetExample {  
  
    public static void main(String[] args) {  
  
        // Create a HashSet to store strings  
        HashSet<String> names = new HashSet<>();  
  
        // Add elements to the HashSet  
        names.add("Alice");  
        names.add("Bob");  
        names.add("Charlie");  
        names.add("Alice"); // Duplicate element, will not be added  
  
        // Printing the HashSet  
        System.out.println("HashSet: " + names);  
    }  
}
```

## LinkedHashSet

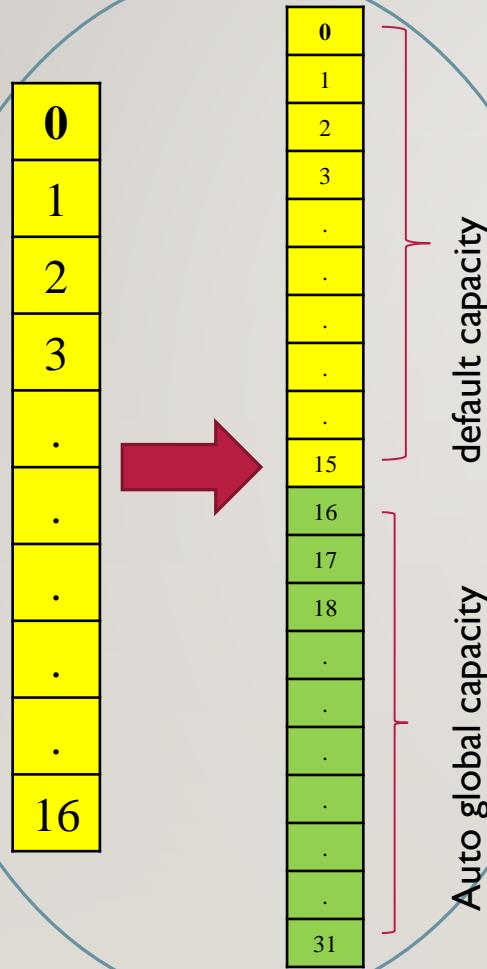
- LinkedHashSet is a predefined implementation class of the Set interface.
- It was introduced in the version of JDK 1.4.
- It is present in the java.util package.
- LinkedHashSet is not index-based.
- It does not allow duplicate values.
- It allows null value.
- LinkedHashSet is both homogeneous and heterogeneous.
- LinkedHashSet follows insertion order.
- The initial capacity of LinkedHashSet is 16 continuous memory blocks.
- The fill ratio threshold value is 75%.
- It will auto-grow half of its previous capacity\*2.

**Exm : retrieve common element from 2 collection and store in 3<sup>rd</sup> collection?**

```
package LinkedHashSet;  
import java.util.LinkedHashSet;  
public class PrintCommonObject {  
    public static void main(String[] args) {  
        LinkedHashSet<Integer> 11 = new LinkedHashSet<>();  
        // ... (add elements to 11)  
        LinkedHashSet<Integer> 12 = new LinkedHashSet<>();  
        // Find common elements and store them  
        LinkedHashSet<Integer> commonElements = new  
        LinkedHashSet<>(11);  
        commonElements.retainAll(12);  
        // Print the common elements  
        System.out.println("Common elements: " +  
        commonElements);  
    } }
```

```
LinkedHashSet arraylist = new LinkedHashSet();
```

Default capacity = 16 , auto global capacity=16  
AGC=Half of the previous \*2



**Can we throw exception manually/explicitly?**

- Yes, we can throw an exception by using the throw keyword. It will throw the exception called method to the caller method.

**Are we allowed to use only a try block without a catch block and finally block?**

- No, because both catch and try blocks are mutually combined. Without a catch block, how will you know which exception he needs to handle? You will get confused, hence must await with a try-catch block.

**Is a finally block always executed in a Java program?**

- Yes, a finally block gets executed in every condition irrespective of checking of conditions (abnormal or normal).

**What will happen if an exception is thrown by the main method?**

- We can able to throw an exception from the main method to the caller method by using the throws keyword.

## TreeSet:

- ❑ TreeSet is a predefined implementation class of the Set interface.
- ❑ It was introduced in JDK 1.2.
- ❑ It is present in the java.util package.
- ❑ It is *not* index-based (meaning you can't access elements by index like in an array or list).
- ❑ It does *not* allow duplicate values. If you try to add a duplicate, it's ignored.
- ❑ It does *not* allow null values. If you try to store null, you'll get a NullPointerException.
- ❑ TreeSet is *only homogeneous*. If a programmer tries to store heterogeneous (different types) objects, a ClassCastException will be thrown.
- ❑ TreeSet does *not* follow insertion order because internally it follows *sorting order*.
- ❑ When we try to add an object, it instantly sorts.

Exa:

```
package set;
import java.util.TreeSet;

public class TreeSet {
    public static void main(String[] args) {
        TreeSet<Integer> t = new TreeSet<Integer>();
        t.add(6);
        t.add(3);
        t.add(8);
        t.add(5);
        t.add(1);
        t.add(7);
        t.add(11);
        t.add(9);
        t.add(13);
        System.out.println(t);
    }
}
```

## Diff b/w List vector and set implementation classes ?

	List (Interface)			Queue	Set (Interface)		
Feature	ArrayList	LinkedList	Vector	PriorityQueue	HashSet	LinkedHashSet	TreeSet
Introduced	JDK 1.2	JDK 1.2	JDK 1.0 (Legacy)	JDK 1.5	JDK 1.2	JDK 1.4	JDK 1.2
Insertion Order	Follows	Follows	Follows	Follows minimum heap order (Parent-Child)	Not followed (Hashing order)	Follows insertion order	Not followed (Sorting order)
Index-based	Index-based	Index-based	Index-based	Not index-based	Not index-based	Not index-based	Not index-based
Duplicate Values	Allowed	Allowed	Allowed	Not allowed	Not allowed	Not allowed	Not allowed
Null Values	Allowed	Allowed	Allowed	Not allowed	Not allowed	Allows one null value	Not allowed
Initial Capacity	10 continuous memory blocks	No initial capacity	10 continuous memory blocks	No initial capacity	16 continuous memory blocks	16 continuous memory blocks	No fixed capacity

Feature	ArrayList	LinkedList	Vector	PriorityQueue	HashSet	LinkedHashSet	TreeSet
Homogeneous/Heterogeneous	Homogeneous / Heterogeneous	Homogeneous / Heterogeneous	Homogeneous / Heterogeneous	Homogeneous	Homogeneous / Heterogeneous	Homogeneous / Heterogeneous	Homogeneous
Auto-grow capacity	Half of previous capacity + 1	No auto-growth	Half of previous capacity + 1	No auto-growth capacity	Half of previous capacity + 1	Half of previous capacity + 1	Don't have
Fill ratio	100%	Does not have	100%	Does not have	75%	75%	Don't have
Thread-safe	Not thread-safe	Not thread-safe	Thread-safe	Not thread-safe	Not thread-safe	Not thread-safe	Not thread-safe
Time complexity to add an element	O(1)	O(1)	O(1)	O(log n)	O(1)	O(1)	O(log n)
Time complexity to get an element by index	O(1)	O(n)	O(1)	Not applicable	Not applicable	Not applicable	Not applicable

## Cursor

- What is the purpose of cursor?
  - The purpose of cursor is to retrieve/return the object one by one from the collection.
- What is cursor?
  - Cursor is nothing but traversing.
  - Traversing means cursor should move from one object to another object.
- In Java we have 3 predefined cursors:
  - Iterator
  - ListIterator
  - Enumeration

Collection	Iterator	ListIterator	Enumeration
ArrayList	✓	✓	X
LinkedList	✓	✓	X
Vector	✓	✓	✓
PriorityQueue	✓	X	X
HashSet	✓	X	X
LinkedHashSet	✓	X	X
TreeSet	✓	X	X

## Iterator

- Iterator is a predefined interface in java.
- It is present in the `java.util` package.
- It was introduced in the version of JDK 1.2.
- Iterator cursor is also called as universal cursor, because it is applicable to all type of collection.
- Iterator cursor travels only in forward direction, that means it will retrieve only forward object

## Iterator having 3 predefined methods:

1. **next(): Object**
2. **hasNext(): boolean**
3. **remove(): void**

Exa : import java.util.ArrayList;

import java.util.Iterator;

public class IteratorExample {

    public static void main(String[] args) {

        // Create an ArrayList to store strings

        ArrayList<String> names = new ArrayList<>();

        names.add("Alice");

        names.add("Bob");

        names.add("Charlie");

        // Get an iterator for the ArrayList

        Iterator<String> iterator = names.iterator();

        // Iterate through the ArrayList using the iterator

```
while (iterator.hasNext()) {
```

```
    String name = iterator.next();
```

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

```
}
```

```
// Remove an element using the iterator
```

```
iterator = names.iterator();
```

```
while (iterator.hasNext()) {
```

```
    String name = iterator.next();
```

```
    if (name.equals("Bob")) {
```

```
        iterator.remove();
```

```
}
```

```
}
```

```
System.out.println("ArrayList after removing Bob: " +  
names);
```

```
}
```

```
}
```

## ListIterator

- ListIterator is a predefined interface which is present in java.util package.
- It was introduced in the version of JDK 1.2.
- ListIterator travels both forward direction and backward direction.

### ListIterator having 5 predefined methods:

1. **next(): Object**
2. **hasNext(): boolean**
3. **previous(): Object**
4. **hasPrevious(): boolean**
5. **remove(): void**
6. **add(): void**

Exa :

```
import java.util.ArrayList;
import java.util.ListIterator;
public class ListIteratorExample {
```

```
public static void main(String[] args) {
    // Create an ArrayList 1 to store strings
    ArrayList<String> names = new ArrayList<>();
    names.add("Alice");
    names.add("Bob");
    names.add("Charlie");
    names.add("David");
    // Get a ListIterator for the ArrayList
    ListIterator<String> iterator = names.listIterator();
    // Iterate forward
    while (iterator.hasNext()) {
        String name = iterator.next();
        System.out.println(name);
    }
    System.out.println();
```

```

// Iterate backward

while (iterator.hasPrevious()) {

    String name = iterator.previous();
    System.out.println(name);
}

// Add an element at the beginning
iterator.add("Eve");

System.out.println("ArrayList after adding Eve: " +
names);

// Remove an element
iterator.next(); // Move to the next element (Bob)
iterator.remove();

System.out.println("ArrayList after removing Bob: " +
names);

}
}

```

## Enumeration/Legacy Cursor

- Enumeration is a predefined interface which is present in `java.util` package.
- It was introduced in the version of JDK 1.0.
- Enumeration is also called as legacy cursor.
- Enumeration travels only forward direction.

## Disadvantage

- In Enumeration, we cannot remove the object because we don't have a `remove` method in the Enumeration interface.

## Enumeration having two predefined methods:

1. `nextElement(): Object`
2. `hasMoreElements(): Boolean`

```

Exa : import java.util.Enumeration;
import java.util.Vector;
public class EnumerationExample {
    public static void main(String[] args) {
        // Create a Vector 1 to store strings
        Vector<String> names = new Vector<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        // Get an Enumeration for the Vector
        Enumeration<String> enumeration = names.elements();
        // Iterate through the Vector using the Enumeration
        while (enumeration.hasMoreElements()) {
            String name = enumeration.nextElement();
            System.out.println(name);
        }
    }
}

```

### **java.lang Package:**

- Object class, Object, String, System, Exception, RuntimeException, Double, Integer

### **java.util Package:**

- Collection<E>, List<E>, ArrayList<E>, LinkedList<E>, Queue

### **Arrays**

- File, InputStream, OutputStream, Reader, Writer, FileInputStream, FileOutputStream

### **Difference between length() and length variable:**

- ❑ length() is used to find the length of a String. It is a method, so it requires parentheses () and is not applicable for arrays.
- ❑ length variable is used to find the length of an array.
- ❑ In Java, array size is constant. After declaration time, we cannot change the array size throughout the program. (Java arrays are fixed)

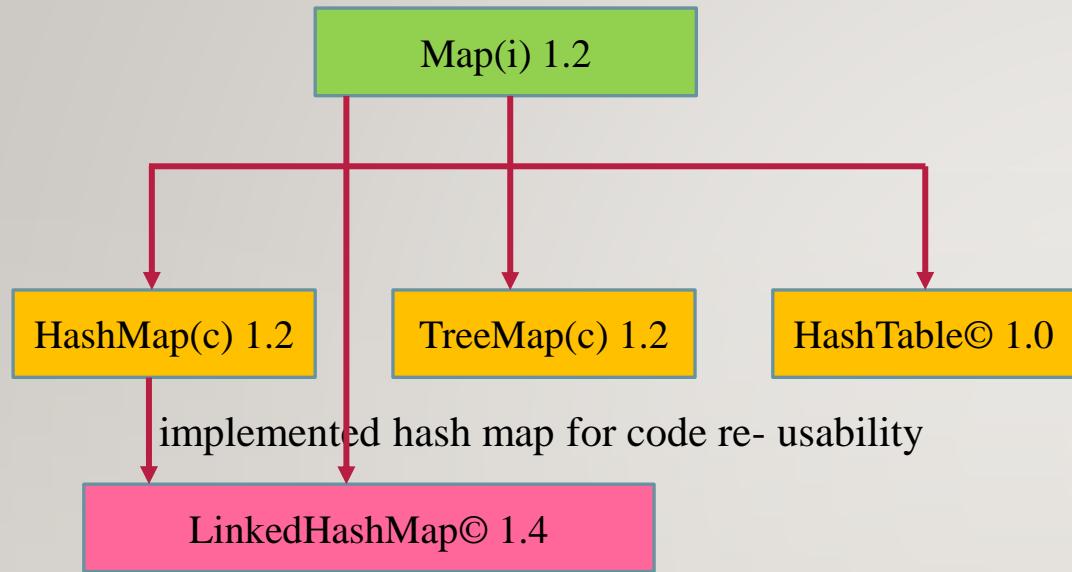
## Diff b/w Iterator, ListIterator and Enumeration ?

Feature	Iterator	ListIterator	Enumeration
Working with	Various collections like List, Set, etc.	Works with List only	Works with legacy collections (Vector)
Direction	Forward only	Bidirectional	Forward only
Modification	Can modify the collection	Can modify the list	Cannot modify the collection
Superclass	None	Iterator	None
Operations	next(), hasNext(), remove()	next(), hasNext(), previous(), hasPrevious(), add(), remove()	nextElement(), hasMoreElements()
Purpose	General purpose iterator	List-specific bidirectional access	Legacy code, read-only
Introduced	JDK 1.2	JDK 1.2	JDK 1.0 (legacy cursor)

## Diff b/w final, finally and finalize()?

Feature	final	finally	finalize()
Purpose	To make something unchangeable	To ensure code execution	To de-refer all object before garbage collection
Usage	For classes, methods, and variables	In try-catch-finally blocks	Invoked by the garbage collector
Syntax	<code>final int a = 10; public final class A { }</code>	<code>try { // code } catch (Exception e) { // handle exception } finally { // code to be executed always }</code>	<code>protected void finalize() throws Throwable { }</code>

## Hierarchy of map Interface :



## Map Object

- In collections, we can store only one object at a time. To overcome this limitation, we are going for Map Object.
- Map is an interface which is having 3 implementation classes.
  1. **HashMap**
  2. **LinkedHashMap**
  3. **TreeMap**

Map Object will be containing object in the form of key and value.

- Key should not be duplicated and value can be duplicated.
- Each key and value pair are known as entry object.
- Because of this reason Map Object is also known as collection of entry objects.
- Whenever we want to store a single object, go with collection, whenever we want to store in the form of key and value pair at that time make use of Map Object.

## **Map Interface Methods:**

### **1. put(KeyObject, ValueObject): ValueObject**

- This method is used for adding the element inside the Map Object.
- This method is taking two inputs in the form of key and value.
- If we print this method, the mapped value will be printed.

### **2. containsKey(KeyObject): boolean**

- This method will be checking whether the given key object is present or not.
- If the given key is present, this method will be returning true, if not, this method will be returning false.

### **3. remove(KeyObject): ValueObject**

- This method will be removing the given key.

### **4. containsValue(ValueObject): boolean**

- This method will be checking whether the given value is present or not.

- If the given value is present, this method will return true; if not, this method will return false.

### **5. keySet(): Set<K>**

- Returns a Set view of the keys contained in this map.
- The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

### **6. values(): Collection<V>**

- Returns a Collection view of the values contained in this map.
- The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

### **7. replace(K key, V oldValue, V newValue): boolean**

- Replaces the value associated with the specified key with the given value only if the current value associated with the key is equal to the given old Value.
- Returns true if the value was replaced, false otherwise.

### **8. entrySet(): Set<Map.Entry<K,V>>**

- Returns a Set view of the mappings contained in this map.

- Each element in the set is a Map.Entry, which represents a key-value pair.
- The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

Entry Interface contains 3 methods which is used for performing separately on key and separately on values, the methods are as follow:

- 1. getKey(): Key**
- 2. getValue(): Value**
- 3. setValue(newValue): void**

#### Note:

This three method are based on cursor. It is always give cursor pointing value on key.

#### HashMap :

- HashMap is an inbuilt class and implementation class of the Map interface.
- We can create HashMapObject in 4 ways:
  - 1. HashMap()**
  - 2. HashMap(int initialCapacity)**
  - 3. HashMap(int initialCapacity, float loadFactor)**
  - 4. HashMap(Map<K,V> m)**
- If we give initial capacity less than 0, it will give an exception.

#### Characteristics of HashMap

- HashMap will be taking objects in the form of key and value.
- Key should not be duplicated, but value can be duplicated.
- One null key is allowed, multiple null values are allowed.
- The data structure of HashMap is Hashtable.
- The output order of HashMap will be in the form of Hashing Technique order.

**Exa :**

```
public class HashMapExample {  
    public static void main(String[] args) {  
        // Create a HashMap to store key-value pairs  
        HashMap<String, Integer> ages = new  
        HashMap<>();  
  
        // Add key-value pairs to the HashMap  
        ages.put("Alice", 25);  
        ages.put("Bob", 30);  
        ages.put("Charlie", 28);  
        ages.put("David", 30); // Duplicate value is allowed  
  
        // Accessing values  
        int aliceAge = ages.get("Alice");  
        System.out.println("Alice's age: " + aliceAge);
```

```
// Checking if a key exists  
  
        boolean containsBob = ages.containsKey("Bob");  
        System.out.println("Does the HashMap contain Bob? "  
        + containsBob);  
  
        // Removing a key-value pair  
        ages.remove("Charlie");  
        System.out.println("HashMap after removing Charlie: "  
        + ages);  
  
        // Getting all keys  
        System.out.println("Keys: " + ages.keySet());  
  
        // Getting all values  
        System.out.println("Values: " + ages.values());  
    }  
}
```

## #LinkedHashMap

- LinkedHashMap is an implementation class of the Map interface.
- At the same time, it inherits the HashMap class for code reusability.
- It was introduced in JDK version 1.4.
- In LinkedHashMap, we can store keys and values, but keys should not be duplicated. Values can be duplicated.
- The data structure of LinkedHashMap is a HashTable + LinkedList implementation. Because of the internal key linkages created, the output order will be in the form of insertion order.
- Whenever we want to store keys and values, and we want to get the insertion order at that time, make use of LinkedHashMap objects.

## Notes:-

- LinkedHashMap is inheriting HashMap to perform constructor chaining and code reusability.

```
Exa : import java.util.LinkedHashMap;  
  
public class LinkedHashMapExample {  
  
    public static void main(String[] args) {  
  
        // Create a LinkedHashMap to store employee  
        information  
  
        LinkedHashMap<String, Employee> employees = new  
        LinkedHashMap<>();  
  
        // Create Employee objects  
  
        Employee emp1 = new Employee("Alice", 30, "Software  
        Engineer");  
  
        Employee emp2 = new Employee("Bob", 25, "Data  
        Scientist");  
  
        Employee emp3 = new Employee("Charlie", 28,  
        "Product Manager");  
  
        // Add employees to the LinkedHashMap  
  
        employees.put("Employee1", emp1);  
        employees.put("Employee2", emp2);  
        employees.put("Employee3", emp3);
```

```
// Iterate through the LinkedHashMap and print employee details  
  
for (String key : employees.keySet()) {  
  
    Employee employee = employees.get(key);  
  
    System.out.println("Employee ID: " + key);  
    System.out.println("Name: " + employee.getName());  
    System.out.println("Age: " + employee.getAge());  
    System.out.println("Role: " + employee.getRole());  
    System.out.println();  
}  
  
static class Employee {  
  
    private String name, role;  
    private int age;  
  
    public Employee(String name, int age, String role) {  
        this.name = name;  
        this.age = age;  
        this.role = role;  
    }
```

```
public String getName() {  
    return name;  
}  
  
public int getAge() {  
    return age;  
}  
  
public String getRole() {  
    return role;  
}  
}
```

## TreeMap

- TreeMap is an implementation class of the Map interface.
- It was introduced in the JDK version of 1.2.
- TreeMap will store objects in the form of key and value.
- Key should not be duplicated, but value can be duplicated.
- The data structure of TreeMap is Red-Black Tree.
- Red-Black Sorting is nothing but keys will be sorted according to the sorted key all the values are getting placed.
- The output order of TreeMap will be in the form of sorting order.

## HashTable

- According to data structure, it is a table which contains key and value.
- According to Map interface, Hashtable is an implementation class of Map Interface which was introduced in JDK 1.0.
- Because it was introduced in 1.0, it is also known as a legacy class in Map Interface.

## Exa :

```
import java.util.TreeMap;  
  
public class TreeMapExample {  
  
    public static void main(String[] args) {  
  
        // Create a TreeMap to store key-value pairs 1  
  
        TreeMap<String, Integer> ages = new TreeMap<>();  
  
        // Add key-value pairs to the TreeMap  
  
        ages.put("Alice", 25);  
  
        ages.put("Bob", 30);  
  
        ages.put("Charlie", 28);  
  
        ages.put("David", 30); // Duplicate value is allowed  
  
        // Printing the TreeMap  
  
        System.out.println("TreeMap: " + ages);  
  
    }  
}
```

## Diff b/w HashMap, LinkedHashMap and TreeMap?

Feature	HashMap	LinkedHashMap	TreeMap
<b>Order of Elements</b>	No specific order (hash-based)	Insertion order	Sorting ordering of keys
<b>Null Keys and Values</b>	Allows one null key and multiple null values	Allows one null key and multiple null values	Does not allow null keys
<b>Performance</b>	Generally faster for retrieval and insertion	Slower than HashMap, but faster than TreeMap	Slower than HashMap and LinkedHashMap, but maintains sorted order
<b>Use Cases</b>	When order doesn't matter and performance is critical	When insertion order needs to be preserved	When elements need to be sorted based on keys

## Diff b/w List and Map?

Feature	List	Map
Data Structure	Stores elements in a object.	Stores data in key-value object pairs
Duplicate Elements	Allows duplicate elements	Does not allow duplicate keys, but allows duplicate values
Null Elements	Allows multiple null elements	Allows at most one null key and any number of null values
Access	Elements are accessed by index	Elements are accessed by key
Implementation Classes	ArrayList, LinkedList, Vector	HashMap, TreeMap, LinkedHashMap, Hashtable

## Thread

- Thread is actually a path which is given to JVM for execution purpose by the Thread Scheduler.
- Thread scheduler is the 4th resource given by the Java Development Kit (JDK).
- According to java library, Thread is an inbuilt class which is present inside lang package, it was introduced in JDK version of 1.0.

## Multi-Threading

- In two method, there is no relation or connection between two methods at that time go for multithreading.
- The process of creating Multiple Threads by using single thread is known as multithreading.

## Purpose of Multithreading:

- To reduce the CPU time at the same time developer time
- Creation of videogames and animation
- Servers are built using Multithreading
- User Defined Thread Creation

## Ways to Create a Thread:

### 1. Our Class Inheriting Thread Class

➤ Rules to create:

- Our class has to inherit Thread class to get the property of Thread class
- Override the run() method present in Thread class because whatever the logic provided inside run() will be executed inside our defined Thread or child Thread.

### 2. Our Class Implementing Runnable Interface

- Create a class, class will implements the Runnable interface and Implement the run() Method:
- Create a Thread object.
- For Start the Thread Call the start() method on the Thread object to initiate the execution of the run() method.

## if we call start() again by using same child Thread object?

- It will throw IllegalThreadStateException.

## Thread execution:

- 1) Thread creation
- 2) Thread execution
- 3) Thread Scheduler

## Diff b/w Thread Class and Runnable Interface

Feature	Thread Class	Runnable Interface
Inheritance	Extends Thread class	Implements Runnable interface
Object Creation	Create a subclass of Thread	Create a class that implements Runnable and pass its object to a Thread constructor
Thread Start	Directly call the start() method on the thread object	Create a Thread object and call its start() method, passing the Runnable object as an argument
Multiple Inheritance	Cannot be used with multiple inheritance as Java doesn't support it	Can be used with multiple inheritance as interfaces can be implemented multiple times
Flexibility	Less flexible as you can only create one thread per class	More flexible as you can create multiple threads sharing the same Runnable object

### Exa : By implementing RunnableInterface

```
public class RunnableThreadExample {  
    public static void main(String[] args) {  
        MyRunnableThread thread = new MyRunnableThread();  
        Thread t = new Thread(thread);  
        t.start();  
    }  
  
    static class MyRunnableThread implements Runnable {  
        public void run() {  
            for (int i = 0; i < 5; i++) {  
                System.out.println("Thread 1: " + i);  
                try {  
                    Thread.sleep(1000); // Sleep for 1 second  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

### Exa : By inheriting thread class

```
public class ThreadClassExample extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Thread 2: " + i);  
            try {  
                Thread.sleep(1000); // Sleep for 1 second  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        ThreadClassExample thread = new ThreadClassExample();  
        thread.start();  
    }  
}
```

## Thread Life Cycle

A thread in Java goes through several states during its lifetime:

### Stage 1: New State

- When we create a thread object at that time newly thread is created.

### Stage 2: Ready/Runnable

- When developer calls start() explicitly, this stage is runnable state.

### Stage 3: Running

- When JVM calls the run() implicitly, this stage is running state.

### Stage 4: Waiting/dead-lock State

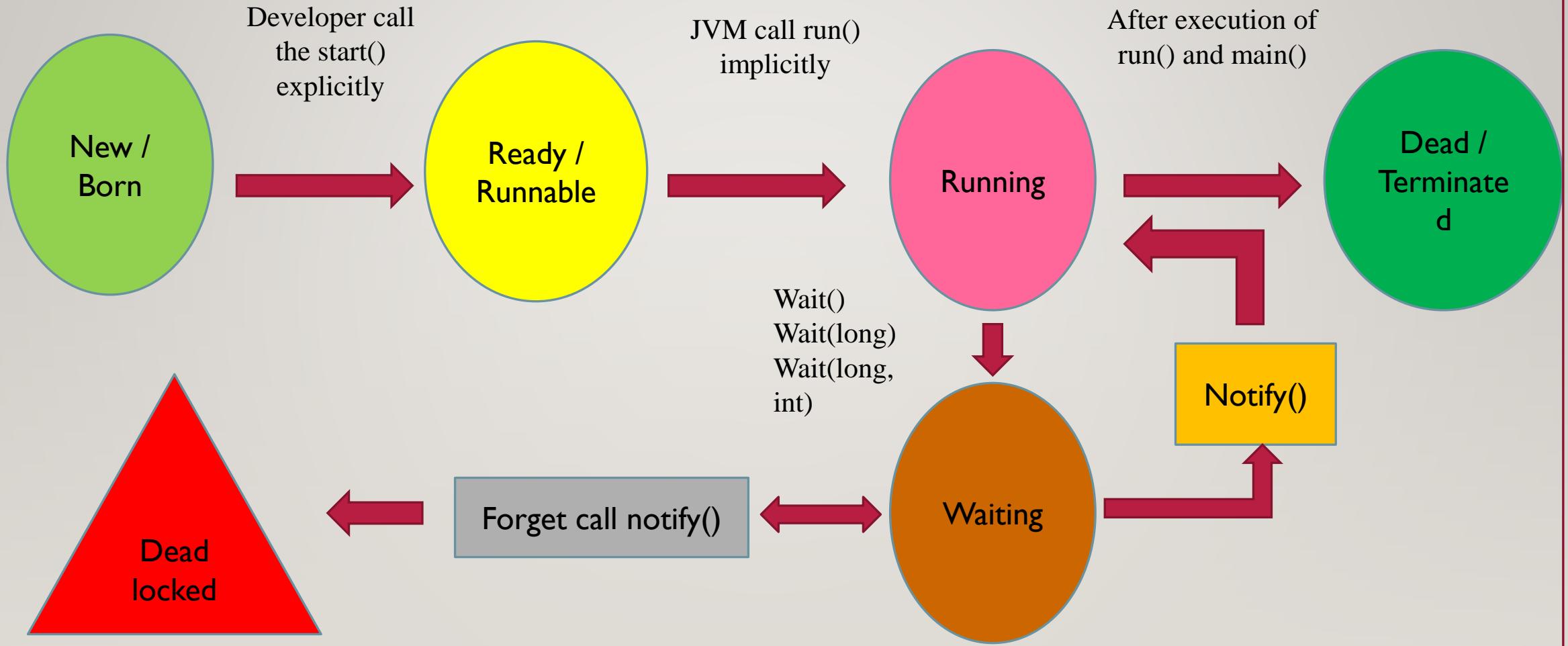
- When the developer calls the wait() method and the executing thread forgets to call notify(), at that time it will be in a dead-locked state, this process we are calling as deadlock situation.

- If the executing thread is calling the notify() method, then it will be going again back to the running stage.

### Stage 5: Dead State:

- The thread has finished executing its task or has been terminated.
- It no longer exists in the system.

## Thread life cycle :



## **Thread Property:**

Every thread in this Java world has 2 properties:

1. Name
2. Priority

## **Thread Name:**

- The default name given by the JVM is main for the main thread.
- The default name given by the JVM for user-defined threads is Thread-0, Thread-1, and so on.
- We can change the default name given by the JVM.
- Yes, we can modify/change the thread name using `setName(String newName)`.
- By using `getName()`, we can retrieve the name which we have changed.

## **Inter-Thread Communication: (Multithreading)**

- The process of making one thread to communicate with another thread is known as Inter-thread communication. Whenever we want one thread to communicate with

another thread at that time we have to make use of inter-thread communication methods.

- There are 5 methods which are used for performing the communication in two threads:

1. `wait(): void`
2. `wait(long milliseconds): void`
3. `wait(long milliseconds, int nanos): void`
4. `notify(): void`
5. `notifyAll(): void`

note :- all this method is present inside object class.

## Thread Property :

**void setName(String name)**

- Sets the name of the thread to the specified name.

**String getName()**

- Returns the name of the thread.

**void setPriority(int priority)**

- Sets the priority of the thread. The priority is an integer between Thread.MIN\_PRIORITY (1) and Thread.MAX\_PRIORITY (10).

**int getPriority()**

- Returns the priority of the thread.

**static Thread currentThread()**

- Returns a reference to the currently executing thread.

Return :

- (1) thread name
- (2) thread priority
- (3) Parent thread method name where it called.

## Synchronization

- It is a process to make single thread to access single object in other words making an object thread-safe.
- Synchronization can be achieved by a keyword known as synchronized.
- synchronized keyword is applicable only for methods and it is not applicable for variables and constructors.

Exa :2

```
package Synchronization;  
class Football {  
    public synchronized void kickFootball(String playerName) {  
        try {  
            Thread.sleep(2000);  
            System.out.println("Football came to " + playerName);  
            Thread.sleep(2000);  
            System.out.println(playerName + " is kicking the Football");  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}  
public class Player extends Thread {  
    private String playerName;  
    private Football footballObject;
```

```
public Player(String playerName, Football footballObject) {  
    this.playerName = playerName;  
    this.footballObject = footballObject;  
}  
public void run() {  
    footballObject.kickFootball(playerName);  
}  
public static void main(String[] args) {  
    Football football = new Football();  
    Player player1 = new Player("Messi", football);  
    Player player2 = new Player("Ronaldo", football);  
    player1.start();  
    player2.start();  
}
```

## Advantages and Disadvantages of Synchronization?

Feature	Advantages	Disadvantages
Thread Safety	Only one thread can access a at a time it will insure.	Can lead to performance degradation due to context switch saps.
Consistency	shared data remains consistent across multiple threads.	Can introduce performance highly concurrent applications.
Deadlock Prevention	Can be used to prevent deadlocks if implemented carefully.	Improper use of synchronization can lead to deadlocks.
Simplicity	Relatively simple to implement using the synchronized keyword.	Can be complex to use in more intricate scenarios, requiring careful consideration of locking mechanisms.

## Diff B/W wait method and sleep method?

Feature	sleep()	wait()
Class	Thread	Object
Purpose	Pauses the current thread stop execution for a specified duration.	Pauses the current thread stop execution and releases the lock on the object.
Lock	Does not release the lock on any object.	Releases the lock on the object that the notify() method was called on.
Usage	Primarily used for pausing the thread execution for a specific time, such as for implementing delays.	Primarily used for thread synchronization and communication between threads.
Example	Thread.sleep(1000); (pauses for 1 second)	object.wait(); (pauses till the notify)

## # Explanation of printing o/p in java

- **System:** This is a final class in the `java.lang` package. It provides access to system properties and environment variables.
- **out:** This is a public static field of the `System` class. It's an instance of the `PrintStream` class, which is used for writing text to an output stream, typically the console.
- **println():** This is a method of the `PrintStream` class. It takes an argument, converts it to a string, and prints it to the standard output stream (usually the console). After printing, it adds a newline character, hence the name "println" (print line).

Exa : **package** User;

**import** Lang.SystemLocal;

**public class** User {

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

    SystemLocal.outLocal.printlnLocal("hi");

}

**package** Lang;

**import** IO.PrintStreamLocal;

**public class** SystemLocal {

**public static final** PrintStreamLocal     outLocal=new

PrintStreamLocal();

}

**package** IO;

**public class** PrintStreamLocal {

**public void** printlnLocal(String message) {

        System.out.println(message);

}

## Java Bin Class

- the classes with contain getter and setter method those classes called bin class.
- All encapsulated programs are not JavaBin Class programs but all JavaBin Class programs are encapsulated.
- we can not only initialized the non-static variable by using method and by using constructor we can also use getters and setters() for initializing. non static variable

### Condition to Create setter's method :

- setters method should be public and nonstatic.
- it should have argument but it should not have return type
- this method is used for setting or modifying non-static variable.

### Condition to create getters method:

- getters method should be public and non static.
- it should have return type but it should not have argument.
- it is use for fetching/getting/returning/retrieving non static variable.

```
public class Person implements Serializable {
```

```
    private String firstName;
```

```
    private int age;
```

```
    // Default constructor
```

```
    public Person() {
```

```
}
```

```
    // Parameterized constructor
```

```
    public Person(String firstName, String lastName, int age) {
```

```
        this.firstName = firstName;
```

```
        this.age = age;
```

```
}
```

```
    // Getters and setters
```

```
    public String getFirstName() {
```

```
        return firstName;
```

```
}
```

```
public void setFirstName(String firstName) {  
    this.firstName = firstName;  
}  
  
public int getAge() {  
    return age;  
}  
  
public void setAge(int age) {  
    this.age = age;  
}
```

### Garbage Collector :

- The process of automatically cleaning memory used by objects that are not needed is known as garbage collection.
- In older languages like C++, developers were responsible for both creating and destroying objects.
- To address this issue, modern languages like Java have implemented automatic garbage collection. The garbage collector periodically scans the heap for objects that are no longer used and clean their memory.
- If objects don't have references, they are eligible for garbage collection.

### There are several ways to make an object eligible for garbage collection:

1. **Nullifying the object:** Setting the reference to the object to null.
2. **Reassigning the object:** Assigning a new object to the same reference.
3. **Creating the object inside a method:** The object becomes eligible for garbage collection when the method returns.

## Property of setter , getter way and constructor way of initialization ?

Method	Description	Advantages	Disadvantages
Constructor	Initializes instance variables during object creation.	Efficient, values are set at object creation.	Less flexible, values cannot be changed after object creation without creating a new object.
Setter Methods	Initializes or modifies instance variables after object creation.	Flexible, values can be changed dynamically.	More variable, requires multiple method calls.
Getter Methods	Retrieves the values of instance variables.	Provides access to private instance variables.	No direct initialization, primarily used for retrieval.

## Enum :

Enum is a keyword used for creating a user-defined data type. While classes can also be used to create user-defined data types, using enums is more concise and efficient for representing a fixed set of values.

By using enums, we can create our own data types and represent a group of named constants.

Enums were introduced in Java in JDK version 1.5.

### Characteristics of enum types:

1. Enum constants are implicitly public, static, and final.
2. To access enum constants from another class, use the class name.
3. When declaring enum constants, semicolons are optional after the last constant.

Exa : public enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
FRIDAY, SATURDAY, SUNDAY

}

public class EnumExample {

public static void main(String[] args) {

Day today = Day.MONDAY;

if (today == Day.MONDAY) {

System.out.println("It's Monday!");

}

// Iterate over all enum values

for (Day day : Day.values()) {

System.out.println(day);

}

}

}

## Default Method

- Default methods were introduced in Java 8.
- If a new method is added to an interface, all implementing classes must be updated to provide an implementation for the new method.
- To avoid this breaking change, we can use default methods.
- Default methods can only be declared within interfaces, not classes.
- Subclasses that implement an interface with a default method are not required to provide their own implementation.

Exa :

```
interface Shape {  
    void area();  
    default void draw() {  
        System.out.println("Drawing a shape");  
    }  
}  
  
class Circle implements Shape {  
    @Override  
    public void area() {  
        // Implementation for calculating the area of a circle  
    }  
}  
  
class Rectangle implements Shape {  
    @Override  
    public void area() {  
        // Implementation for calculating the area of a rectangle  
    }  
}
```

## File Handling :

### File:

A file is a storage medium where we can store various types of data, such as audio, video, text documents, code, and more.

### Why File Handling?

- **Data Persistence:** Local variables and objects in memory are temporary and lost when the program or method ends.
- **Data Protection:** To ensure data is preserved beyond the program's execution, we store it in files.
- **Data Management:** File handling allows us to perform operations like reading, writing, updating, and deleting data from files.

### Streams:

A stream is a sequence of data that flows from one place to another.

Java provides three standard streams:

1. **System.in:** Represents the standard input stream, typically the keyboard.
2. **System.out:** Represents the standard output stream, typically the console.

3. **System.err:** Represents the standard error stream, also typically the console.

In order to create streams, we need to use the following classes:

1. **File:** Represents a file or directory in the file system.
2. **FileInputStream:** Reads bytes from a file.
3. **OutputStream:** Writes bytes to a file.
4. **BufferedInputStream:** Reads bytes from a file with buffering for improved performance.
5. **BufferedOutputStream:** Writes bytes to a file with buffering for improved performance.
6. **ObjectInputStream:** Reads serialized objects from a file.
7. **ObjectOutputStream:** Writes serialized objects to a file.
8. **FileReader:** Reads characters from a file.
9. **FileWriter:** Writes characters to a file.

## File Creation:

- To create a new file, we use the `createNewFile()` method, which is non-static and returns a boolean value. This method is part of the `File` class.
- To access this method, we first create a `File` object and provide the desired file name with its extension in the constructor.
- The `createNewFile()` method can throw an `IOException`. It's our responsibility to handle this exception using a try-catch block.
- The method returns true if the file is created successfully, otherwise, it returns false.

## Exa :

```
File textFile = new File("TextFile.txt");

try {
    boolean isCreated = textFile.createNewFile();
    if (isCreated) {
        System.out.println("File created successfully");
    } else {
        System.out.println("File already exists");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

## File Writing :

- To write to a file, we first need to create the file.
- After creating the file, we use the write() method, which is non-static and overloaded. This method is part of the FileWriter class.
- To access the write() method, we create a FileWriter object, passing either a File object or a file name as a parameter.
- Along with the write() method, we should use the flush() and close() methods. flush() pushes the data to the underlying storage, and close() releases the resources associated with the stream. Both methods are part of the OutputStreamWriter class.

## Exa :

```
File demoFile = new File("Demo.txt");
try {
    if (demoFile.createNewFile()) {
        System.out.println("File created successfully");
        FileWriter writer = new FileWriter(demoFile);
        writer.write("Hello, world!");
        writer.flush();
        writer.close();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

## Reading from a File:

- To read from a file, we use the `read()` method, which is part of the `FileReader` class. This class inherits from `InputStreamReader`.
- The `read()` method reads a single character at a time. To read multiple characters, we typically use loops and conditional checks.
- If the character is present in the file, the `read()` method returns the character's ASCII value. If the end of the file is reached, the `read()` method returns -1.

## Exa :

```
try{  
    File demoFile = new File("Demo.txt");  
    FileReader reader = new FileReader(demoFile);  
    int character;  
    while ((character = reader.read()) != -1) {  
        System.out.print((char) character);  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

### Exa : File creation, write and read code

```
import java.io.*;  
  
public class FileOperations {  
  
    public static void main(String[] args) {  
  
        String fileName = "example.txt";  
  
        // Create a new file  
  
        File file = new File(fileName);  
  
        try {  
  
            file.createNewFile();  
  
            // Write to the file  
  
            FileWriter writer = new FileWriter(file);  
  
            writer.write("This is a line of text.\n");  
  
            writer.write("This is another line.\n");  
  
            System.out.println("File written successfully.");  
  
            // Read from the file  
  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
FileReader reader = new FileReader(file);  
  
int character;  
  
while ((character = reader.read()) != -1) {  
  
    System.out.print((char) character);  
  
}  
  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}  
}
```

## Stream API

The Stream API in Java provides a way to process collections of objects in a functional style. It allows you to perform operations like filtering, mapping, and reducing on collections and provide efficient way .

### Important Stream Operations

#### Filter

**Definition:** The filter() operation selects elements from a stream that satisfy a given condition (predicate).

#### Example (Java):

```
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;
```

```
public class FilterExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5,  
6, 7, 8, 9, 10);  
  
        List<Integer> evenNumbers = numbers.stream()  
            .filter(n -> n % 2 == 0)  
            // Keep only even numbers  
            .collect(Collectors.toList());  
  
        System.out.println(evenNumbers);  
        // Output: [2, 4, 6, 8, 10]  
    }  
}
```

## 2. Map

- **Definition:** The map() operation transforms each element in a stream into a new element using a given function.
- If we have to apply operation on each element then go with map for **Example :**

```
import java.util.Arrays;  
  
import java.util.List;  
  
import java.util.stream.Collectors;  
  
public class MapExample {  
  
    public static void main(String[] args) {  
  
        List<String> words = Arrays.asList("hello", "world", "java");  
  
        List<Integer> wordLengths = words.stream()  
            .map(String::length) // Transform each word to its length  
            .collect(Collectors.toList());  
  
        System.out.println(wordLengths); // Output: [5, 5, 4]  
    }  
}
```

## 3. Reduce

- **Definition:** The reduce() operation combines all elements in a stream into a single result using a given function.
- reduce() is combines the elements of the stream into a single result. Here a is always resultant value initially it should be 0.
- **Example (Java):**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class ReduceExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
        int sum = numbers.stream()  
            .reduce(0, (a, b) -> a + b); // Cal. sum of all numbers  
  
        System.out.println(sum); // Output: 15  
    }  
}
```

#### 4. Distinct

- Definition: The distinct() operation returns a stream with duplicate elements removed.
- Example (Java):

```
import java.util.Arrays;  
  
import java.util.List;  
  
import java.util.stream.Collectors;  
  
public class DistinctExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 4, 5);  
  
        List<Integer> distinctNumbers = numbers.stream()  
            .distinct() // Remove duplicate numbers  
            .collect(Collectors.toList());  
  
        System.out.println(distinctNumbers);  
    }  
}
```

#### 5. Sorted

- **Definition:** The sorted() operation returns a stream with elements sorted in natural order or according to a custom comparator.
- **Example (Java):**

```
import java.util.Arrays;  
  
import java.util.List;  
  
import java.util.stream.Collectors;  
  
public class SortedExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(5, 2, 1, 4, 3);  
  
        List<Integer> sortedNumbers = numbers.stream()  
            .sorted() // Sort numbers in natural order  
            .collect(Collectors.toList());  
  
        System.out.println(sortedNumbers);  
    }  
}
```

## 6. **ForEach**

- **Definition:** The forEach() operation performs an action for each element in a stream.
- **Example (Java):**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class ForEachExample {  
  
    public static void main(String[] args) {  
  
        List<String> words = Arrays.asList("hello", "world");  
  
        words.stream()  
  
            .forEach(System.out::println); // Print each word  
  
        // Output:  
  
        // hello  
  
        // world  
    }  
}
```

## 7. **Collect**

- **Definition:** The collect() operation gathers the elements of a stream into a collection (e.g., List, Set, Map).
- **Example (Java):**

```
import java.util.Arrays;  
  
import java.util.List;  
  
import java.util.stream.Collectors;  
  
public class CollectExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
        List<Integer> evenNumbers = numbers.stream()  
  
            .filter(n -> n % 2 == 0)  
  
            .collect(Collectors.toList()); // Collect even numbers  
        // into a List  
  
        System.out.println(evenNumbers); // Output: [2, 4]  
    }  
}
```

## 8. flatMap

**Definition:** The flatMap() operation is used to flatten a stream of collections into a single stream. It takes a function that returns a stream for each element in the original stream, and then concatenates all the resulting streams into one **Example:**

```
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
import java.util.stream.Stream;  
public class FlatMapExample {  
    public static void main(String[] args) {  
        List<List<Integer>> listOfLists = Arrays.asList(  
            Arrays.asList(1, 2, 3),  
            Arrays.asList(4, 5),  
            Arrays.asList(6, 7, 8, 9)  
        );
```

```
List<Integer> numbers = listOfLists.stream()
```

```
.flatMap(List::stream)  
.collect(Collectors.toList());
```

```
System.out.println(numbers);  
// Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

// Example with String arrays

```
List<String[]> arrayOfArrays = Arrays.asList(  
    new String[]{"hello", "world"},  
    new String[]{"java", "streams"}  
)
```

```
List<String> words = arrayOfArrays.stream()  
.flatMap(Arrays::stream)  
.collect(Collectors.toList());
```

```
System.out.println(words); // Output: [hello, world, java,  
streams]  
}}
```

## 9. findFirst

**Definition:** The `findFirst()` operation returns an `Optional` containing the first element of the stream, or an empty `Optional` if the stream is empty.

**Example (Java):**

```
import java.util.Arrays;  
import java.util.List;  
import java.util.Optional;  
  
public class FindFirstExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
        Optional<Integer> firstEven = numbers.stream()  
            .filter(n -> n % 2 == 0)  
            .findFirst();
```

```
System.out.println(firstEven.orElse(-1));
```

// Output: 2 (or -1 if no even number is found)

```
List<Integer> emptyList = Arrays.asList();  
Optional<Integer> firstOfEmpty =  
emptyList.stream().findFirst();  
System.out.println(firstOfEmpty.orElse(-1));  
// Output: -1  
}
```

## 10. findAny

- **Definition:** The findAny() operation returns an Optional containing any element of the stream. It's useful in parallel streams where finding the absolute first element might be computationally expensive, **Example :**

```
import java.util.Arrays;  
import java.util.List;  
import java.util.Optional;  
public class FindAnyExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
        Optional<Integer> anyEven = numbers.stream()  
            .filter(n -> n % 2 == 0)  
            .findAny();  
        System.out.println(anyEven.orElse(-1));  
    }  
}
```

## 11. anyMatch

- **Definition:** The anyMatch() operation returns true if any element in the stream matches the given predicate, otherwise false, **Example :**

```
import java.util.Arrays;  
import java.util.List;  
public class AnyMatchExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
        boolean hasEven = numbers.stream()  
            .anyMatch(n -> n % 2 == 0);  
        System.out.println(hasEven); // Output: true  
        boolean hasGreaterThan10 =  
            numbers.stream().anyMatch(n -> n > 10);  
        System.out.println(hasGreaterThan10); // Output: false  
    }  
}
```

## 12. allMatch

**Definition:** The allMatch() operation returns true if all elements in the stream match the given predicate, otherwise false, **Example in Java:**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class AllMatchExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(2, 4, 6, 8, 10);  
  
        boolean allEven = numbers.stream()  
            .allMatch(n -> n % 2 == 0);  
  
        System.out.println(allEven);  
  
        List<Integer> mixedNumbers = Arrays.asList(2, 3, 4, 6, 8, 10);  
  
        boolean allEvenMixed =  
mixedNumbers.stream().allMatch(n -> n % 2 == 0);  
  
        System.out.println(allEvenMixed); // Output: false  
  
    }  
}
```

## 13. noneMatch

**Definition:** The noneMatch() operation returns true if no elements in the stream match the given predicate, otherwise false, **Example :**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class NoneMatchExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 9);  
  
        boolean noEven = numbers.stream()  
            .noneMatch(n -> n % 2 == 0);  
  
        System.out.println(noEven); // Output: true  
  
        List<Integer> mixedNumbers = Arrays.asList(1, 3, 4, 5, 7, 9);  
  
        boolean noEvenMixed =  
mixedNumbers.stream().noneMatch(n -> n % 2 == 0);  
  
        System.out.println(noEvenMixed); // Output: false  
  
    }  
}
```

## 13. count

- **Definition:** The count() operation returns the number of elements in the stream, **Example :**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class CountExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
        long evenCount = numbers.stream()  
  
            .filter(n -> n % 2 == 0)  
  
            .count();  
  
        System.out.println(evenCount); // Output: 2  
    }  
}
```

## 14. min and max

- **Definition:** The min() and max() operations return an Optional containing the minimum or maximum element

in the stream according to a given comparator, **Example**

```
import java.util.Arrays;  
  
import java.util.Comparator;  
  
import java.util.List;  
  
import java.util.Optional;  
  
public class MinMaxExample {  
  
    public static void main(String[] args) {  
  
        List<Integer> numbers = Arrays.asList(5, 2, 1, 4, 3);  
  
        Optional<Integer> min = numbers.stream()  
  
            .min(Comparator.naturalOrder());  
  
        System.out.println(min.orElse(-1)); // Output: 1  
  
        Optional<Integer> max = numbers.stream()  
  
            .max(Comparator.naturalOrder());  
  
        System.out.println(max.orElse(-1)); // Output: 5  
    }  
}
```

## Arrays:

class provides static methods to manipulate arrays in Java. Here's a breakdown of the important methods with examples:

### 1. asList()

- Definition: Returns a fixed-size list backed by the specified array. Changes to the list will affect the array and vice-versa.
- Example:

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class AsListExample {  
  
    public static void main(String[] args) {  
  
        String[] array = {"apple", "banana", "orange"};  
  
        List<String> list = Arrays.asList(array);  
  
        System.out.println(list); // Output: [apple, banana,  
orange]}
```

```
list.set(0, "grape"); // Modifying the list
```

```
System.out.println(Arrays.toString(array)); // Output:  
[grape, banana, orange] (array is also changed)
```

```
// list.add("kiwi"); // This will throw  
UnsupportedOperationException because the list is fixed-size.  
}  
}
```

## 2. `toString()`

- **Definition:** Returns a string representation of the contents of the specified array.
- **Example:**

```
import java.util.Arrays;  
  
public class ToStringExample {  
  
    public static void main(String[] args) {  
  
        int[] numbers = {1, 2, 3, 4, 5};  
  
        System.out.println(Arrays.toString(numbers));  
        // Output: [1, 2, 3, 4, 5]  
  
        String[] fruits = {"apple", "banana"};  
  
        System.out.println(Arrays.toString(fruits));  
        // Output: [apple, banana]  
    }  
}
```

## 3. `deepToString()`

- **Definition:** Returns a string representation of the "deep contents" of the specified array. Use this for multidimensional arrays.

- **Example:**

```
import java.util.Arrays;  
  
public class DeepToStringExample {  
  
    public static void main(String[] args) {  
  
        int[][] matrix = {{1, 2}, {3, 4, 5}};  
  
        System.out.println(Arrays.deepToString(matrix));  
        // Output: [[1, 2], [3, 4, 5]]  
  
        String[][] names = {{"Mr.", "Smith"}, {"Ms.",  
                            "Jones"}};  
  
        System.out.println(Arrays.deepToString(names));  
        // Output: [[Mr., Smith], [Ms., Jones]]  
    }  
}
```

#### 4. equals() and deepEquals()

- **Definition:** equals() checks if two arrays are equal (same length and same elements in the same order).  
deepEquals() does the same for multidimensional arrays.
- **Example:**

```
import java.util.Arrays;

public class EqualsExample {

    public static void main(String[] args) {

        int[] arr1 = {1, 2, 3};
        int[] arr2 = {1, 2, 3};
        int[] arr3 = {3, 2, 1};
        int[] arr4 = {1, 2, 3, 4};

        System.out.println(Arrays.equals(arr1, arr2));
        // Output: true
    }
}
```

```
System.out.println(Arrays.equals(arr1, arr3)); // Output:  
false
```

```
System.out.println(Arrays.equals(arr1, arr4)); //  
Output: false
```

```
int[][] matrix1 = {{1, 2}, {3, 4}};
int[][] matrix2 = {{1, 2}, {3, 4}};
int[][] matrix3 = {{1, 2}, {4, 3}};
```

```
System.out.println(Arrays.deepEquals(matrix1,
matrix2)); // Output: true
```

```
System.out.println(Arrays.deepEquals(matrix1,
matrix3)); // Output: false
```

```
}
```

```
}
```

## 5. fill()

- **Definition:** Assigns the specified value to each element of the specified array.
- **Example:**

```
import java.util.Arrays;  
  
public class FillExample {  
  
    public static void main(String[] args) {  
  
        int[] numbers = new int[5];  
  
        Arrays.fill(numbers, 10);  
  
        System.out.println(Arrays.toString(numbers));  
        // Output: [10, 10, 10, 10, 10]  
  
    }  
}
```

## 6. sort()

- **Definition:** Sorts the specified array into ascending order. There are overloaded versions for different data types and for sorting a range within the array, **Example:**

```
import java.util.Arrays;  
  
public class SortExample {  
  
    public static void main(String[] args) {  
  
        int[] numbers = {5, 2, 8, 1, 9, 4};  
  
        Arrays.sort(numbers);  
  
        System.out.println(Arrays.toString(numbers));  
  
        // Output: [1, 2, 4, 5, 8, 9]  
  
        String[] fruits = {"orange", "apple", "banana"};  
  
        Arrays.sort(fruits);  
  
        System.out.println(Arrays.toString(fruits)); // Output:  
        [apple, banana, orange]  
    }  
}
```

## 7. binarySearch()

- **Definition:** Searches for the specified value in the specified *sorted* array using the binary search algorithm. Returns the index of the search key, if it is contained in the array; otherwise,  $(-\text{insertion point}) - 1$ .
- **Example:**

```
import java.util.Arrays;

public class BinarySearchExample {

    public static void main(String[] args) {

        int[] numbers = {1, 2, 4, 5, 8, 9}; // Must be sorted!

        int index = Arrays.binarySearch(numbers, 5);

        System.out.println(index); // Output: 3

        int notFound = Arrays.binarySearch(numbers, 6);

        System.out.println(notFound);

        // Output: -5 (-4+1) where 4 would be the insertion
        // point)
    }
}
```

```
String[] fruits = {"apple", "banana", "orange"};

int fruitIndex = Arrays.binarySearch(fruits, "banana");
// Must be sorted!

System.out.println(fruitIndex); // Output: 1

}
```

## 7. binarySearch()

- **Definition:** Searches for the specified value in the specified *sorted* array using the binary search algorithm. Returns the index of the search key, if it is contained in the array; otherwise,  $(-\text{insertion point}) - 1$ .
- **Example:**

```
import java.util.Arrays;

public class BinarySearchExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 4, 5, 8, 9}; // Must be sorted!
        int index = Arrays.binarySearch(numbers, 5);
        System.out.println(index); // Output: 3

        int notFound = Arrays.binarySearch(numbers, 6);
    }
}
```

System.out.println(notFound); // Output: -5 ( $-(4+1)$  where 4 would be the insertion point)

```
String[] fruits = {"apple", "banana", "orange"};
int fruitIndex = Arrays.binarySearch(fruits, "banana");
// Must be sorted!

System.out.println(fruitIndex); // Output: 1
}
```

## 8. copyOf() and copyOfRange()

- **Definition:** copyOf() creates a new array that is a copy of the specified array, truncating or padding with nulls (if needed) to obtain the specified length. copyOfRange() copies a specific range of the original array.
- **Example:**

```
import java.util.Arrays;

public class CopyOfExample {
    public static void main(String[] args) {
        int[] original = {1, 2, 3, 4, 5};
        int[] copy1 = Arrays.copyOf(original, 3); // Copy
        first 3 elements
        System.out.println(Arrays.toString(copy1)); //
        Output: [1, 2, 3]
```

```
int[] copy2 = Arrays.copyOf(original, 7); // Pad with 0s
System.out.println(Arrays.toString(copy2));
// Output: [1, 2, 3, 4, 5, 0, 0]
int[] copyRange = Arrays.copyOfRange(original, 1, 4);
// Copy from index 1 (inclusive) to 4 (exclusive)
System.out.println(Arrays.toString(copyRange));
// Output: [2, 3, 4]
}
```

## Modifying Collections

- **addAll(Collection<? super T> c, T... elements):** Adds all of the specified elements to the specified collection.

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

```
public class AddAllExample {  
  
    public static void main(String[] args) {  
  
        List<String> list = new ArrayList<>();  
  
        Collections.addAll(list, "apple", "banana",  
"orange");  
  
        System.out.println(list);  
  
        // Output: [apple, banana, orange]  
    }  
}
```

- **fill(List<? super T> list, T obj):** Replaces all of the elements of the specified list with the specified element.

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

```
public class FillExample {  
  
    public static void main(String[] args) {  
  
        List<String> list = new  
ArrayList<>(Arrays.asList("a", "b", "c"));  
  
        Collections.fill(list, "x");  
  
        System.out.println(list); // Output: [x, x, x]  
    }  
}
```

**copy(List<? super T> dest, List<? extends T> src):** Copies all of the elements from one list into another. The destination list must be at least as long as the source list.

```
import java.util.*;

public class CopyExample {

    public static void main(String[] args) {
        List<String> src = Arrays.asList("apple", "banana");
        List<String> dest = new
        ArrayList<>(Arrays.asList("1", "2", "3"));
        // Destination must be large enough
        Collections.copy(dest, src);
        System.out.println(dest); // Output: [apple, banana, 3]
    }
}
```

- **swap(List<?> list, int i, int j):** Swaps the elements at the specified positions in the specified list.

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

```
public class SwapExample {

    public static void main(String[] args) {
        List<String> list = new
        ArrayList<>(Arrays.asList("a", "b", "c"));
        Collections.swap(list, 0, 2);
        System.out.println(list); // Output: [c, b, a]
    }
}
```

- **reverse(List<?> list):** Reverses the order of the elements in the specified list.

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

```
public class ReverseExample {  
    public static void main(String[] args) {  
        List<String> list = new  
        ArrayList<>(Arrays.asList("a", "b", "c"));  
        Collections.reverse(list);  
        System.out.println(list); // Output: [c, b, a]  
    }  
}
```

- **rotate(List<?> list, int distance):** Rotates the elements in the specified list by the specified distance.

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

```
public class RotateExample {  
    public static void main(String[] args) {  
        List<String> list = new  
        ArrayList<>(Arrays.asList("a", "b", "c", "d", "e"));  
        Collections.rotate(list, 2); // Rotate right by 2  
        System.out.println(list); // Output: [d, e, a, b, c]  
    }  
}
```

**shuffle(List<?> list):** Randomly permutes the specified list using a default source of randomness.

**shuffle(List<?> list, Random rnd):** Randomly permutes the specified list using the specified source of randomness.

```
import java.util.*;

public class ShuffleExample {

    public static void main(String[] args) {
        List<Integer> list = new
        ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
        Collections.shuffle(list);
        System.out.println(list); // Output: (random
        permutation of the list)

        Collections.shuffle(list, new Random(42)); // Using
        a specific seed for reproducibility

        System.out.println(list); // Output: (a specific
        random permutation)
    }
}
```

**sort(List list):** Sorts the specified list into ascending order, according to the natural ordering of its elements.

**sort(List list, Comparator c):** Sorts the specified list according to the order induced by the specified comparator.

```
import java.util.*;

public class SortExample {

    public static void main(String[] args) {
        List<Integer>
list = new ArrayList<>(Arrays.asList(5, 2, 8, 1));
        Collections.sort(list);
        System.out.println(list); // Output: [1, 2, 5, 8]

        List<String> strings = new
        ArrayList<>(Arrays.asList("zebra", "apple", "Banana"));

        Collections.sort(strings,
        String.CASE_INSENSITIVE_ORDER);

        System.out.println(strings); // Output: [apple, Banana,
        zebra]
    }
}
```

## Searching and Finding

### **binarySearch(List<? extends Comparable<? super T>>**

**list, T key):** Searches the specified list for the specified object using the binary search algorithm. The list *must* be sorted prior to making this call.

### **binarySearch(List<? extends T> list, T key,**

**Comparator<? super T> c):** Searches the specified list for the specified object using the binary search algorithm. The list *must* be sorted according to the specified comparator prior to making this call.

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

```
public class BinarySearchExample {  
    public static void main(String[] args) {  
        List<Integer> list = Arrays.asList(1, 3, 5, 7, 9); // Must  
        be sorted!  
        int index = Collections.binarySearch(list, 5);  
        System.out.println(index); // Output: 2
```

```
List<String> strings = Arrays.asList("apple", "banana",  
"zebra"); // Must be sorted!
```

```
int stringIndex = Collections.binarySearch(strings,  
"banana");
```

```
System.out.println(stringIndex); // Output: 1
```

```
}
```

```
}
```

- **min(Collection<? extends T> coll):** Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
- **min(Collection<? extends T> coll, Comparator<? super T> comp):** Returns the minimum element of the given collection, according to the order induced by the specified comparator.
- **max(Collection<? extends T> coll):** Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
- **max(Collection<? extends T> coll, Comparator<? super T> comp):** Returns the maximum element of the given collection, according to the order induced by the specified comparator.

```
import java.util.*;

public class MinMaxExample {

    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    }
}
```

```
System.out.println(Collections.min(numbers)); // Output: 1

System.out.println(Collections.max(numbers));
// Output: 9

List<String> strings = Arrays.asList("zebra", "apple",
"Banana");

System.out.println(Collections.min(strings,
String.CASE_INSENSITIVE_ORDER));
// Output: apple

System.out.println(Collections.max(strings,
String.CASE_INSENSITIVE_ORDER));
// Output: zebra

}
```

## Creating Special Collections

**emptyList(), emptySet(), emptyMap():** Returns empty, immutable lists, sets, and maps, respectively.

```
import java.util.*;

public class EmptyCollectionsExample {

    public static void main(String[] args) {

        List<String> emptyList = Collections.emptyList();

        System.out.println(emptyList); // Output: []

        // emptyList.add("test");

        // Throws UnsupportedOperationException

        Set<Integer> emptySet = Collections.emptySet();

        System.out.println(emptySet); // Output: []

        Map<String, Double> emptyMap = 

        Collections.emptyMap();

        System.out.println(emptyMap); // Output: {}

        // Trying to add elements will result in UnsupportedOperationException
    }
}
```

```
try {

    emptyList.add("test");

} catch (UnsupportedOperationException e) {

    System.out.println("Caught      exception:      " + e.getMessage());

}

try {

    emptySet.add(1);

} catch (UnsupportedOperationException e) {

    System.out.println("Caught      exception:      " + e.getMessage());

}

try {

    emptyMap.put("key", 1.0);

} catch (UnsupportedOperationException e) {

    System.out.println("Caught      exception:      " + e.getMessage());

}
}}
```

## Comparable and Comparator

Both are interfaces in Java used for sorting objects. They define ways to compare two objects and determine their relative order and they have different purposes and usage:

### 1. Comparable

**Definition:** The Comparable interface is implemented by a class whose objects need to be ordered "naturally." It defines a *natural ordering* for objects of that class. It has a single method:

**int compareTo(T o);**

- This method compares the current object (this) with the specified object o.
- It returns:
  - A negative integer if this is less than o.
  - Zero if this is equal to o.
  - A positive integer if this is greater than o.

**Usage:** When a class implements Comparable, its objects can be sorted directly using methods like Collections.sort() or Arrays.sort().

#### Example:

```
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
class Student implements Comparable<Student> {  
  
    String name;  
  
    int rollNumber;  
  
    public Student(String name, int rollNumber) {  
  
        this.name = name;  
  
        this.rollNumber = rollNumber;  
    }  
}
```

```
@Override
```

```
public int compareTo(Student other) {  
    // Compare based on roll number (natural ordering)  
    return this.rollNumber - other.rollNumber; // Ascending  
order  
    // return other.rollNumber - this.rollNumber; //  
Descending order  
}
```

```
@Override
```

```
public String toString() {  
    return "Name: " + name + ", Roll Number: " +  
rollNumber;  
}
```

```
public class ComparableExample {
```

```
    public static void main(String[] args) {  
        List<Student> students = new ArrayList<>();  
        students.add(new Student("Alice", 10));  
        students.add(new Student("Bob", 5));  
        students.add(new Student("Charlie", 15));  
        Collections.sort(students);  
        // Sorts based on compareTo()  
        for (Student student : students) {  
            System.out.println(student);  
        }  
        // Output: // Name: Bob, Roll Number: 5// Name:  
Alice, Roll Number: 10 // Name: Charlie, Roll Number:  
15  
    }}
```

## 2. Comparator

- Definition: The Comparator interface defines a comparison function that does *not* require the objects being compared to implement Comparable. It's used to define custom sorting logic or provide multiple ways to sort objects of the same class. It has a single method:

```
int compare(T o1, T o2);
```

- This method compares the two specified objects o1 and o2.
- It returns:
  - A negative integer if o1 is less than o2.
  - Zero if o1 is equal to o2.
  - A positive integer if o1 is greater than o2.
- Usage: Comparator is used with methods like Collections.sort(list, comparator) or Arrays.sort(array, comparator).

- Example:

```
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
  
class Student { // Doesn't implement Comparable now  
  
    String name;  
  
    int rollNumber;  
  
    public Student(String name, int rollNumber) {  
  
        this.name = name;  
  
        this.rollNumber = rollNumber;  
    }  
  
    @Override  
  
    public String toString() {  
  
        return "Name: " + name + ", Roll Number: " + rollNumber;  
    } }
```

```
public class ComparatorExample {  
    public static void main(String[] args) {  
        List<Student> students = new ArrayList<>();  
        students.add(new Student("Alice", 10));  
        students.add(new Student("Bob", 5));  
        students.add(new Student("Charlie", 15));  
        // Comparator to sort by name  
        Comparator<Student> nameComparator =  
            Comparator.comparing(s -> s.name);  
        Collections.sort(students, nameComparator);  
        System.out.println("Sorted by Name:");  
        for (Student student : students) {  
            System.out.println(student);  
        }  
    }  
}
```

```
// Output:  
  
// Sorted by Name:// Name: Alice, Roll Number: 10  
// Name: Bob, Roll Number: 5// Name: Charlie, Roll  
Number: 15  
  
// Comparator to sort by roll number in descending order  
Comparator<Student> rollNumberComparatorDescending =  
    Comparator.comparingInt(s -> s.rollNumber);  
  
Collections.sort(students,  
    rollNumberComparatorDescending);  
  
System.out.println("Sorted by Roll Number Descending:");  
  
for (Student student : students) {  
    System.out.println(student);  
}  
  
// Output:// Sorted by Roll Number Descending:  
// Name: Charlie, Roll Number: 15// Name: Alice, Roll  
Number: 10// Name: Bob, Roll Number: 5  
}  
}
```

## Diff b/w Comparator and Comparable?

Feature	Comparable	Comparator
Interface	java.lang.Comparable	java.util.Comparator
Method	int compareTo(T o)	int compare(T o1, T o2)
Implemented by	The class whose objects are to be sorted	A separate class or lambda expression
Purpose	Defines natural ordering	Defines custom ordering or multiple orderings
Sorting Method	Collections.sort(list) or Arrays.sort(array)	Collections.sort(list, comparator) or Arrays.sort(array, comparator)

## Serialization

It is the process of converting an object's state into a byte stream.

## Deserialization

It is the reverse process: reconstructing an object from a byte stream.

This is useful for objects to storage (like files or databases) or transmitting them over a network.

### example of serialization and deserialization:

```
import java.io.*;  
  
class Person implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    // Important for versioning  
  
    String name;  
  
    int age;  
  
    transient String secret;  
  
    // This field will not be serialized
```

```
public Person(String name, int age, String secret) {
```

```
    this.name = name;
```

```
    this.age = age;
```

```
    this.secret = secret;
```

```
}
```

```
@Override
```

```
public String toString() {
```

```
    return "Name: " + name + ", Age: " + age + ", Secret: " +  
secret;
```

```
}
```

```
}
```

```
public class SerializationExample {
```

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

```
        Person person = new Person("Alice", 30, "My super  
secret!");
```

```

// Serialization

try (FileOutputStream fileOut = new
FileOutputStream("person.ser");
ObjectOutputStream out = new
ObjectOutputStream(fileOut)) {
    out.writeObject(person);
    System.out.println("Object serialized successfully.");
} catch (IOException i) {
    i.printStackTrace();
}

// Deserialization

Person serializedPerson = null;
try (FileInputStream fileIn = new
FileInputStream("person.ser");
ObjectInputStream in = new
ObjectInputStream(fileIn)) {

```

```

deserializedPerson = (Person) in.readObject();
System.out.println("Object deserialized successfully.");
} catch (IOException i) {
    i.printStackTrace();
} catch (ClassNotFoundException c) {
    System.out.println("Person class not found");
    c.printStackTrace();
}
if (deserializedPerson != null) {
    System.out.println("Deserialized object: " +
deserializedPerson);
    // Output: Deserialized object: Name: Alice, Age: 30,
    Secret: null (secret is transient)
}}
```

## Explanation and Key Points:

1. Present in side `java.io.Serializable` interface.
2. This is a marker interface (it has no methods) that indicates that objects of this class can be serialized.
3. **serialVersionUID:** The private static final long `serialVersionUID = 1L;` field is crucial for versioning. It's a unique identifier for the class version. If you modify the class structure (e.g., add or remove fields), you should change the `serialVersionUID` to avoid `InvalidClassException` during deserialization of objects serialized with an older version of the class. If you don't explicitly declare `serialVersionUID`, the JVM calculates it based on the class structure, which can lead to issues if the class is modified.

4. **transient Keyword:** The `transient` keyword is used to mark fields that should *not* be serialized. In the example, the `secret` field is marked as `transient`, so it's not saved to the file, and its value is null after deserialization. This is useful for sensitive information that shouldn't be persisted.
5. **ObjectOutputStream:** The `ObjectOutputStream` is used to write objects to an output stream (in this case, a file stream). The `writeObject()` method serializes the object.
6. **ObjectInputStream:** The `ObjectInputStream` is used to read objects from an input stream. The `readObject()` method deserializes the object. It's important to cast the returned object to the correct type.

## Java 8 Features:

### 1. Lambda Expressions

- **Definition:** Lambda expressions provide a concise way to represent anonymous functions (functions without a name). They enable functional programming in Java.
- **Example:**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class LambdaExample {  
    public static void main(String[] args) {  
        List<String> strings = Arrays.asList("apple",  
        "banana", "kiwi");  
    }  
}
```

```
// Using lambda expression to print each string
```

```
strings.forEach(s -> System.out.println(s));
```

```
// Lambda expression with multiple statements
```

```
strings.forEach(s -> {  
    String upperCase = s.toUpperCase();  
    System.out.println(upperCase);  
});
```

```
// Lambda expression for comparing strings (used in  
sorting)
```

```
strings.sort((s1, s2) -> s1.compareToIgnoreCase(s2));  
System.out.println(strings);  
}  
}
```

## 2. Functional Interfaces

- **Definition:** Functional interfaces are interfaces with only one abstract method. They can be used with lambda expressions. The `@FunctionalInterface` annotation can be used to explicitly mark an interface as functional.
- **Example:**

```
@FunctionalInterface  
  
interface MyFunction<T, R> {  
    R apply(T t);  
}  
  
public class FunctionalInterfaceExample {  
    public static void main(String[] args) {  
        MyFunction<String, Integer> stringLength = s ->  
            s.length();  
  
        int length = stringLength.apply("Hello");  
  
        System.out.println(length); // Output: 5  
    }  
}
```

## 3. Method References

- **Definition:** Method references are a shorthand for lambda expressions that refer to existing methods.

- **Example:**

```
import java.util.Arrays;  
  
import java.util.List;  
  
public class MethodReferenceExample {  
    public static void main(String[] args) {  
        List<String> strings = Arrays.asList("apple",  
            "banana", "kiwi");  
  
        // Method reference to System.out::println  
        strings.forEach(System.out::println);  
  
        // Method reference to String::toUpperCase  
        strings.stream().map(String::toUpperCase).forEach(System  
            .out::println);  
    }  
}
```

## 4. Stream API

- **Definition:** The Stream API provides a way to process collections of data in a functional style, allowing for operations like filtering, mapping, and reducing.
- **Example:**

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5,
6, 7, 8, 9, 10);
        List<Integer> evenNumbers = numbers.stream()
            .filter(n -> n % 2 == 0)
            .collect(Collectors.toList());
        int sumOfSquares = numbers.stream()
```

```
.map(n -> n * n)
.reduce(0, Integer::sum);
System.out.println(evenNumbers);
System.out.println(sumOfSquares);
}
```

## 5. Default Methods in Interfaces

- **Definition:** Default methods allow you to add new methods to interfaces without breaking existing implementations.
- **Example:**

```
interface MyInterface {  
    void myMethod();  
  
    default void defaultMethod() {  
        System.out.println("Default implementation");  
    }  
}  
  
class MyClass implements MyInterface {  
    @Override  
    public void myMethod() {  
        System.out.println("My implementation");  
    }  
}
```

```
public class DefaultMethodExample {  
  
    public static void main(String[] args) {  
  
        MyClass obj = new MyClass();  
  
        obj.myMethod();      // Output: My implementation  
  
        obj.defaultMethod(); // Output: Default implementation  
    }  
}
```

## 6. Static Methods in Interfaces

- **Definition:** Java 8 allows static methods in interfaces, similar to classes.
- **Example:**

```
interface MyInterface {  
    static void staticMethod() {  
        System.out.println("Static method in interface");  
    }  
}  
  
public class StaticMethodInterfaceExample {  
    public static void main(String[] args) {  
        MyInterface.staticMethod(); // Output: Static  
        method in interface  
    }  
}
```

a

## 7. Date and Time API (java.time)

- **Definition:** A new Date and Time API was introduced to address the shortcomings of the old Date and Calendar classes.
- **Example:**

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateTimeExample {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        LocalTime now = LocalTime.now();
        LocalDateTime currentDateTime =
LocalDateTime.now();
```

```
System.out.println("Today's Date: " + today);
System.out.println("Current Time: " + now);
System.out.println("Current Date and Time: " +
currentDateTime);

DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
String formattedDateTime =
currentDateTime.format(formatter);
System.out.println("Formatted Date and Time: " +
formattedDateTime);
}
```

## 8. Optional

- **Definition:** The Optional class is a container object that may or may not contain a non-null value. It helps avoid NullPointerExceptions.
- **Example:**

```
import java.util.Optional;

public class OptionalExample {
    public static void main(String[] args) {
        String str = "Hello";
        Optional<String> optionalStr =
Optional.ofNullable(str);

        if (optionalStr.isPresent()) {
            System.out.println(optionalStr.get());
        }
    }
}
```

```
optionalStr.ifPresent(s ->
System.out.println(s.toUpperCase()));

String orElseValue = optionalStr.orElse("Default
Value");

System.out.println(orElseValue);

Optional<String> emptyOptional =
Optional.ofNullable(null);

String orElseEmpty = emptyOptional.orElse("Default
Value for Empty");

System.out.println(orElseEmpty);

}
```

# Course : Java

# Part : Advance JAVA



## Advanced Java

Java is used to develop applications.

**To develop an application using Java, you can use the following approaches:**

### 1. Standalone Application:

- A standalone application is a self-contained program that runs on a single computer.
- Examples include console applications, GUI applications, and desktop applications.

### 2. Web Application:

- A web application is an application that runs on a web server and is accessed through a web browser.
- It interacts directly with the user.

### 3. Enterprise Application:

- An enterprise application is a large-scale application that typically runs on a server with multiple clients.
- It often uses a distributed architecture.

## Components of a Web Application

A web application typically consists of two main components:

### 1. Front-end:

- It is User interface which is visible to the user.
- It is primarily built using HTML, CSS, and JavaScript.

### 2. Back-end:

- The back-end handles the server-side logic and data processing.
- It typically involves databases, servers, and application logic written in languages like Java, Python, or Node.js.

## Types of Web Applications

### 1. Static Web Applications:

- These applications doesn't change dynamically.
- They are primarily HTML and CSS-based.

- **Client-side Web Applications:**

- **Server-side Web Applications:**

## Java Editions

There are four main editions of Java:

**Java SE (Standard Edition):** Used for developing standalone applications.

**Java EE (Enterprise Edition):** Used for developing web and enterprise applications.

**Java ME (Micro Edition):** Used for developing applications for embedded devices.

**JavaFX:** Used for developing rich desktop applications.

## Perspectives

Perspectives are used to specify the development environment to the specific type of application you're building.

There are two main perspectives in Java development environments:

**Java Perspective:** Primarily used for Java SE development.

**Java EE Perspective:** Primarily used for Java EE development.

## There are two main types of classes in advanced Java:

- Normal Class:** A regular Java class that can be executed from the main method.
- Servlet Class:** A specialized class that extends the HttpServlet class and is used to handle HTTP requests and generate dynamic web content

### steps to create servlet class

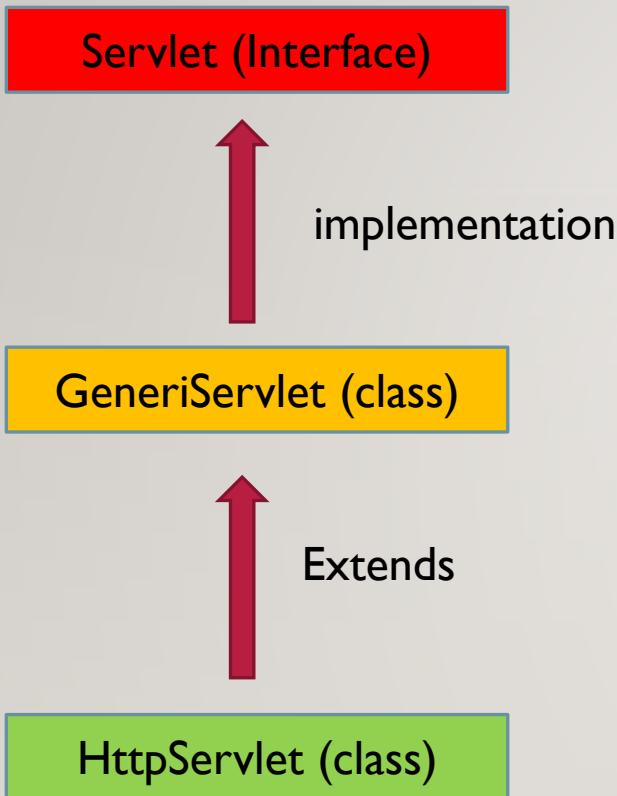
create a normal class in the project.

make a class to behave as implementation class (or) subclass for predefined servlet classes and interface.

All predefined classes are present in javax.servlet package  
javax.servlet package is present in a jar file called servlet-api.jar

servlet-api.jar file is present inside Tomcat server

## Servlet Hierarchy :



To create a servlet class by implementing the Servlet interface, you need to override all of its methods:

The Servlet interface defines five methods that must be implemented by any class that implements it:

1. **service():** This method is called by the servlet container to handle incoming requests. It's typically overridden to process requests and generate responses.
2. **destroy():** This method is called by the servlet container when the servlet is being unloaded. It's used to release resources and perform any necessary cleanup.
3. **getServletInfo():** This method returns information about the servlet, such as its author, version, and description.
4. **init():** This method is called by the servlet container when the servlet is first loaded. It's used to initialize the servlet and any resources it needs.
5. **getServletConfig():** This method returns a ServletConfig object, which provides information about the servlet's configuration.

**Exa :**

```
public class MyServlet implements Servlet {  
    public void service(ServletRequest req, ServletResponse res)  
        throws ServletException, IOException {}  
  
    public void destroy() {  
        // Clean up resources  
    }  
  
    public String getServletInfo() {  
        return "My Servlet";  
    }  
  
    public void init(ServletConfig config) throws  
        ServletException {  
        // Initialize the servlet  
    }  
  
    public ServletConfig getServletConfig() {  
        // Return the servlet configuration  
    } }
```

## GenericServlet

To create a Servlet class by using the GenericServlet class, you need to override only one method:

The GenericServlet class is an abstract class that provides a basic implementation of the Servlet interface. It has one abstract method, service(), which you must override to handle requests.

### Exa :

```
public class MyServlet extends GenericServlet {  
    public void service(ServletRequest req,  
                        ServletResponse res) throws ServletException,  
                        IOException {  
        // Implement the logic to handle the request and  
        // generate the response  
    }  
}
```

## HttpServlet :

If we want to create a Servlet class by using HttpServlet class we no need to complete any method.

HttpServlet class is an abstract class which is present with all complete methods

### Exa :

```
public class Myservlet extends HttpServlet {  
    // no need to complete any method.  
}
```

## PrintWriter

- The PrintWriter class is a predefined class in the java.io package.
- It is used to write text to a character-output stream.
- To print output, we use two methods:
  - 1.print(): Prints the specified string without a newline.
  - 2.println(): Prints the specified string followed by a newline.
- Both print() and println() are non-static methods, so they must be called on an instance of the PrintWriter class.
- To create a PrintWriter object, we typically use the getWriter() method from a HttpServletResponse object.

## Syntax:

```
PrintWriter out = resp.getWriter();
```

The getWriter() method takes no arguments and returns a PrintWriter object. This PrintWriter object can then be used to write text to the response.

## Using PrintWriter:

```
out.print("Hello, world!");  
out.println("This is a new line.");
```

## Fetching Data from Front-end:

In the front-end (HTML), we use input tags to capture user input. Each input tag should have a unique name attribute to identify the data it represents.

## Example:

```
<form action="your_servlet" method="post">  
<input type="text" name="username">  
<input type="password" name="password">  
<input type="submit" value="Submit"></form>
```

## Back-end

To fetch data from the front-end in the back-end, we use the getParameter() method.

The getParameter() method is a non-static method present in the ServletRequest interface. It takes a string argument representing the name of the parameter and returns the corresponding value as a string.

## Syntax:

```
String value = req.getParameter("parameterName");
```

**Exa:** String email = req.getParameter("email");

```
String password = req.getParameter("password");
```

## Annotations

- it use to do auto configuration which start from @ and follow pascal case.
- To replace XML-based configuration and reduce line of code, we can use the @WebServlet annotation. This annotation should be placed above the servlet class declaration.

## @WebServlet Annotation

The @WebServlet annotation is used to specify the URL patterns that a servlet should handle. It takes a URL pattern as an argument.

## Syntax:

```
@WebServlet("/url-pattern")
```

## Handling Requests:

To handle HTTP GET, PUT and POST requests, we override the doGet(), doPut() and doPost() methods of the HttpServlet class. These methods receive HttpServletRequest and HttpServletResponse objects as parameters.

## Syntax:

```
protected void doGet(HttpServletRequest req,  
HttpServletResponse resp) throws ServletException,  
IOException {
```

```
    // Code to handle GET requests
```

```
}
```

```
protected void doPost(HttpServletRequest req,  
HttpServletResponse resp) throws ServletException,  
IOException {
```

```
    // Code to handle POST requests
```

```
}
```

- Servlet chaining is a technique in Java web applications where one servlet can forward a request to another servlet to handle processing.

## There are two main ways to achieve servlet chaining:

### 1. Using RequestDispatcher:

- The RequestDispatcher interface provides methods to forward requests to another resource, such as a servlet.

- To obtain a RequestDispatcher object, you can use the getRequestDispatcher() method of the ServletRequest object.

## 1. Using sendRedirect():

- The sendRedirect() method redirects the client to a different URL. This is useful for redirecting to external resources or to a different servlet.

### RequestDispatcher Interface:

The RequestDispatcher interface is used to forward requests to another resource, such as a servlet or a JSP. It's located in the javax.servlet package.

### Obtaining a RequestDispatcher Object:

To get a RequestDispatcher object, we use the getRequestDispatcher() method of the HttpServletRequest interface.

### Syntax:

```
RequestDispatcher rd =  
req.getRequestDispatcher("/path/to/resource");
```

The getRequestDispatcher() method takes a URL path as an argument and returns a RequestDispatcher object.

### Servlet Chaining:

To combine the functionality of multiple servlets, we can use servlet chaining. This involves sending the request and response objects from one servlet to another.

### Methods for Servlet Chaining:

Two common methods for servlet chaining are:

#### 1. forward():

- This method forwards the request to another resource, such as a servlet or a JSP.
- The original request and response objects are passed to the target resource.
- The client's browser URL remains unchanged.

#### 2. include():

- This method includes the content of another resource into the current response.
- The original request and response objects are also passed to the included resource.
- The client's browser URL remains unchanged.

## **sendRedirect() Method**

The `sendRedirect()` method is used to redirect the client's browser to a different URL. This is useful for:

- Redirecting to a different page or servlet.
- Redirecting after a form submission.
- Redirecting after a successful operation.

### **Syntax:**

```
response.sendRedirect("url");
```

- The `url` parameter can be a relative or absolute URL.
- It redirects the client's browser to the specified URL, and the original request and response objects are discarded.

### **Sending Additional Data with `sendRedirect()`:**

The `sendRedirect()` method itself doesn't directly allow you to send additional data to the target URL. However, you can use alternative methods to achieve this:

#### **1. Hidden Input Fields:**

- Create hidden input fields in your HTML form with the desired data.

- When the form is submitted, the values of these hidden fields will be included in the request parameters.
- In the target servlet, you can access these values using the `getParameter()` method.

### **2. URL Rewriting:**

- Append the additional data as parameters to the URL in the `sendRedirect()` method.
- The target servlet can then extract the data from the request URL.

### **Example:**

```
<form action="targetServlet" method="post">
<input type="hidden" name="hiddenData" value="secretValue">
<input type="submit" value="Submit">
</form>// In the target servletString hiddenValue =
request.getParameter("hiddenData");
```

## Sending Data from Front-end to Back-end:

To send data from the front-end to the back-end using URL rewriting, you need to include the data as parameters in the URL. Use the & symbol to separate multiple parameters.

### Example:

http://example.com/myServlet?name=Alice&age=30

In the back-end servlet, you can retrieve these values using the getParameter() method:

```
String name = request.getParameter("name"); int age =  
Integer.parseInt(request.getParameter("age"));
```

## Sending Data Between Servlets:

To send data from one servlet to another, you can use:

- **Session Objects:**

- Create a session object using HttpSession session = request.getSession().
- Store data in the session using session.setAttribute("key", value).
- Retrieve data in the target servlet using session.getAttribute("key").

- **Cookie Objects:**

- Create a cookie object using Cookie cookie = new Cookie("name", "value").
- Set the cookie's attributes (e.g., expiration time, path, domain).
- Add the cookie to the response using response.addCookie(cookie).
- Retrieve the cookie in the target servlet using request.getCookies().

## HttpSession Interface:

- **Definition:** HttpSession is an interface in the jakarta.servlet.http package. It represents a session between a client and a server.
- **Creating a Session:** To get a session object, we use the getSession() method of the HttpServletRequest interface. This method creates a new session if one doesn't exist or returns an existing session.

```
HttpSession session = request.getSession();
```

- **Storing and Accessing Data:** We can store and retrieve data from the session using the setAttribute() and getAttribute() methods.

```
// Store data
session.setAttribute("key", value); // Retrieve
dataObject value = session.getAttribute("key");
```

## HttpSession Interface:

- The HttpSession interface represents a session between a client and a server. It's used to store session-specific data, such as user preferences, shopping cart items, or authentication information.

- The HttpSession interface is used to manage user sessions in web applications. It allows you to store and retrieve data associated with a specific user session.

## Key Methods:

- **setAttribute(String key, Object value):**
  - Stores an object under the specified key in the session.
  - The value can be any object type.
- **getAttribute(String key):**
  - Retrieves the object associated with the specified key from the session.
  - The returned object needs to be cast to the appropriate type.

## setAttribute() Method:

The setAttribute() method is used to store an object in the session. It takes two arguments:

1. **Key:** A string that identifies the object.
2. **Value:** The object to be stored.

### Exa :

```
HttpSession session = request.getSession();
session.setAttribute("id", 101);
session.setAttribute("name", "Hiuga");
// Retrieve values
int id = (Integer) session.getAttribute("id");
String name = (String) session.getAttribute("name");
```

### Setting Session Timeout

To set the maximum inactive interval for a session, we use the `setMaxInactiveInterval()` method. This method is available in the `HttpSession` interface.

### Syntax:

```
session.setMaxInactiveInterval(seconds);
```

# Course : Java

## Part : JSP (Java Server Pages)

Code With Ved Prakash

## JSP (JavaServer Pages)

- JSP stands for JavaServer Pages.
- It allows you to combine HTML code with Java code within a single file.
- This approach helps to overcome the limitations of using `PrintWriter` to generate dynamic HTML content.
- When a programmer needs to print only static HTML, they can use `PrintWriter`.
- However, when dynamic content and server-side logic are required, JSPs are a more convenient and efficient solution.
- The JSP engine automatically generates a servlet class for each JSP file at runtime.

## JSP Tags

JSP (JavaServer Pages) files allow you to combine HTML and Java code within a single file. To structure and separate the different types of code, JSP provides three main types of tags:

### 1. Declaration Tag:

- Used to declare variables and methods that can be accessed throughout the JSP page.
- Syntax: `<%! Java code %>`

### 2. Scriptlet Tag:

- Used to execute Java code within the JSP page.
- Syntax: `<% Java code %>`

### 3. Expression Tag:

- Used to print the value of an expression directly to the output.
- Syntax: `<%= expression %>`

## Syntax:

```
<%= expression %>
```

```
<%!
```

```
    int principle = 1000000;
```

```
    double interestRate = 10.5;
```

```
    int periodInYears = 10;
```

```
%>
```

```
<%
```

```
    double rateOfInterest = interestRate / 12 / 100;
```

```
    int periodInMonths = periodInYears * 12;
```

```
    double emi = principle * rateOfInterest * Math.pow(1 +
rateOfInterest, periodInMonths) / (Math.pow(1 +
rateOfInterest, periodInMonths) - 1);
```

```
%>
```

EMI: <%= emi %>



**Course : Java**  
**Part : JDBC Architecture:**

Code With Ved Prakash

## **JDBC Architecture:**

To understand database connectivity using JDBC, it's important to understand the key components of JDBC architecture:

### **1. Java Application:**

- It is used to collect user information.
- Java application cannot store user information permanently so we make use of database application to store data permanently.

### **2. JDBC Driver:**

- A software component that enables communication between a Java application and a specific database.
- It translates JDBC calls into database-specific calls.

### **3. JDBC API:**

- A set of interfaces and classes that define the standard way to interact with databases.
- It provides methods for connecting to databases, executing SQL queries, and processing results.

- It provide connection b/w database and java application.

### **2. Database Driver:**

- A software component provided by the database vendor that implements the JDBC API and interacts with the database.

## Collecting User Information in Java Applications

We can collect user information in Java applications through two primary methods:

### 1. Basic Level Data Collection:

- This involves using the Scanner class to read input from the console.
- It's suitable for simple applications where user input is limited to text-based commands or data.

### 2. High Level Data Collection:

- This involves using web technologies like HTML forms to collect user input.
- The user interacts with the web page, fills out forms, and submits the data to the server.

### Key Points:

- **Java Language:** Java applications can only understand Java code.
- **Database Language:** Databases understand SQL queries.

➤ **JDBC Driver's Role:** The JDBC driver converts Java-specific data and commands into SQL queries that the database can execute.

### Reasons for using MySQL:

- **Cost-Effective:** MySQL is an open-source database, making it free to use.
- **Ease of Use:** It's relatively easy to learn and set up.
- **Popularity:** MySQL is a widely used database, making it a popular choice for developers.

### Creating a Database in MySQL Workbench:

1. Open MySQL Workbench.
2. Connect to your MySQL server.
3. Right-click on the server connection and select "Create Schema."
4. Specify the desired name for your new database.
5. Click "Apply" and then "Finish."

## **CRUD operation query :**

SELECT \* FROM table\_name; -- Selects all columns and rows

SELECT column1, column2 FROM table\_name; -- Selects specific columns

INSERT INTO table\_name (column1, column2) VALUES (value1, value2);

UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;

DELETE FROM table\_name WHERE condition;

## **JDBC Connection steps**

- ○ For making connection b/w java and database we are using Connection interface
- For Getting connection object we are using getConnection() method which is present in DriverManager class.
- For taking input we have 2 interface PreparedStatement and statement.
- If we want to take run time input then we are using PreparedStatement otherwise Statement interface, for getting this object we need to call PreparedStatement() or statement() which is present in side connection interface and it is non-static method so call by orv name.
- Now for executing query we have 2 method executeQuery() and executeUpdate().
- Which is present in side PreparedStatement or Statement interface and non-static method.
- For executeUpdate() return type is intiger type and for executeQuery() return type is ResultSet.

## Establishing a Database Connection in Java

To establish a connection between a Java application and a database, we use the `Connection` interface from the `java.sql` package. Here's a breakdown of the steps involved:

### 1. Load the JDBC Driver:

- Load the appropriate JDBC driver for your database (e.g., MySQL, Oracle, PostgreSQL). This is typically done using the `Class.forName()` method.

### 2. Establish a Connection:

- Use the `DriverManager.getConnection()` method to create a connection to the database.
- Provide the database URL, username, and password as arguments.

Exa :

```
try {  
    // Replace with your JDBC driver class name  
    Class.forName("com.mysql.cj.jdbc.Driver");  
  
    Connection connection = DriverManager.getConnection(  
        "jdbc:mysql://localhost:3306/mydatabase", "username",  
        "password");  
  
    // Use the connection to execute SQL queries  
}  
catch (ClassNotFoundException | SQLException e) {  
    e.printStackTrace();  
}
```

## getConnection() Method:

- The getConnection() method is a static method within the DriverManager class.
- It takes a URL as an argument, which specifies the database connection details (e.g., database URL, username, password).
- It returns a Connection object, representing the established database connection.

```
Connection connection = DriverManager.getConnection(url);
```

Example:

```
try {  
    Connection connection =  
        DriverManager.getConnection("jdbc:mysql://localhost:3306/  
        mydatabase", "username", "password");  
  
    // Use the connection to execute SQL queries  
}  
catch (SQLException e) {  
    e.printStackTrace();  
}
```

## URL (Uniform Resource Locator)

**Example:** <https://www.google.com/search>

### Key Components of a URL:

- **Protocol:** The communication protocol used to access the resource (e.g., HTTP, HTTPS, FTP).
- **Domain Name:** The address of the server hosting the resource.
- **Path:** The specific location of the resource within the server.

### User Information:

- Usernames and passwords are essential for security when accessing databases.
- They help protect sensitive data from unauthorized access.

### Database Connection Strings:

#### MySQL:

`jdbc:mysql://localhost:3306?user=root&password=12345`

#### Oracle:

`jdbc:oracle:thin:@localhost:1521:XE?user=scott&password = tiger`

## **Java Applications:**

Java applications can combine both Java code (for business logic) and HTML code (for user interface) to create dynamic web applications.

## **Creating a Platform for Query Execution**

A platform, such as a Java application, serves as a bridge between the Java code and the database. It allows you to write Java code to execute SQL queries and process the results.

### **Key Points:**

- **Java Code vs. SQL Code:**
  - Java code is understood by the Java compiler.
  - SQL code is interpreted by the database engine.
- **Platform's Role:**
  - The platform is responsible for:
    - Converting Java code into SQL queries.
    - Sending the SQL queries to the database.
    - Processing the results returned by the database.
    - Presenting the results to the user.

## **Registering a JDBC Driver:**

To establish a database connection in Java, you need to register the appropriate JDBC driver. This is typically done using the Class.forName() method.

### **Steps:**

#### **Load the Driver Class:**

- Use Class.forName() to load the driver class.
- Provide the fully qualified class name of the JDBC driver as an argument.

```
Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL
```

#### **Establish the Connection:**

- Use DriverManager.getConnection() to create a connection to the database.
- Provide the database URL, username, and password as arguments.

```
Connection connection = DriverManager.getConnection(  
    "jdbc:mysql://localhost:3306/mydatabase", "username",  
    "password");
```

## ClassNotFoundException:

- The Class.forName() method can throw a ClassNotFoundException if the specified driver class is not found in the classpath.
- This is a checked exception, meaning you must handle it using a try-catch block.

Exa :

```
try {  
    Class.forName("com.mysql.cj.jdbc.Driver");  
    // code  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

## Types of Platforms:

There are two types of platforms for executing SQL queries.

### 1. Statement Platform:

- Uses Statement interface to execute static SQL queries.
- Less efficient for repeated queries with different parameters.

### 2. PreparedStatement Platform:

- Uses PreparedStatement interface to execute parameterized SQL queries.
- More efficient and specially for dynamic queries.

## Statement Interface:

The Statement interface provides methods for executing SQL statements. Here are two common methods:

### 1. executeUpdate(String query):

- Used to execute SQL statements that modify data, such as INSERT, UPDATE, and DELETE statements.
- Returns the number of rows affected by the statement.

## 2. `executeQuery(String query):`

- Used to execute SQL statements that retrieve data, such as SELECT statements.
- Returns a ResultSet object containing the result set.

### Processing Resultant Data

In a database system, data can exist in two primary states:

#### 1. Actual Data:

- This is the data that is permanently stored in the database.
- It's accessed and modified through SQL queries.

#### 2. Resultant Data:

- This is temporary data that is generated as a result of executing a SQL query.
- It's typically stored in memory and can be accessed and processed by the application.

### Database Query Execution:

When a SQL query is sent to the database, it undergoes a three-step process:

1. **Compilation:** The database compiler checks the syntax and semantics of the query.

2. **Execution:** The database engine executes the query, accessing and manipulating the actual data.

3. **Output:** The results of the query are returned to the application in the form of a result set.

### Resultant Data :

- Data that is temporarily stored in memory after being retrieved from the database.
- It's not permanently stored in the database.

`select * from employees;`

### Steps :

- The query is sent to the database.
- db compiler checks the syntax and semantics of the query.
- The database engine executes the query, retrieves the data from the employees table.
- The retrieved data is stored in buffer memory as a result set.
- The result set is sent back to the application.

## Result Set and Buffer Memory

When you execute a SELECT query using a Statement object, the result is stored in a ResultSet object. This ResultSet object acts as a cursor over the result set, allowing you to iterate through the rows of data.

### Buffer Memory:

- The ResultSet object maintains an internal buffer to store the data fetched from the database.
- This buffer memory typically holds a subset of the entire result set to optimize performance.
- The buffer memory is associated with two pointers:
  - **BFR (Before First Record):** Points to the position before the first row.
  - **ALR (After Last Record):** Points to the position after the last row.

### Navigating the Result Set:

To move the cursor within the result set, you can use various methods provided by the ResultSet interface:

- **first():** Moves the cursor to the first row.

- **last():** Moves the cursor to the last row.
- **relative(int rows):** Moves the cursor relative to the current position.
- **next():** Moves the cursor to the next row.
- **previous():** Moves the cursor to the previous row.
- **absolute(int row):** Moves the cursor to the specified row number.

### Getter Methods in ResultSet:

Getter methods in the ResultSet interface are used to retrieve data from a specific column in the result set. They accept either the column name or the column index as an argument.

### Common Getter Methods:

- **getInt(columnLabel):** Retrieves an integer value from the specified column.
- **getString(columnLabel):** Retrieves a string value from the specified column.
- **getDouble(columnLabel):** Retrieves a double value from the specified column

Exa :

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

```
while (rs.next()) {  
    int id = rs.getInt("empid");  
    String name = rs.getString("ename");  
    double salary = rs.getDouble("salary");  
    // ...  
}
```

## PreparedStatement

A PreparedStatement is an interface in the java.sql package that represents a precompiled SQL statement. It's used to execute SQL statements with parameters, providing several advantages:

- **Performance:** Prepared statements are precompiled, which can significantly improve performance, especially for frequently executed queries.
- **Security:** Using prepared statements helps prevent SQL injection attacks by separating SQL statements from user-provided data.
- **Code Readability:** Prepared statements make your code more readable and maintainable.

## Using PreparedStatement:

### 1. Create a PreparedStatement:

```
PreparedStatement preparedStatement =  
connection.prepareStatement("INSERT INTO employee  
VALUES (?, ?, ?);");
```

## Set Parameters:

```
preparedStatement.setInt(1, 101);  
preparedStatement.setString(2, "Dinga");  
preparedStatement.setInt(3, 1000);
```

## Using a PreparedStatement:

- **Setting Parameters:**
  - Use setInt, setString, setDouble, etc., to set values for the placeholders in the SQL statement.
- **Executing the Statement:**
  - Use executeUpdate() for INSERT, UPDATE, and DELETE statements.
  - Use executeQuery() for SELECT statements.

## Setter Methods in PreparedStatement

Setter methods in the PreparedStatement interface are used to set values for the placeholders in a prepared statement. These methods accept two arguments:

1. **Placeholder Position:** An integer indicating the position of the placeholder in the SQL statement, starting from 1.
2. **Value:** The value to be assigned to the placeholder.

## Common Setter Methods:

- setInt(int placeHolder, int value): Sets an integer value.
- setString(int placeHolder, String value): Sets a string value.
- setDouble(int placeHolder, double value): Sets a double value

## Exa :

```
PreparedStatement pstmt =  
connection.prepareStatement("INSERT INTO employees (id,  
name, salary) VALUES (?, ?, ?)");  
  
pstmt.setInt(1, 101);  
pstmt.setString(2, "Alice");  
pstmt.setDouble(3, 50000);  
  
pstmt.executeUpdate();
```

**EXA:**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class EmployeeDetail {
    public static void main(String[] args) {
        String url =
"jdbc:mysql://localhost:3306?user=root&password=12345";
        String select = "select * from training_employee";
        try (Connection connection =
DriverManager.getConnection(url);
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(select))
        while (resultSet.next()) {
            int id = resultSet.getInt("emp_id");
```

```
ResultSet resultSet = statement.executeQuery(select)) {
    while (resultSet.next()) {
        int id = resultSet.getInt("emp_id");
        String name = resultSet.getString("emp_name");
        double salary = resultSet.getDouble("emp_salary");
        int deptno = resultSet.getInt("emp_deptno");
        System.out.println("Emp id: " + id);
        System.out.println("Emp name: " + name);
        System.out.println("Emp salary: " + salary);
        System.out.println("Emp deptno: " + deptno);
    }
} catch (SQLException e) {
    System.err.println("Error fetching employee details: " +
e.getMessage());
    e.printStackTrace(); // For debugging
}}
```

Exa :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;

public class Registration {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first name: ");
        String firstName = scanner.nextLine();
        String insert = "insert into userinfo (firstname, lastname, email, password) values (?, ?, ?, ?)";
        String url =
"jdbc:mysql://localhost:3306/test?user=root&password=12345";
```

```
try (Connection connection = DriverManager.getConnection(url);
      statement.setString(1, firstName);
      int rowsInserted = statement.executeUpdate();
      if (rowsInserted > 0) {
          System.out.println("Registration successful.");
      } else {
          System.out.println("Registration failed.");
      }
  } catch (SQLException e) {
      System.err.println("Error during registration: " +
e.getMessage());
      e.printStackTrace(); // For debugging
  }
}
```

## Exa : 8 Employee detail

```
<!DOCTYPE html>
<html>
<head>
    <title>Employee Detail</title>
</head>
<body><form action="empAction">
    <input type="text" placeholder="Enter Id" name="id">
    <input type="text" placeholder="Enter Name"
name="name">
    <input type="number" placeholder="Enter Salary"
name="salary">
    <input type="text" placeholder="Enter Department No"
name="deptno">
    <input type="submit" value="Submit">
</form>
</body></html>
```

```
public class EmpAction extends HttpServlet {
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        String id = request.getParameter("id");
        String name = request.getParameter("name");
        String salary = request.getParameter("salary");
        String deptno = request.getParameter("deptno");
        String insert = "insert into employee (emp_id, emp_name,
emp_salary, emp_deptno) values (?, ?, ?, ?)";
        String url =
"jdbc:mysql://localhost:3306/test?user=root&password=12345";
        try (Connection connection =
DriverManager.getConnection(url);
        PreparedStatement statement =
connection.prepareStatement(insert)) {
            statement.setString(1, id);
            statement.setString(2, name);
```

```
statement.setString(3, salary);

        statement.setString(4, deptno);

        int rowsInserted = statement.executeUpdate();
        if (rowsInserted > 0) {

            response.getWriter().println("Employee details
inserted successfully.");
        } else {

            response.getWriter().println("Error inserting
employee details.");
        }

    } catch (SQLException e) {

        response.getWriter().println("Error: " +
e.getMessage());

        e.printStackTrace(); // For debugging
    }
}
```

### xml file (.web.xml)

```
<web-app>

    <servlet>

        <servlet-name>employeeDetail</servlet-name>
        <servlet-class>com.EmployeeDetail</servlet-class>
    </servlet>

    <servlet-mapping>

        <servlet-name>employeeDetail</servlet-name>
        <url-pattern>/employeeDetail</url-pattern>
    </servlet-mapping>
</web-app>
```

**Exa : reducing xml config with**

**@WebServlet("/StudentLogin") annotation**

```
package com.jsp.student.servlet;  
  
import javax.servlet.http.HttpServlet;  
  
import javax.servlet.http.HttpServletRequest;  
  
import javax.servlet.http.HttpServletResponse;  
  
import java.io.IOException;
```

```
@WebServlet("/StudentLogin")  
  
public class StudentLogin extends HttpServlet {  
  
    @Override  
  
    protected void doPost(HttpServletRequest req,  
    HttpServletResponse resp) throws ServletException,  
    IOException {  
  
        String email = req.getParameter("email");  
  
        String pass = req.getParameter("pass");
```

```
StudentDAO dao = new StudentDAOImplementation();
```

```
StudentLogin studentLogin = dao.studentLogin(email, pass);
```

```
    // Assign the value from backend to object
```

```
}
```

```
}
```

# Course : Java

## Part : Spring Framework



## Spring Framework

- Open-source, loosely coupled, lightweight Java framework.
- Reduces complexity of developing enterprise applications.
- Supports various other frameworks like Hibernate.

## Modules

- Core Container: Manages objects (beans) in an application.
- Data Access/Integration: Simplifies database interactions.
- Web: Simplifies web application development (MVC, security).
- AOP (Aspect-Oriented Programming): Implements cross-cutting concerns.
- Messaging: Supports asynchronous communication.
- Test: Offers tools for unit and integration testing.

## Benefits

- Reduced complexity
- Loose coupling (components independent of each other)
- Improved maintainability
- Increased productivity

## Configuration

- XML-based configuration: Defines beans and dependencies in XML files.
- Java-based configuration: Uses annotations for configuration within Java classes.

## Core Container

- Manages object (bean) creation, configuration, and lifecycle.

## Bean Scopes

- **Singleton:** Single instance for entire application lifecycle (default).
- **Prototype:** New instance for every bean request.

## Dependency Injection

- Spring injects dependencies into beans:
  - **Constructor Injection:** Dependencies passed through the constructor.
  - **Setter Injection:** Dependencies injected using setter methods.

## Annotations

- Common annotations for configuration and dependency injection:
  - `@Component`: Marks a class as a Spring bean.
  - `@Autowired`: Marks a field or method for dependency injection.
  - `@Qualifier`: Used with `@Autowired` to specify a particular bean.
  - `@Bean`: Used in Java-based configuration to define a bean method.

## Spring JDBC

- Offers an abstraction layer for db Simplifies JDBC operation.

## Principles

- **Inversion of Control (IoC):** Spring manages object creation and lifecycle.
- **Dependency Injection (DI):** Objects rely on Spring for dependencies.

## IoC Container

- Manages object creation (beans).
- Focuses on business logic, not object creation.
- Uses ApplicationContext or BeanFactory for creation.
  - ApplicationContext supports XML and annotations.
  - BeanFactory supports only XML.

## Bean

- Special object created by the IoC container.

## Spring Configuration XML File

- Used for Java configurations.
- Uses tags to define beans.
- bean tag with id and class attributes defines a bean.

## ApplicationContext

- Interface in org.springframework.context package.
- Supports both XML and annotation configuration.
- Implementation class: ClassPathXmlApplicationContext (in org.springframework.context.support).

## Accessing Objects from IoC Container

- Use getBean( ) method in ApplicationContext.
- Provide bean id as an argument.
- Downcast the returned object to the required class.

## Creating a Spring Core Java Project

1. Create a Java project.
2. Access Spring Core jar files.
3. Configure Java build path.
4. Create an XML file or import an existing one.
5. Create a class with the main method.
6. Access the IoC container and objects in the main method.

## getBean( ) Method Overloads

1. Provide a class reference variable (downcast required).
2. Provide a reference variable and class name.
3. Provide only class name (throws exception if multiple objects exist).

## Bean Scope

- Controls bean behavior using the scope attribute in the bean tag.
- Values:
  - singleton: Single instance.
  - prototype: New instance for each request.

## Example:

```
<bean id="ref" class="fully.qualified.class.name"></bean>Demo  
demo = (Demo) context.getBean("demo");demo.sample();
```

## Syntax:

- 1.Demo demo = (Demo) context.getBean("demo");
- 2.Demo demo = context.getBean("demo", Demo.class);
- 3.Demo demo = context.getBean(Demo.class); (throws exception if multiple)

## Spring Configuration XML File

- Used for Java configurations.
- Uses tags to define beans.
- bean tag with id and class attributes defines a bean.

## ApplicationContext

- Interface in org.springframework.context package.
- Supports both XML and annotation configuration.
- Implementation class: ClassPathXmlApplicationContext (in org.springframework.context.support).

## Accessing Objects from IoC Container

- Use getBean( ) method in ApplicationContext.
- Provide bean id as an argument.

Downcast the returned object to the required class

## getBean( ) Method Overloads

1. Provide a class reference variable (downcast required).
2. Provide a reference variable and class name.
3. Provide only class name.

## Creating a Spring Core Java Project

1. Create a Java project.
2. Access Spring Core jar files.
3. Configure Java build path.
4. Create an XML file or import an existing one.
5. Create a class with the main method.
6. Access the IoC container and objects in the main method.

## Accessing Objects from IoC Container

- Use getBean( ) method in ApplicationContext.
- Provide bean id as an argument.
- Downcast the returned object to the required class.

## Constructor Injection (continued):

- **index attribute:** Used to specify the index position of a variable in the constructor arguments.
- **Index always starts with 0.**

## c Namespace:

- XML property to avoid <constructor-arg> tag.
- Import URL:  
    xmlns:c="http://www.springframework.org/schema/c"
- With c namespace, no need for name, index, type, value, or <value> tag.
- Mention variable names directly within the bean tag.

## Example:

```
<bean id="emp4" class="org.jsp.employee.Employee"  
c:name="King"    c:deptno="30"    c:salary="50000"  
c:id="105"></bean>
```

## Constructor Injection for Objects:

- Use ref attribute within <constructor-arg> tag.

## Syntax:

```
<constructor-arg ref="referenceVariableName"></constructor-arg>
```

## Passing Reference Variable:

- Use <ref> tag with bean attribute to pass the reference variable of the dependent.

## Syntax:

```
<constructor-arg> <ref bean= "referenceVariableName"> </ref>  
</constructor-arg>
```

## Passing List of Objects:

- Use <list> tag within <constructor-arg> tag.
- Inside <list>, use <ref> tag to pass object reference variables.

## Syntax:

```
<constructor-arg>  
<list>  <ref>refVariable1</ref>  
<ref>refVariable2</ref>  
<ref>refVariable3</ref></list>  
</constructor-arg> a
```

## Setter Injection

- Process of injecting data (dependencies) using setter methods.
- Involves:
  - 1.Default constructor to create an object.
  - 2.Setter methods to initialize non-static properties.
  - 3.IoC container uses setter methods after object creation.

### Syntax:

- 1.Create setter methods for non-static properties.
- 2.Use <property> tag in the bean definition:
  - a. name attribute: specifies the property name.
  - b. value attribute: specifies the value to inject.

### XML

```
<bean id="student3" class="com.jsp.setters.Student">
<property name="id" value="101"></property></bean>
```

## p Namespace:

- XML property to avoid <property> tag.
- Import URL:  
[xmlns:p=http://www.springframework.org/schema/p](http://www.springframework.org/schema/p)

- With p namespace, directly define properties within the bean tag.

## Example:

```
<bean id="student3" class="com.jsp.setters.Student"
p:id="104" p:studentName="Raja"
p:studentEmailId="raja@gmail.com"
p:studentPercentage="85.5"></bean>
```

## Setter Injection for Objects:

- Use ref attribute within <property> tag to inject a reference to another bean.

### Syntax:

```
<property name="address" ref="add"></property>
```

## Passing Reference Variable:

- Use <ref> tag with bean attribute to pass the reference variable of the dependent.

### Syntax:

```
<property>
<ref bean="referenceVariable"></ref>
</property>
```

## # Setter Injection vs. Constructor Injection?

Feature	Setter Injection	Constructor Injection
<b>Method</b>	Uses setter methods to inject	Uses constructor arguments to inject
<b>XML Tag</b>	<property>	<constructor-arg>
<b>Identification</b>	Setter method name (name attribute)	Constructor argument name/type/index
<b>Speed</b>	Slower (injection after creation)	Faster (injection during creation)
<b>IoC Container Creation</b>	Creates target bean first	Creates target bean first
<b>Default Constructor</b>	Uses 0-parameter constructor	Uses parameter constructors
<b>XML Namespace</b>	p namespace (optional)	c namespace (optional)

## Autowire

- This annotation is used to inject automatic dependency injection.
- Replaces manual ref attribute usage.

## Values:

- byName: Injects by property name (id must match variable name).
- byType: Injects by class name (setter injection).
- constructor: Injects through constructor arguments.

## Note:

- @Autowired annotation is for non-primitive data types.
- Autowire values use camel case (e.g., byName).

## Annotations

- Placed above class, method, or variable names.
- Executed at compile time or runtime.
- Optional in Java, but mandatory for Spring configuration.
- Reduce XML configuration and code size.

## Important Spring Core Annotations:

- @Component: Creates a bean (object).
- @Autowired: Performs dependency injection (automatically).
- @Qualifier: Qualifies a bean for injection (if multiple candidates exist).
- @Bean: Defines a bean method in Java configuration.
- @ComponentScan: Scans for @Component annotations in packages.
- @Configuration: Marks a class as a Spring configuration class.
- @Value: Injects values from properties files or environment variables.

## @Component:

- Replaces the <bean> tag for creating beans.
- Placed above the class definition.
- Requires <context:component-scan base-package="com.jsp.annotation.AutoWiredAnnotation"> in XML configuration to scan for @Component annotations.

### **Autowired:**

- Performs automatic dependency injection for non-primitive data types.
- Used in three ways:
  - Above the non-primitive data type field.
  - Above the setter method.
  - Above a constructor with arguments.

### **@Qualifier:**

- Used to specify which bean to inject when multiple beans of the same type are available.
- Avoids ambiguity in autowiring.

### **@Value:**

- Injects values into fields or methods.
- Used for primitive data types.
- Can be used on fields, setter methods, or constructor arguments.

### **Java-Based Configuration**

- Reduces XML configuration.
- @Configuration annotation on configuration classes.

- Defines beans using @Bean annotation on methods.
- Uses @ComponentScan to scan for @Component annotated classes.

### **Example:**

```
@Configuration  
@ComponentScan("com.example")  
public class AppConfig {
```

```
    @Bean  
    public MyService myService() {  
        MyService service = new MyService();  
        service.setRepository(myRepository());  
        return service;  
    }
```

```
    @Bean  
    public MyRepository myRepository() {  
        return new MyRepositoryImpl();  
    }
```

## Spring JDBC

- Simplifies database operations.
- Provides JdbcTemplate class for common database operations.
- Handles connection management and exception handling.

## JdbcTemplate

- Pre-built methods for:
  - Insert, update, delete operations (update())
  - Select queries (queryForObject(), query())
- Requires a DataSource object to establish database connections.
- Uses RowMapper interface to map database results to Java objects.

## Configuration

- XML-based or Java-based configuration.
- Define DataSource and JdbcTemplate beans.

## Example (XML-based):

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
    value="com.mysql.cj.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
  <property name="username" value="root" /> <bean>
```

```
<bean id="jdbcTemplate"
  class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource" />
</bean>
```

## Example (Java-based):

```
@Configuration
public class AppConfig {
  @Bean
  public DataSource dataSource() {
    DriverManagerDataSource dataSource = new
    DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
    dataSource.setUsername("root");
    dataSource.setPassword("password");
    return dataSource;
  }
  @Bean
  public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
  }
}
```

## RowMapper

- Interface used to map database results to Java objects.
- `mapRow()` method is implemented to map each row.
- `BeanPropertyRowMapper` can be used for simplify mapping.

## Database Operations

- **Insert, Update, Delete:**

```
jdbcTemplate.update(sql, new  
PreparedStatementSetter() {  
    // Set parameters here  
});
```

- **Select (single row):**

```
Object result = jdbcTemplate.queryForObject(sql, new  
RowMapper<Object>() {  
    // Map result to an object  
});
```

- **Select (multiple rows):**

```
List<Object> results = jdbcTemplate.query(sql, new  
RowMapper<Object>() {  
    // Map each row to an object });
```

# Course : Java

# Part : Spring data



## Spring Data JPA (Java persistence API)

- Simplifies data access with JPA.
- Provides JpaRepository interface for common CRUD operations.
- Uses annotations for mapping entities to database tables.

### Additional Topics:

- Spring MVC (Model-View-Controller)
- Spring Security
- Spring Boot
- Spring AOP
- Spring Batch

### Spring Integration

**Note:** This is a brief overview. For more in-depth information, refer to official Spring documentation and tutorials.

Sources and related content

### XML-Based Configuration:

```
<bean id="temp"
  class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="ds"></property></bean>
```

```
<bean id="ds"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="url"
    value="jdbc:mysql://localhost:3306/demo"></property>
  <property name="username" value="root"></property>
  <property name="password" value="12345"></property>
  <property name="driverClassName"
    value="com.mysql.jdbc.Driver"></property>
</bean>

<bean id="empdao"
  class="com.jsp.jdbc.SpringJdbcByUsingXmlFile.Dao.Employee
  DaoImp">
  <property name="jdbcTemplate" ref="temp"></property>
</bean>

<bean id="emp"
  class="com.jsp.jdbc.SpringJdbcByUsingXmlFile.model.Employee"></bean>
```

## Java-Based Configuration:

```
@Configuration  
 @ComponentScan(basePackages = "org.jsp.jdbc")  
 public class EmployeeConfig {  
     @Bean  
     public JdbcTemplate getJdbcTemplate() {  
         return new JdbcTemplate(getDataSource()); }  
     @Bean  
     public DataSource getDataSource() {  
         DriverManagerDataSource dataSource = new  
             DriverManagerDataSource();  
         dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");  
         dataSource.setUrl("jdbc:mysql://localhost:3306/demo");  
         dataSource.setUsername("root");  
         dataSource.setPassword("12345");  
         return dataSource; } }
```

## Database Operations:

### Write Operations (update())

- ❑ Use update() method from JdbcTemplate.
- ❑ Arguments:
  - String sql: Write query (INSERT, UPDATE, DELETE).
  - PreparedStatementSetter pss: Provides runtime values for placeholders.

### Syntax:

```
jdbcTemplate.update(String sql, PreparedStatementSetter pss);
```

### Example:

```
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";  
jdbcTemplate.update(sql, (ps) -> { ps.setString(1, "John  
Doe"); ps.setString(2, "john.doe@example.com");});
```

### Read Operations

- Use queryForObject() for single data retrieval.
- Use query() for multiple data retrieval.
- Both methods are argument methods.
- Arguments:
  - String sql: Read query (SELECT).
  - RowMapper<T> rowMapper: Maps database records to Java objects.

## Syntax:

```
// Single data  
Object result = jdbcTemplate.queryForObject(String sql,  
RowMapper<T> rowMapper);  
  
// Multiple  
dataList<T> results = jdbcTemplate.query(String sql,  
RowMapper<T> rowMapper);
```

## RowMapper Interface

- Used to map database data to Java objects.
- Contains mapRow() method (incomplete) for mapping each row.
- Arguments:
  - ResultSet: Retrieves data from the database.
  - int rowNum: Current row number.

## Example (Simple Mapping):

```
class User {  
    private String name;  
    private String email;  
    // Getters and setters  
}
```

```
public class UserMapper implements RowMapper<User> {  
    @Override  
    public User mapRow(ResultSet rs, int rowNum) throws  
        SQLException  
    {  
        User user = new User();  
        user.setName(rs.getString("name"));  
        user.setEmail(rs.getString("email"));  
        return user;  
    }  
}
```

## RowMapper Implementation:

- Create a class implementing RowMapper<T>.
- Specify the class type for the mapped object.
- Override mapRow() method.
- Use ResultSet getters to retrieve data for each row.
- Set retrieved data to object properties.

### Example:

```
class User {  
    private String name;  
    private String email;  
    // Getters and setters  
}  
  
public class UserMapper implements RowMapper<User> {  
    @Override  
    public User mapRow(ResultSet rs, int rowNum) throws  
        SQLException  
    {  
        User user = new User();  
        user.setName(rs.getString("name"));  
        user.setEmail(rs.getString("email"));  
        return user;  
    }  
}
```

### BeanPropertyRowMapper:

- Simplifies object mapping for classes with matching property names.
- Use BeanPropertyRowMapper<T>(EntityClass.class) constructor.

### Syntax:

```
BeanPropertyRowMapper<User> mapper = new  
    BeanPropertyRowMapper<>(User.class);
```

### JPA Project Setup (Eclipse):

#### 1.Create New Project:

- Go to "File" -> "New" -> "Other..."
- Select "JPA Project" and click "Next."

#### 2.Project Details:

- Enter project name, target runtime, and JPA version.
- Choose "Basic JPA configuration."
- Click "Next."

#### 3.JPA Provider:

- Select "EclipseLink" and desired version.
- Choose "Disable Library Configuration."

#### 4.Database Connection:

- Select the previously created MySQL connection.

## JPA Project Structure:

- **Source:** Contains Java source files for your application.
- **Build:** Used for testing and building the project.
- **META-INF:** Contains persistence.xml for configuring JPA.

## persistence.xml Configuration:

1. Open persistence.xml in the META-INF folder.
2. Set Transaction-Type to "RESOURCE\_LOCAL."
3. Choose "populated from connection" for connection details.
4. Select the previously created MySQL connection.
5. Save the file.

## Entity Class:

- Represents database tables.
- Mapped to database tables using Object-Relational Mapping (ORM).
- Follows conventions for table and column names:
  - Entity class name suggests table name.
  - Variable names suggest column names.
- Annotated with @Entity to mark it as a JPA entity.
- Primary key identified by @Id annotation.

## Creating an Entity Class:

1. Create a JPA project.
2. Configure database connection in persistence.xml.
3. Create a package in the source folder.
4. Right-click on package -> New -> JPA Entity.
5. Enter class name and choose variables for table columns.
6. Select the primary key field.
7. Finish.

## Generating Tables from Entity Class:

1. Right-click project -> JPA Tools.
2. Select "Generate Tables from Entities."
3. Finish.
4. Verify table creation in your database application.

## Key Differences:

### Connection Establishment:

- JDBC uses a static method `DriverManager.getConnection(url)`.
- Hibernate uses `Persistence.createEntityManagerFactory("ProjectName")` to create an `EntityManagerFactory`.

➤ **Platform:**

- ❑ JDBC uses static platform types for executing queries.
- ❑ Hibernate uses the EntityManager interface for managing entity object states.

**Diff b/w JDBC vs. Hibernate:**

Feature	JDBC	Hibernate
<b>Connection</b>	DriverManager.getConnection(url)	Persistence.createEntityManagerFactory("ProjectName")
<b>Return Type</b>	Connection interface	EntityManagerFactory interface
<b>Platform</b>	Static methods (Statement, PreparedStatement, CallableStatement)	EntityManager

## Hibernate EntityManager Operations (Formatted)

### EntityManager:

- Manages entity lifecycles in JPA applications.
- Used to interact with the database.

### Opening EntityManager:

```
EntityManagerFactory factory =  
Persistence.createEntityManagerFactory("ProjectName");  
EntityManager manager = factory.createEntityManager();
```

### Starting Transaction:

```
manager.getTransaction().begin();
```

### Database Operations:

- Perform operations like insert, update, delete, and retrieve.

### Saving Changes:

- manager.getTransaction().commit();

### Closing Resources:

```
manager.close();  
factory.close();
```

### Insert Operation (persist())

1. Create an entity object.
2. Initialize object variables using setters.
3. Call manager.persist(entityObject).

### Syntax:

```
manager.persist(entityObject);
```

### Retrieve Record (find())

- Use manager.find(EntityClass.class, primaryKeyValue).

### Syntax:

```
EntityClass record = manager.find(EntityClass.class,  
primaryKeyValue);
```

### Note:

- find() returns null if the primary key doesn't exist.

### Update Operation

1. Use find() to retrieve the record.
2. Modify object state using setters.
3. No separate update method required. Changes are saved in commit().

### **Example:**

```
Employee emp = manager.find(Employee.class, 103);
emp.setSalary(45000);
manager.getTransaction().commit();
```

### **Delete Operation (remove())**

1. Use find() to retrieve the record (optional for safety).
2. Call manager.remove(entityObject).

### **Syntax:**

```
manager.remove(entityObject);
```

# Course : Java

# Part : Spring data JPA

## Spring Data JPA

Spring Data JPA is a high-level abstraction layer for JPA repositories. It simplifies data access by providing a repository interface-based approach.

### Creating a Maven Project:

1. Create a Maven project with the spring-boot-starter-data-jpa dependency.
2. Add dependencies for database driver (e.g., MySQL Connector/J), Hibernate, and Spring Data JPA.

### Configuring Persistence.xml:

- Configure the persistence unit name, JPA provider, and database connection details.

### Entity Class:

- Annotate the class with @Entity.
- Define fields and getters/setters.
- Use @Id to mark the primary key field.

### Repository Interface:

- Extend JpaRepository<T, ID> interface.
- Specify the entity type (T) and ID type (ID).
- Inherit CRUD methods: save(), deleteById(), findById(), findAll(), delete().

## Example:

```
@Entity  
public class User {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    private String email;  
    // Getters and setters  
}
```

```
public interface UserRepository extends JpaRepository<User,  
Long> {  
    // Custom query methods can be added here  
}
```

### Spring Configuration:

- Configure DataSource and EntityManagerFactory beans.
- Use @EnableJpaRepositories annotation to scan for repository interfaces.

## diff b/w Spring Data JPA and Spring JPA?

Feature	Spring Data JPA	JPA
Definition	A framework built on top of JPA to simplify data access.	A specification for object-relational mapping in Java.
Purpose	To reduce boilerplate code and simplify data access operations.	To define the standard for how Java objects are mapped to relational databases.
Repository Implementation	Automatically generates implementations for repository interfaces.	Requires manual implementation of repository interfaces.
Querying	Supports method-based querying, JPQL, and native SQL.	Uses JPQL or native SQL for complex queries.
Integration with Spring	Seamlessly integrates with other Spring components.	Can be integrated with Spring but requires more configuration.
Complexity	Simpler to use, with a more developer-friendly API.	More complex to use, especially for complex queries.

### Example:

```
@Configuration  
@EnableJpaRepositories("com.example.repository")  
public class AppConfig {  
    @Bean  
    public DataSource dataSource() { // code }  
    @Bean  
    //public LocalContainerEntityManagerFactoryBean  
    entityManagerFactory() { // code }  
    @Bean  
    transactionManager() {  
        JpaTransactionManager transactionManager = new  
        JpaTransactionManager();  
        transactionManager.setEntityManagerFactory(  
            entityManagerFactory().getObject());  
        return transactionManager;  
    }  
}
```

### Key Points:

- Spring Data JPA simplifies database operations by providing a repository-based approach.
- It automatically generates implementations for common CRUD operations.
- Custom query methods can be defined using JPQL or Spring Data JPA's query methods.
- Consider using `@Query` annotation or `@NamedQuery` for complex queries.
- Leverage Spring Data JPA's features like pagination, sorting, and specification for advanced data access scenarios.

## JpaRepository

It provides a set of predefined methods for common database operations.

### Core Methods:

#### •**save(Entity entity):**

- Persists or updates an entity.
- Returns the saved entity.

#### •**deleteById(ID id):**

- Deletes an entity by its primary key.
- Throws EmptyResultDataAccessException if no entity is found.

#### •**findById(ID id):**

- Returns an Optional<Entity> containing the entity if found, otherwise an empty Optional.

#### •**findAll():**

- Returns a List of all entities.

#### •**delete(Entity entity):**

- Deletes an entity.

## Custom Query Methods:

- Spring Data JPA allows creating custom query methods using method names.
- Use keywords like findBy, getBy, findDistinctBy, countBy, etc., followed by property names and keywords like And, Or, Not, Between, LessThan, GreaterThan, LessThanEqual, GreaterThanEqual, Like, StartingWith, EndingWith, Containing, OrderBy, Asc, and Desc.

### Example:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findByFirstName(String firstName);  
    List<User> findByFirstNameAndLastName(String firstName, String lastName);  
    List<User> findByAgeGreaterThanOrEqual(int age);}
```

## Spring Data JPA Specifications:

- Create custom query specifications using Specification interface.
- Combine multiple criteria using CriteriaBuilder and Predicate.
- Dynamically construct complex queries.

## Example:

```
public interface UserRepository extends JpaRepository<User,  
Long>, JpaSpecificationExecutor<User> { // ... }  
  
Usage: Specification<User> spec = (root, query,  
criteriaBuilder) -> {  
  
    return criteriaBuilder.and(  
  
        criteriaBuilder.equal(root.get("firstName"), "John"),  
        criteriaBuilder.greaterThanOrEqualTo(root.get("age"), 30)  
    );  
  
    List<User> users = userRepository.findAll(spec);
```

## Creating Custom Methods:

- Use keywords like findBy, readOnlyBy, getBy, etc.
- Combine keywords with variable names from the entity class.
- Utilize operators like And, Or, Not, Between, etc.

## Example:

```
public interface UserRepository extends JpaRepository<User,  
Long> {  
  
    User findByEmailIdAndPassword(String email, String  
    password);  
}
```

# Course : Java

# Part : Spring Web MVC

## Spring Web MVC - Model-View-Controller Pattern

### MVC Design Pattern:

- A development approach for building web applications.
- Separates the application into three layers:
  - **Model:** Holds application data (POJOs).
  - **View:** Renders the UI (JSP/HTML files).
  - **Controller:** Handles user requests and business logic.

### Benefits:

- Improved code organization and maintainability.
- Easier testing and reusability of components.

### Spring MVC Implementation:

- Spring MVC provides a framework for building MVC applications.
- Leverages key concepts like dependency injection and annotations.

### Components:

- **Model:** POJO classes representing application data.
- **View:** JSP or HTML templates for displaying the UI.
- **Controller:** Handles user requests, interacts with the model, and returns views.

### DispatcherServlet (Front Controller):

- Receives requests from the user.
- Maps requests to appropriate controllers.
- Manages the application flow.

### Configuration:

- DispatcherServlet configured in web.xml file.
- Spring configuration file details view resolver and beans.

### View Resolver:

- Responsible for resolving view names to actual JSP/HTML templates.
- Uses InternalResourceViewResolver by default.
- Configured with prefix and suffix for view paths.

## Creating a Spring MVC Application:

1. Configure DispatcherServlet in web.xml.
2. Create a Spring configuration file with view resolver configuration.
3. Develop a controller class annotated with @Controller.
4. Create view templates (JSP/HTML files).

## Benefits of Spring MVC:

- Simplified MVC implementation.
- Easier application development and testing.
- Leverage Spring features like dependency injection and annotations.

## Setting Up a Spring MVC Project

### 1. Project Setup:

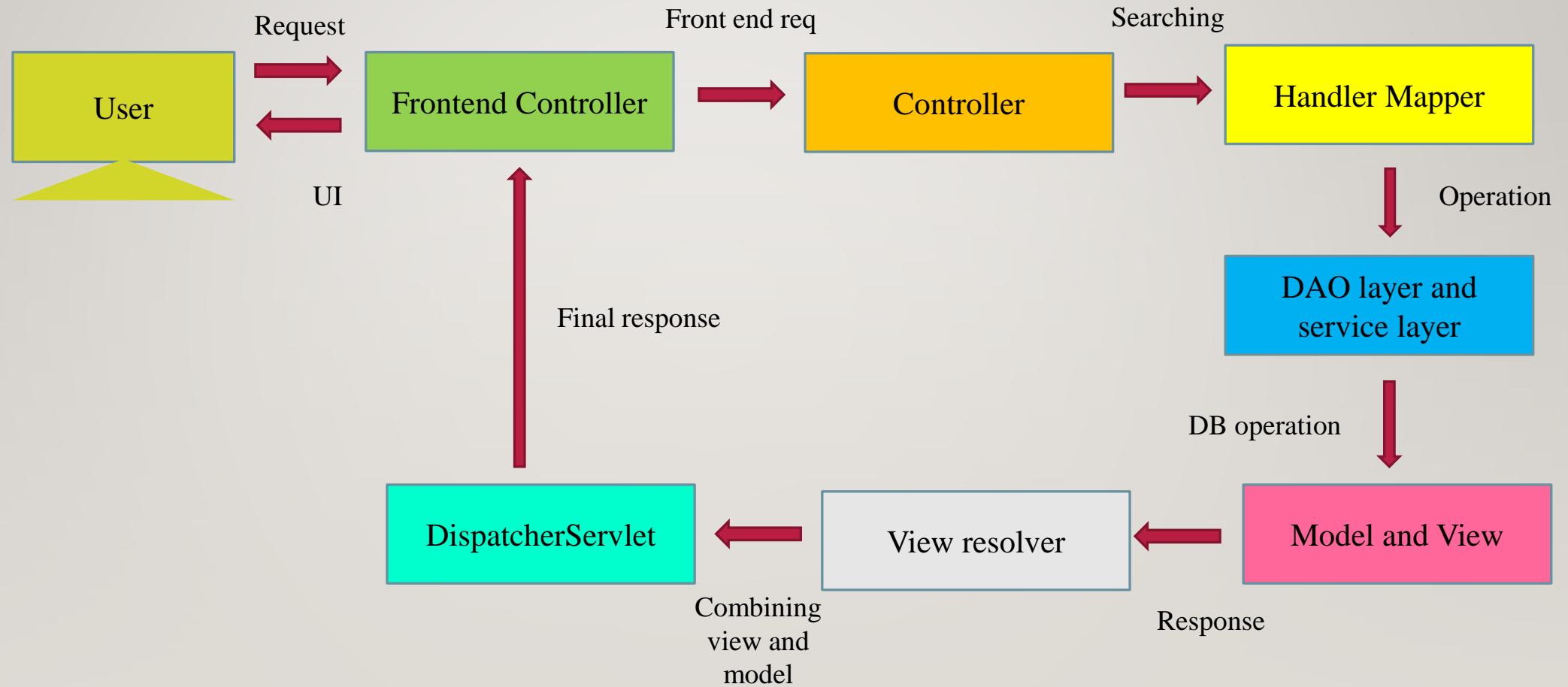
- ❑ **Create a Maven Project:** Use a Maven archetype to create a new web application project.
- ❑ **Set Java Version:**
  - Right-click on the project, go to "Properties" -> "Java Build Path".
  - Add Library -> JRE System Library -> Select Java 1.8.

### Key Points:

- **DispatcherServlet:** The central controller of the Spring web application.
- **View Resolver:** Maps view names to physical views (JSPs, HTML, etc.).
- **@Controller:** Annotation to mark a class as a controller.
- **@RequestMapping:** Annotation to map HTTP requests to controller methods.
- **Model:** Used to pass data from the controller to the view.

By following these steps, you'll have a basic Spring MVC application set up. You can further customize it with more advanced features like form handling, validation, data binding, and more.

## Diagram of Spring MVC Architecture :



Course : Java  
Part : Spring Boot

Code With Ved Prakash



## Spring Boot

### A Simplified Approach to Spring Development

Spring Boot is a powerful tool that streamlines the development of Spring-based web applications. It offers two core features:

- **Auto-Configuration:** Spring Boot automatically configures your application based on the dependencies you include in your project. This eliminates the need for extensive manual configuration.
- **Opinionated Configuration:** Spring Boot provides default configurations for common scenarios, reducing the amount of boilerplate code you need to write.

### Key Benefits of Spring Boot:

- **Simplified Setup:** Quickly set up a Spring-based application with minimal configuration.
- **Dependency Management:** Easily manage dependencies and their versions.

- **Embedded Server:** Run your application directly without deploying it to a separate server.
- **Auto-Configuration:** Leverage automatic configuration for common scenarios.
- **Production-Ready Features:** Benefit from features like metrics, health checks, and security.

### Creating a Spring Boot Project:

#### Use Spring Initializr:

- Visit <https://start.spring.io/>
- Select your project dependencies (e.g., Spring Web, Spring Data JPA, Spring Security).
- Generate the project.

#### Use Your IDE:

- Create a new Spring Starter Project in your IDE (e.g., IntelliJ IDEA, Eclipse).
- Select your dependencies and project settings.

- @Getter @Setter @ToString @AllArgsConstructor  
@NoArgsConstructor @Data
- @Id @Repository, @Controller(o/p with frontend page), @Repository,
- @RestController(without frontend page);- @controller + @ResponseBody
- @Column
- (updatable = false):- first time only it will fill the data then it will not update.
- @Column(insertable = false):-Not necessary during insertion time.
- @CreatedDate :- automatically capture the created date.
- @CreatedBy :- Automatically populates the annotated field with the identifier of the user who created the entity.
- @MappedSuperclass :-act as a supper class in entity class  
  @EntityListeners(AuditingEntityListener.class):- act as a supper class in entity class.
- @Entity :-is use to create table as a entity class.

- @Table(name="xyz"):- it is also use to give name of the table in the db.
- @GeneratedValue(strategy = GenerationType.IDENTITY):- auto generated
- @Transactional :-using upside of method in case exception occurs it will rollback automatically.
- @Modifying :- when the query is not a select query, but instead performs an update, insert, or delete operation.
- @NotEmpty(message = "Branch Address cannot be a null ")
- @Schema(description = "Account type of Eazy Bank account", example = "Savings")
- @RequestMapping(path="/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})
- @GetMapping @PostMapping @PutMapping @PatchMapping @DeleteMapping
- @RequestBody:-map the json/xml data to our entity object.
- @ResponseBody :- map the entity object to json/xml format.
- @RequestParam, @QueryParam:- used to extract parameters from HTTP requests

- @ResponseStatus(value = HttpStatus.BAD\_REQUEST/NOT\_FOUND) :- we are declearing this up side of exception class/variable.
  - @ControllerAdvice/@RestControllerAdvice :- for handling any global exception up of global exception class regarding controller class.
  - @ExceptionHandler(Exception.class) :- class is going to handle the exception.
  - @Transactional :-any error will be come at run time it will roll back
  - @Modifying :- telling this method is going to modify
  - void deleteByCustomerId(Long customerId); //repository methods
  - For validating data eject dependency
- ```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- And Extend your global exception to the not mandatory extends

### ResponseEntityExceptionHandler

- @NotEmpty(message = "Name can not be a null or empty")
- @Size(min = 5, max = 30, message = "The length of the customer name should be between 5 and 30")
- @Email(message = "Email address should be a valid value")
- @Min(value = 0, message = "Price must be greater than or equal to 0")
- @Max(value = 10000, message = "Price must be less than or equal to 10000")
- @Future(message = "Event date must be in the future")
  - private Date eventDate;
- @Past(message = "Birthdate must be in the past")
- @AssertTrue(message = "User must accept terms");

```
private boolean acceptedTerms;
```

➤ @AssertFalse(message = "User must not be blocked")

```
private boolean blocked;
```

➤ @DecimalMin(value = "0.0", inclusive = false, message = "Price must be greater than 0") @DecimalMax(value = "10000.0", inclusive = true, message = "Price must be less than or equal to 10000") private BigDecimal price;

➤ @Pattern(regexp = "[a-zA-Z0-9]+", message = "Username must be alphanumeric")

➤ @SerializedName(value="customerId", alternate = {"customerID", "customerid"})

➤ @Pattern(regexp = "^(\\\$|[0-9]{10})", message = "Mobile number must be 10 digits")

➤ @PositiveOrZero(message = "Total loan amount paid should be equal or greater than zero")

➤ @PositiveOrZero(message = "Total loan amount paid should be equal or greater than zero")

➤ @Validated :- it is used to tell the class validate all the validation in controller class

➤ Spring Open API's (Swagger UI)

<dependency>

    <groupId>org.springdoc</groupId>

    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

    <version>2.3.0</version>

</dependency>

```

➤ @OpenAPIDefinition(
    info = @Info(
        title = "Accounts microservice REST API
Documentation",
        description = "Olive Bank Accounts microservice REST
API Documentation",
        version = "v1",
        contact = @Contact(
            name = "Prakash",
            email = "prakash@eazybytes.com",
            url = "https://olivecrypto.com"
        ),
        license = @License(
            name = "Apache 2.0",
            url = " https://olivecrypto.com"
        )),
    externalDocs = @ExternalDocumentation(
        description = "OliveBank Accounts microservice REST
API Documentation",
        url = " https://olivecrypto.com/swagger-ui.html"
    )
)

```

It is regarding swagger header, written up side of main method

```

➤ @Tag(
    name = "CRUD REST APIs for Accounts in OliveBank",
    description = "CRUD REST APIs in OliveBank to CREATE,
UPDATE, FETCH AND DELETE account details"
)
Head of the api's, written up side of class
➤ @Operation(
    summary = "Create Account REST API",
    description = "REST API to create new Customer & Account
inside OliveBank"
)
@ApiResponse({
    @ApiResponse(
        responseCode = "201",
        description = "HTTP Status CREATED"
    ),
    @ApiResponse(
        responseCode = "500",
        description = "HTTP Status Internal Server Error",
        content = @Content(
            schema = @Schema(implementation = ErrorResponseDto.class)
        )
    )
}) //It is use to give api description, written up side of method

```

- @Schema(
  - name = "Customer",
  - description = "Schema to hold Customer and Account information"
  - )
  - @Schema(description = "amount", maxLength = 9, minLength = 4, example = "3.00")
  - It is use to give detail about the deoclass/class/variable . Up side of class or veriable
  - 
  - @ComponentScans(
    - @ComponentScan("com.eazybytes.accounts.controller")
)
  - @EnableJpaRepositories("com.eazybytes.accounts.repository")
  - @EntityScan("com.eazybytes.accounts.model")
  - @EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")
  - If we are not declare main class as parent package then use this all annotation.
  - @ConfigurationProperties(prefix = "accounts") :- use to make class as a configuration public record

```
(String message, Map<String, String> contactDetails,
List<String> onCallSupport) {}  

And declare this property on application.yml  

accounts:  

  message: "Welcome to EazyBank accounts related local APIs "  

  contactDetails:  

    name: "John Doe - Developer"  

    email: "john@eazybank.com"  

  onCallSupport:  

    - (555) 555-1234  

    - (555) 523-1345  

//user diffin property  

  @Autowired  

  private AccountsContactInfoDto accountsContactInfoDto;  

  @Inject  

➤ 3Autowired :- same as auto wire but we want to migrate from java to python then use inject.
```

```
@GetMapping("/contact-info")
public ResponseEntity<AccountsContactInfoDto>
getContactInfo() {
    return ResponseEntity
        .status(HttpStatus.OK)
        .body(accountsContactInfoDto);
}
```

It will allow only read not write. Only getter method is their.  
@ConfigurationProperties(prefix = "accounts") :- main class  
use this for fetching all the configure property

## Sort Cut For Eclipse

- control + shift + T / control + shift +R : finding classes or interface in side work space.
- control + H : finding particular word or sentence in side workspace.
- control + F : finding particular word in the file or classes .
- control + right-click : redirect to the pointer class or interface.
- control + K : redirect to particular variable declaration or same key word.
- control + shift + F : format code in organized manner.
- control + shift + W : close all file together.

- control + spacebar : suggestion for importing which you have return key word.
- control + shift + O : remove unused import statement at a time.
- control + / (or) control + shift + / (or) // (or) control + shift + C : comment or multiline comment
- control + shift + c (or) control + shift+ \ : uncomment or multiline uncomment.
- control + O : give out line for current class
- control + 1 : generate return type what evet called method is returning
- alt + shift + Z : generate design making statement or try catch block

- alt + left arrow / right arrow :  
going previous or forward place
- shift + alt + down arrow / up-arrow :  
generate duplicate copy of line for current line
- control + D :  
delete 1 line
- control + shift + X / Y :  
convert selected word to upper case or lower case.
- function +f4 / f6 :  
move next line / method during debug
- control + L :  
go to particular line

@

@

@

@

@

@

@