

# Java Programming Language

## Naming Conventions in Java

Java is a case-sensitive programming language, meaning uppercase and lowercase letters are considered different. We must follow naming conventions for the following:

- Classes
- Interfaces
- Variables
- Methods
- Keywords
- Packages
- Constants

### Classes

In Java, a class name must start with an uppercase letter. If it contains multiple words, each inner word must start with an initial capital letter (PascalCase).

Examples:

<code>// Predefined classes</code>	<code>// User-defined classes</code>
<code>System</code>	<code>Test</code>
<code>File</code>	<code>DemoApp</code>
<code>FileWriter</code>	<code>ExampleApp</code>
<code>BufferedReader</code>	<code>ScannerDemo</code>

### Interfaces

An interface name follows the same convention as class names.

Examples:

<code>// Predefined interfaces</code>	<code>// User-defined interfaces</code>
<code>Runnable</code>	<code>ITest</code>
<code>Serializable</code>	<code>IDemoApp</code>
<code>ListIterator</code>	<code>IExampleDemo</code>

### Variables

A variable name must start with a lowercase letter. If it contains multiple words, each inner word should start with an initial capital letter (camelCase).

Examples:

<code>// Predefined variables</code>	<code>// User-defined variables</code>
<code>out</code>	<code>empId</code>
<code>in</code>	<code>studName</code>
<code>err</code>	<code>deptNo</code>
<code>length</code>	<code>courseFee</code>

## Methods

Method names follow the same convention as variable names.

Examples:

<code>// Predefined methods</code>	<code>// User-defined methods</code>
<code>getPriority()</code>	<code>getStudentInfo()</code>
<code>hashCode()</code>	<code>calculateBillAmt()</code>
<code>toString()</code>	<code>demoApp()</code>

## Keywords

All keywords should be declared in lowercase letters only.

`class, public, static, void, if, else, for, switch, etc.`

## Packages

All packages should be declared in lowercase letters only.

Examples:

<code>// Predefined packages</code>	<code>// User-defined packages</code>
<code>java.lang</code>	<code>com.shiva.www</code>
<code>java.util</code>	<code>com.google.www</code>
<code>java.io</code>	<code>com.ihub.www</code>

## Constants

All constants should be declared in uppercase letters, with words separated by underscores.

Examples:

```
// Predefined constants
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

```
// User-defined constants
int LIMIT = 10;
```

## Assignment

1. Class : QualityThought
2. Interface : IQualityThought
3. Variable : qualityThought
4. Method : qualityThought()
5. Package : com.qualitythought.www
6. Constant : QUALITY\_THOUGHT

## Interview Questions

Q: Which package is the default package in Java?

A: The java.lang package is the default package in Java.

Q: What is a package?

A: A package is a collection of classes and interfaces.

Q: What is a class?

A: It is a collection of variables and methods.

## Java Overview

- Version: Java 8
- JDK: JDK 11
- Creator: James Gosling
- Vendor: Oracle Corporation
- Open source: Yes
- Website: [www.oracle.com/in/java](http://www.oracle.com/in/java)
- Tutorials: [www.javatpoint.com](http://www.javatpoint.com), [www.w3schools.com](http://www.w3schools.com), [www.javaus.com](http://www.javaus.com), [www.dzone.com](http://www.dzone.com), etc.
- Download link: [JDK 11 Download](#)

# Setting Up Java Environment

## Steps to setup environmental variables

1. Ensure JDK 11 is installed successfully.
2. the "lib" directory from the "java\_home" folder. Example: C:\Program Files\Java\jdk-11\lib
3. Set up environmental variables:
  - User variables:
    - Variable name: CLASSPATH
    - Variable value: C:\Program Files\Java\jdk-11\lib;
  - System variables:
    - Variable name: path
    - Variable value: C:\Program Files\Java\jdk-11\bin;
4. Verify the setup by running these commands:
  - cmd> javap
  - cmd> java -version

## Steps to develop your first Java application

1. Ensure JDK 11 is installed and environmental setup is complete.
2. Create a "javaprogram" folder in the D drive.
3. Open a text editor (e.g., Notepad) and write a Hello World program:

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

4. Save the program with the same name as the class name (Test.java) inside the javaprogram folder.
5. Open the command prompt from the javaprogram location.
6. Compile the program: javac Test.java
7. Execute the program: java Test

## Sample Interview Question

Q: John and Jack are best friends in a town. They study together in the same class. One day, while going to school, they saw a poor person. John and Jack decided to help the needy person. John gave 100 rupees from his pocket, and Jack gave 50 rupees from his bag. Write a Java program to calculate their total contribution.

```
class Test {  
    public static void main(String[] args) {  
        int johnContribution = 100;  
        int jackContribution = 50;  
        int totalContribution = johnContribution + jackContribution;  
        System.out.println("Total contribution: " + totalContribution + "  
rupees");  
    }  
}
```

## History of Java

In 1990, Sun Microsystems initiated a project to develop software for consumer electronic devices that could be controlled by a remote, similar to a set-top box. The project was initially called the Stealth project and later renamed to the Green project.

James Gosling, Mike Sheridan, and Patrick Naughton were tasked with developing the project. They met in Aspen, Colorado, to start work with the Graphic System. James Gosling initially considered using C and C++ languages for the project. However, they faced issues with system dependency. Gosling then decided to create a new programming language that would be system-independent.

In 1991, they developed a programming language called Oak. Oak means strength and is also a coffee seed name. It is the national tree for many countries, including Germany, France, and the USA.

Later, in 1995, they renamed Oak to Java. Java is an island in Indonesia where the first coffee seed was produced. During the project's development, the team consumed a lot of coffee. Hence, the symbol of Java became a cup of coffee with a saucer.

## Interview Questions

Q: Who is the creator of Java?

A: James Gosling

Q: In which year was Java developed?

A: In 1995

Q: What was Java originally known as?

A: Oak

# Java Source File Structure

1. A Java program can have multiple classes.
2. If a Java program contains multiple classes, we need to identify which class contains the main method. That class is treated as the main class.
3. If a Java program contains multiple classes with main methods, we need to declare one class as public. That public class will be treated as the main class.

Example:

```
// File: A.java
public class A {
    public static void main(String[] args) {
        System.out.println("A-class");
    }
}

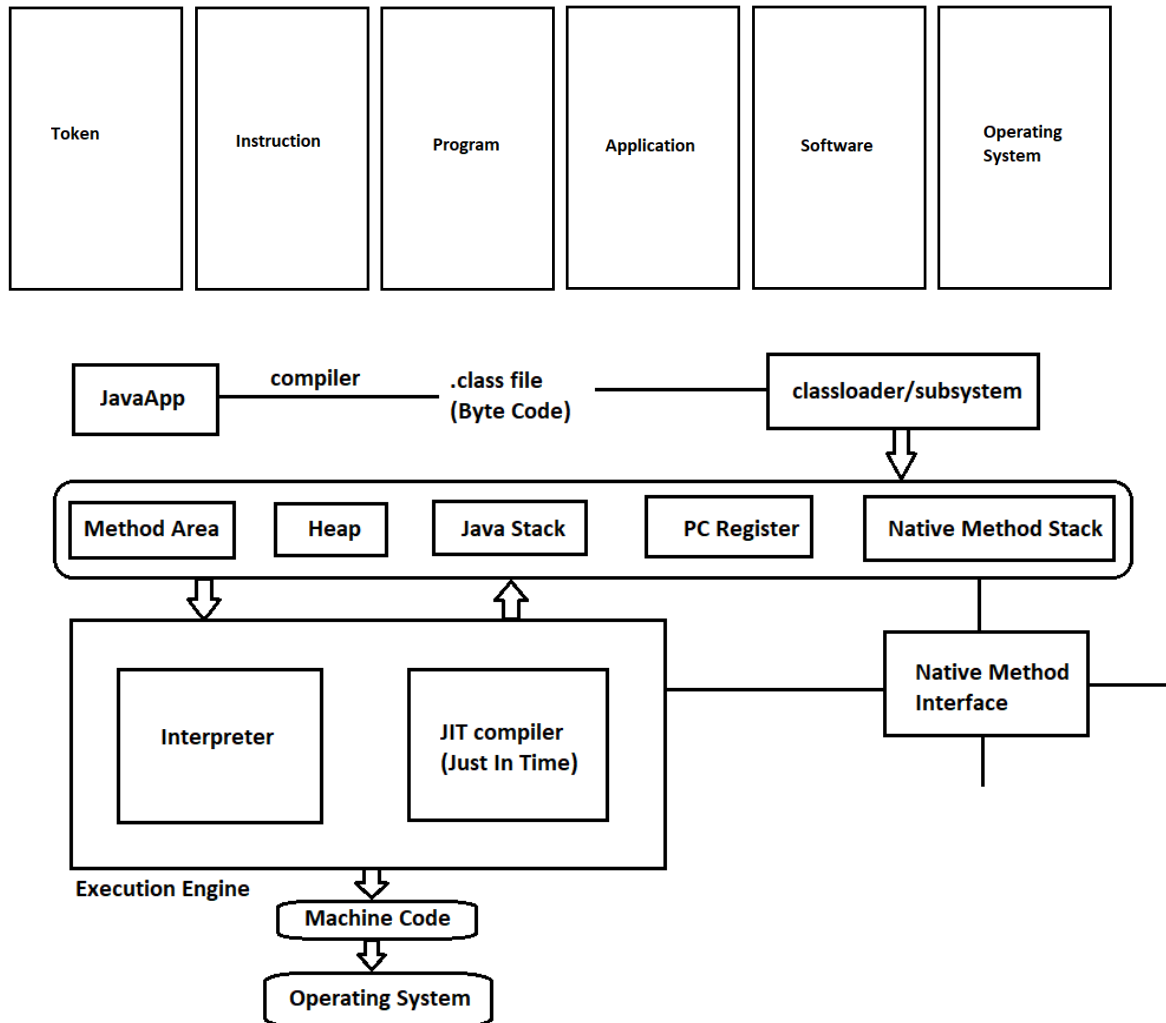
class B {
    public static void main(String[] args) {
        System.out.println("B-class");
    }
}

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

Compile: `javac A.java` (This will create A.class, B.class, and C.class files) Run:

- `java A` (A class will execute)
- `java B` (B class will execute)
- `java C` (C class will execute)

## Internal Architecture of JVM



Java application contains Java code instructions. When compiled, Java code instructions are converted to bytecode instructions in a .class file.

The JVM invokes a module called the classloader or subsystem to load all the bytecode instructions from the .class file. The classloader checks if these bytecode instructions are proper. If they are not proper, it will refuse execution. If they are proper, it allocates memory.

There are five types of memories in Java:

1. Method Area: Contains code of a class, variables, and methods.
2. Heap: Stores object creations.
3. Java Stack: Allocates memory for executing Java methods.
4. PC Register: Program counter register used to track the address of instructions.
5. Native Method Stack: Executes native methods.

The Execution Engine contains an interpreter and JIT (Just-In-Time) compiler. The interpreter executes the program line by line, while the JIT compiler increases the execution speed. Later, the JVM converts bytecode to machine code.

## Interview Questions

Q: What is a native method in Java?

A: A method developed using a language other than Java is called a native method.

Q: Can we execute a native method in Java directly?

A: No, we require a Native Method Interface.

Q: What is the JIT compiler?

A: It is a part of the JVM used to increase the execution speed of our program.

Q: How many types of memory are there in Java?

A: There are five types of memory in Java: Method Area, Heap area, Java Stack, PC Register, and Native Method Stack.

Q: How many classloaders are there in Java?

A: There are three classloaders in Java: Bootstrap classloader, Extension classloader, and Application/System classloader.

## Identifiers

An identifier is a name in Java. It can be a variable name, method name, class name, or label name.

Example:

```
class Test {  
    public static void main(String[] args) {  
        int x = 10;  
        System.out.println(x);  
    }  
}
```

Here, Test, main, args, and x are identifiers.



## Rules for Declaring Identifiers

1. Identifiers can use A-Z, a-z, 0-9, \_, and \$.

ex:

```
A-Z  
a-z  
0-9  
_  
$
```

2. Other characters will result in a compile-time error.

ex:

```
int emp$alary; //valid  
int emp@salary; //invalid  
class Test*App //invalid  
{  
}
```

3. Identifiers must start with a letter, underscore, or dollar symbol, but not with a digit.

ex:

```
int a1234; //valid  
int _abcd; //valid  
int $=20; //valid  
int 1abcd; //invalid
```

4. Reserved words cannot be used as identifier names.

ex:

```
int if; //invalid  
int else; //invalid  
int for; //invalid  
int do; //invalid
```

5. Every identifier is case-sensitive.

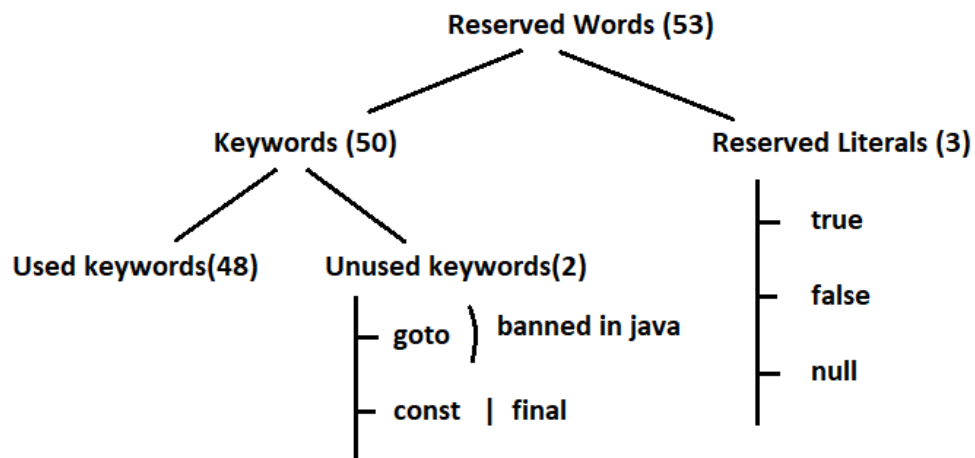
ex:

```
int number;  
int NUMBER;  
int NuMbEr;
```

6. There is no length limit for an identifier, but it's not recommended to use more than 15 characters.

## Reserved Words

Reserved words are identifiers reserved to associate some functionality or meaning. Java supports 53 reserved words, all declared in lowercase letters.

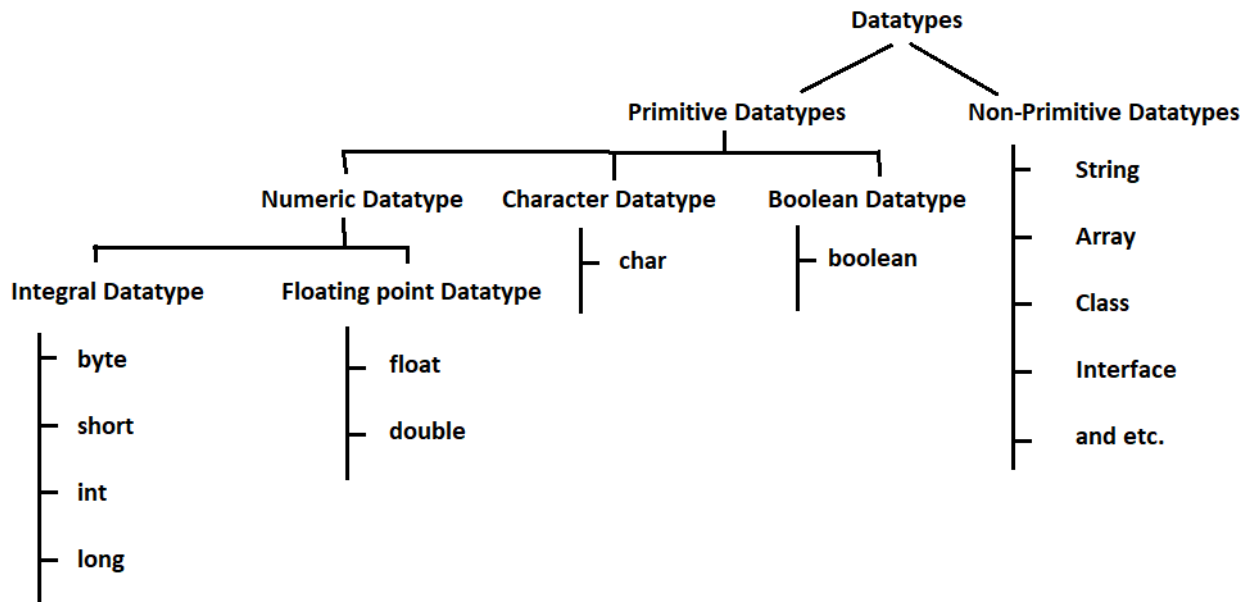


Categories of reserved words:

- Used with respect to class: package, import, class, interface, enum, extends, implements
- Used with respect to object: new, instanceof, this, super
- Used with respect to datatypes: byte, short, int, long, float, double, boolean, char
- Used with respect to modifiers: default, public, private, protected, final, static, abstract, transient, volatile, strictfp, synchronized, native
- Used with respect to return type: void
- Used with respect to flow control: if, else, do, while, for, break, continue, switch, case
- Used with respect to exception handling: try, catch, throw, throws, finally, assert

# Datatypes

A datatype describes what type of value we want to store inside a variable and how much memory should be allocated for a variable. In Java, datatypes are divided into two types: Primitive and Non-Primitive.



## byte

- Smallest datatype in Java
- Size: 1 byte (8 bits)
- Range: -128 to 127 ( $-2^7$  to  $2^7-1$ )

Examples:

```
byte b = 10;    // Valid
byte b = 130;   // Compile-time error
byte b = 10.5;  // Compile-time error
```

## short

- Rarely used datatype in Java
- Size: 2 bytes (16 bits)
- Range: -32,768 to 32,767 ( $-2^{15}$  to  $2^{15}-1$ )

Examples:

```
byte b = 10;
short s = b;    // Valid
short s = "hi"; // Compile-time error
short s = 10.5; // Compile-time error
```

## int

- Most commonly used datatype in Java
- Size: 4 bytes (32 bits)
- Range: -2,147,483,648 to 2,147,483,647 ( $-2^{31}$  to  $2^{31}-1$ )

Examples:

```
int i = true;    // Compile-time error
int i = "hi";    // Compile-time error
int i = 10.5;    // Compile-time error
int i = 'a';     // Valid, prints 97 (ASCII value of 'a')
```

Note: In Java, every character contains a universal Unicode value. For example, 'a' is 97 and 'A' is 65.

## long

If the int datatype is not sufficient to hold a large value, we need to use the long datatype.

Size: 8 bytes (64 bits) Range:  $-2^{63}$  to  $2^{63}-1$

Examples:

```
1) long l = "true";
   System.out.println(l); // Compile-Time Error

2) long l = false;
   System.out.println(l); // Compile-Time Error

3) long l = 10.5;
   System.out.println(l); // Compile-Time Error

4) long l = 'A';
   System.out.println(l); // 65
```

## float vs double

float:

- Use when 4 to 6 decimal points of accuracy are needed
- Size: 4 bytes (32 bits)
- Range:  $-3.4e38$  to  $3.4e38$
- To declare a float value, suffix with 'f' or 'F' Example: 10.5f

double:

- Use when 14 to 16 decimal points of accuracy are needed
- Size: 8 bytes (64 bits)
- Range:  $-1.7e308$  to  $1.7e308$
- To declare a double value, suffix with 'd' or 'D' (optional) Example: 10.5d

Examples for float:

```
1) float f = 10;
   System.out.println(f); // 10.0

2) float f = 'a';
   System.out.println(f); // 97.0

3) float f = "hi";
   System.out.println(f); // Compile-Time Error

4) float f = true;
   System.out.println(f); // Compile-Time Error

5) float f = 10.5f;
   System.out.println(f); // 10.5
```

Examples for double:

```
1) double d = 10;
   System.out.println(d); // 10.0

2) double d = 'a';
   System.out.println(d); // 97.0

3) double d = "hi";
   System.out.println(d); // Compile-Time Error
```

```
4) double d = true;
   System.out.println(d); // Compile-Time Error

5) double d = 10.5d;
   System.out.println(d); // 10.5
```

## boolean

Used to represent boolean values: true or false. Size: Not applicable (1 bit) Range: Not applicable

Examples:

```
1) boolean b = "true";
   System.out.println(b); // Compile-Time Error

2) boolean b = TRUE;
   System.out.println(b); // Compile-Time Error

3) boolean b = true;
   System.out.println(b); // true
```

## char

Represents a single character enclosed in single quotation marks. Size: 2 bytes (16 bits) Range: 0 to 65535

Examples:

```
1) char ch = 'a';
   System.out.println(ch); // a

2) char ch = 65;
   System.out.println(ch); // A

3) char ch = "a";
   System.out.println(ch); // Compile-Time Error
```

Datatypes	Size	Range	Wrapper class	Default value
byte	1 byte	-128 to 127	Byte	0
short	2 bytes	-32768 to 32767	Short	0
int	4 bytes	-2147483648 to 2147483647	Integer	0
long	8 bytes	-2 <sup>63</sup> to 2 <sup>63</sup> -1	Long	0L
float	4 bytes	-3.4e38 to 3.4e38	Float	0.0f
double	8 bytes	-1.7e308 to 1.7e308	Double	0.0d
boolean	-	-	Boolean	false
char	2 bytes	0 to 65535	Character	0(space)

## Interview questions

Q) Write a java program to display range of byte datatype?

byte range : -128 to 127

ex:

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

Q) Write a java program display range of int datatype?

int range : -2147483648 to 2147483647

ex:

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

Q)Is Java purely object oriented or not?

No, java will not consider as purely object oriented programming language because it does not support many OOPS concepts like multiple inheritance, operator overloading and more over we depends upon primitive datatypes which are non-objects.

## Types of variables in java

A name which is given to a memory location is called variable.

Purpose of variable is used to store the data.

In java, we have two types of variables.

### 1) Primitive variables

It is used to represent **primitive values**.

### 2) Reference variables

It **is** used **to** represent object **reference**.

ex:

```
Student s=new Student();
      |
reference variable
```

Based on the position and execution these variables are divided into three types.

1. Instance variables / Non-static variables
2. Static variables / Global variables
3. Local variables / Temporary variables / Automatic variables



# 1) Instance variables

A value of a variable which is varied(changes) from object to object is called instance variable.

Instance variable will be created at the time of object creation and it will destroy at the time of object destruction.Hence scope of instance variable is same as scope of an object.

Instance variable will store in heap area as a part of an object.

Instance variable must and should declare immediately after the class but not inside methods, blocks and constructors.

Instance variable we access directly from instance area but we can't access directly from static area.

To access instance variable from static area we need to create object reference.

```
ex:
class Test
{
    //instance variable
    int i=10;

    public static void main(String[] args)
    {
        System.out.println(i); // C.T.E
    }
}
```

Example:

```
class Test
{
    //instance variable
    int i=10;

    public static void main(String[] args)
    {
        Test t=new Test();
        System.out.println(t.i); //10
    }
}
```

## Note:

If we won't initialize any value to instance variable then JVM will initialize default values.

Example:

```
class Test
{
    //instance variable
    boolean b;

    public static void main(String[] args)
    {
        Test t=new Test();
        System.out.println(t.b); //false
    }
}
```

Example:

```
class Test
{
    //instance variable
    int i=10;

    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test();

        System.out.println(t1.i); //10
        System.out.println(t2.i); //10

        t1.i=100;

        System.out.println(t1.i); //100
        System.out.println(t2.i); //10
    }
}
```

Example:

```

class Test
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
    //non-static method
    public void m1()
    {
        System.out.println("instance-method");
    }
}

```

## 2) Static variables

- A value of a variable which is not varied(change) from object to object is called static variable.
- Static variable will be created at the time of classloading and it will destroy at the time of class unloading.Hence scope of static variable is same as scope of .class file.
- Static variables will store in method area.
- Static variable must and should declare immediately after the class using static keyword but not inside methods, blocks and constructors.
- Static variable we can access directly from instance area as well as from static area.
- Static variable we can access by using object reference and class name.

Example:

```

class Test
{
    //static variable
    static int i=10;

    public static void main(String[] args)
    {
        System.out.println(i); // 10

        Test t=new Test();
        System.out.println(t.i); // 10

        System.out.println(Test.i);//10
    }
}

```

```
}
```

## Note:

If we won't initialize any value to static variable then JVM will initialize default values.

Example:

```
class Test
{
    //static variable
    static String s;

    public static void main(String[] args)
    {
        System.out.println(s); // null

        Test t=new Test();
        System.out.println(t.s); // null

        System.out.println(Test.s); // null
    }
}
```

Example:

```
class Test
{
    //static variable
    static int i=10;

    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test();

        System.out.println(t1.i); //10
        System.out.println(t2.i); //10

        t1.i=100;
    }
}
```

```

        System.out.println(t1.i);//100
        System.out.println(t2.i);//100
    }
}

```

Example:

```

class Test
{
    public static void main(String[] args)
    {
        m1();

        Test t=new Test();
        t.m1();

        Test.m1();
    }
    //static method
    public static void m1()
    {
        System.out.println("static-method");
    }
}

```

### 3) Local variables

- To meet temporary requirements a programmer will declare some variables inside methods, blocks and constructors such type of variables are called local variables.
- Local variable will be created at the time of execution block and it will destroy when execution block is executed. Hence scope of local variable is same as scope of execution block where it is declared.
- Local variables will store in java stack.

Example:

```

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

```

```
        //local variable
        int i=10;
        System.out.println(i);//10
    }
}
```

## Note:

If we won't initialize any value to local variable then JVM will not initialized default values.

Example:

```
class Test
{
    public static void main(String[] args)
    {
        //local variable
        int i;
        System.out.println(i);
    }
}
```

o/p:

C.T.E : variable i might not have been initialized

A local variable will accept only one modifier i.e final.

Example:

```
class Test
{
    public static void main(String[] args)
    {
        //local variable
        final int i=10;

        System.out.println(i);//10
    }
}
```

# Interview Question

Q) What is Literal ?

A value which is assigned to a variable is called literal.

A value which is not change during the program execution is called literal.

Example:

```
int    i    = 10;
|      |    |
|      |    |____ value of a variable / Literal
|      |____ variable name / identifier
|____ datatype / keyword
```

## Main method

- Our program contains main method or not.
- Either it is properly declare or not. It is not a responsibility of a compiler to check.
- It is a liability of a JVM to check for main method.
- If JVM won't find main method then it will throw one runtime error called main method not found.
- JVM always look for main method with following signature.

## signature

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

If we perform any changes in above signature then JVM will throw one runtime error called main method not found.

Q) Explain main method in java?

## public

```
JVM wants to call main method from anywhere.
```

## static

JVM wants to call main method without using object reference.

## void

Main method does not return anything to JVM.

## main

It is an identifier given to main method.

## String[] args

It is a command line arguments

We can perform following changes in main method.

1. Order of modifiers is not important incase of public static we can declare static public also. ex: static public void main(String[] args)
2. We can declare String[] in following acceptable formats.

Example:

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

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

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

3. We can replace args with any java valid identifier.

Example:

```
public static void main(String[] ihub)
```

4. We can change String[] with var-arg parameter.

Example:

```
public static void main(String... args)
```

5. Main method will accept following modifiers.

Example:



synchronized strictfp final

# Command line arguments

Arguments which are passing through command prompt such type of arguments are called command line arguments.

In command line arguments we need to pass our input values at runtime.

Example: javac Test.java

```
java    Test    101    raja    M    1000.0
      |      |      |      |      |
      |      |      |      |      |_____ args[3]
      |      |      |      |_____ args[2]
      |      |_____ args[1]
      |_____ args[0]
```

Example:

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(args[0]);
        System.out.println(args[1]);
        System.out.println(args[2]);
        System.out.println(args[3]);
    }
}
```

## System.out.println()

It is a output statement in java.

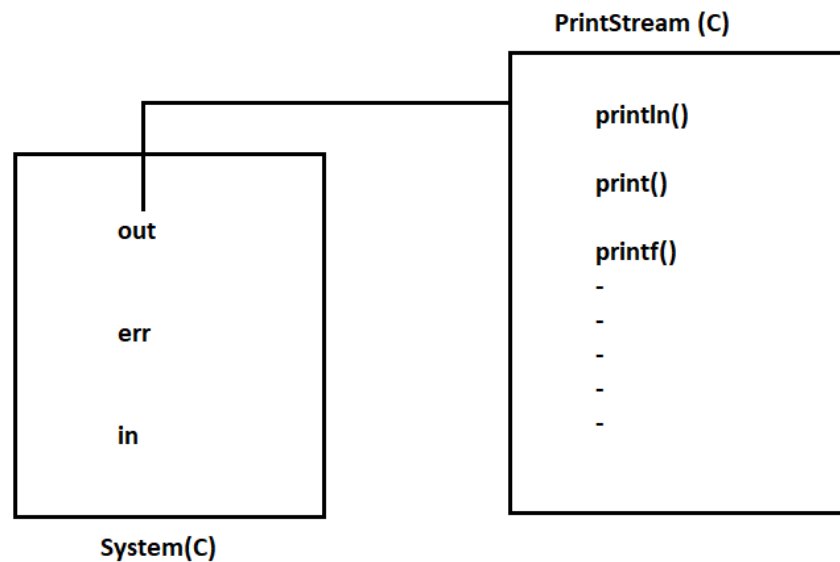
Whenever we want to display any userdefined statements or data then we need to use output stmt.

**syntax:**

```

static variable
    |
System.out.println();
    |
predefined      predefined method final class

```



Example

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");
        System.out.print("stmt2");
        System.out.printf("stmt3");
    }
}

```

## Various ways to display the data in java

1. `System.out.println("Hello World");`
2. `int i=10; System.out.println(i);`
3. `int i=10,j=20; System.out.println(i+" "+j);`
4. `int i=1,j=2,k=3; System.out.println(i+" "+j+" "+k);`

# Fully Qualified Name

Fully qualified name means we will declare a class or interface along with package name.

Fully qualified name is used to improve readability of our code.

Example:

```
class Test
{
    public static void main(String[] args)
    {
        java.util.Date d=new java.util.Date();
        System.out.println(d);
    }
}
```

## Import statements

Whenever we use import statement we should not use fully qualified name.

Using short name also we can achieve.

In java, we have three import statements.

1. Explicit class import
2. Implicit class import
3. Static import

### 1) Explicit class import

This type of import statement is highly recommended because it improves readability of our code.

Example:

```
import java.time.LocalDate;
import java.time.LocalTime;
class Test
{
    public static void main(String[] args)
```

```
{
    LocalDate date=LocalDate.now();
    System.out.println(date);

    LocalTime time=LocalTime.now();
    System.out.println(time);
}
```

## 2) Implicit class import

This type of import statement is not recommended to use because it will reduce readability of our code.

Example:

```
import java.time.*;
class Test
{
    public static void main(String[] args)
    {
        LocalDate date=LocalDate.now();
        System.out.println(date);

        LocalTime time=LocalTime.now();
        System.out.println(time);
    }
}
```

## 3) Static Import

Using static import we can call static members directly. Often use of static import makes our program complex and unreadable.

Example:

```
import static java.lang.System.*;
class Test
{
    public static void main(String[] args)
    {
```

```
        out.println("stmt1");
        out.println("stmt2");
        out.println("stmt3");
    }
}
```

Example with System.exit(0):

```
import static java.lang.System.*;
class Test
{
    public static void main(String[] args)
    {
        out.println("stmt1");
        exit(0);
        out.println("stmt3");
    }
}
```

## Basic Java Programs

### Sum of Two Numbers

```
import java.util.Scanner;
class Example1
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the first number :");
        int a = sc.nextInt();

        System.out.println("Enter the second number :");
        int b = sc.nextInt();

        //logic
        int c = a + b;

        System.out.println("sum of two numbers is =" + c);
    }
}
```

```
}
```

## Sum of Two Numbers Without Using Third Variable

```
import java.util.Scanner;
class Example2
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the first number :");
        int a = sc.nextInt();

        System.out.println("Enter the second number :");
        int b = sc.nextInt();

        System.out.println("sum of two numbers is =" + (a + b));
    }
}
```

## Square of a Given Number

```
import java.util.Scanner;
class Example3
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number :");
        int n = sc.nextInt();

        //logic
        int square = n * n;

        System.out.println("square of a given number is =" + square);
    }
}
```

## Cube of a Given Number

```

import java.util.Scanner;
class Example4
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number :");
        int n = sc.nextInt();

        //logic
        int cube = n * n * n;

        System.out.println("cube of a given number is =" + cube);
    }
}

```

**Alternative approach using Math.pow():**

```

import java.util.Scanner;
class Example4
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number :");
        int n = sc.nextInt();

        //logic
        int cube = (int)Math.pow(n, 3);

        System.out.println("cube of a given number is =" + cube);
    }
}

```

## Area of a Circle

```

import java.util.Scanner;
class Example5

```

```

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

        System.out.println("Enter the radius :");
        int r = sc.nextInt();

        //logic
        float area = 3.14f * r * r;

        System.out.println("Area of a circle is =" + area);
    }
}

```

## Perimeter of a Circle

```

import java.util.Scanner;
class Example6
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the radius :");
        float r = sc.nextFloat();

        //logic
        float perimeter = 2 * 3.14f * r;

        System.out.println("Perimeter of a circle is =" + perimeter);
    }
}

```

## Swapping of Two Numbers

```

import java.util.Scanner;
class Example7
{
    public static void main(String[] args)

```



```

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

    System.out.println("Enter the first number :");
    int a = sc.nextInt();

    System.out.println("Enter the second number :");
    int b = sc.nextInt();

    System.out.println("Before swapping a =" + a + " and b =" + b);

    //logic
    int temp = a;
    a = b;
    b = temp;

    System.out.println("After swapping a =" + a + " and b =" + b);
}
}

```

## Swapping of Two Numbers Without Using Third Variable

```

import java.util.Scanner;
class Example8
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the first number :");
        int a = sc.nextInt();

        System.out.println("Enter the second number :");
        int b = sc.nextInt();

        System.out.println("Before swapping a =" + a + " and b =" + b);

        //logic
        a = a + b;
        b = a - b;
        a = a - b;
    }
}

```

```

        System.out.println("After swapping a =" + a + " and b =" + b);
    }
}

```

## CGPA to Percentage

```

import java.util.Scanner;
class Example9
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the cgpa :");
        float cgpa = sc.nextFloat();

        float percentage = cgpa * 9.5f;

        System.out.println("CGPA to Percentage is =" + percentage);
    }
}

```

## Calculate TDS from Salary

```

import java.util.Scanner;
class Example10
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the salary :");
        int salary = sc.nextInt();

        float tds = (float)salary * 10 / 100;

        System.out.println("10 percent of TDS is =" + tds);
    }
}

```

## Typecasting

The process of converting from one datatype to another datatype is called typecasting.

In Java, typecasting can be performed in two ways:

1. Implicit typecasting
2. Explicit typecasting

## 1. Implicit Typecasting

If we want to store a small value into a bigger variable, we need to use implicit typecasting. A compiler is responsible for performing implicit typecasting. There is no possibility of losing information. It is also known as widening or upcasting.

Example:

```
class Test
{
    public static void main(String[] args)
    {
        byte b = 10;

        int i = b;

        System.out.println(i); // 10
    }
}
```

ex:

```
byte ---> short
                                     --->
                                     int ---> long ---> float ---> double
                                     --->
                                     char
```

## 2. Explicit Typecasting

If we want to store a bigger value into a smaller variable, we need to use explicit typecasting. A programmer is responsible for performing explicit typecasting. There is a possibility of losing information. It is also known as Narrowing or Downcasting.

```
ex:
    byte <--- short
           <---
           int <--- long <--- float <--- double
           <---
           char
```

Example:

```
class Test
{
    public static void main(String[] args)
    {
        float f = 10.5f;

        int i = (int)f;

        System.out.println(i); // 10
    }
}
```

## Types of Blocks

A block is a set of statements which is enclosed in curly braces i.e {}.

In Java, we have three types of blocks:

1. Instance block
2. Static block
3. Local block

### 1. Instance Block

Instance block is used to initialize the instance variables. Instance block must and should be declared immediately after the class. Instance block will execute when an object is created.

Example:

```
class Test
{
```

```

//instance block
{
    System.out.println("instance-block");
}

public static void main(String[] args)
{
    System.out.println("main-method");
    Test t = new Test();
}
}

```

Output:

```

main-method
instance-block

```

## 2. Static Block

Static block is used to initialize the static variables. Static block must and should be declared immediately after the class using the static keyword. Static block will execute at the time of class loading.

Example:

```

class Test
{
    //static block
    static
    {
        System.out.println("static-block");
    }

    public static void main(String[] args)
    {
        System.out.println("main-method");
    }
}

```

Output:

```

static-block

```

main-method

### 3. Local Block

Local block is used to initialize the local variables. Local block must and should be declared inside methods. Local block will execute just like a normal statement.

Example:

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");

        //local block
        {
            System.out.println("stmt2");
        }

        System.out.println("stmt3");
    }
}
```

Output:

```
stmt1
stmt2
stmt3
```

## Java Operators

1. Assignment Operators
2. Conditional/Ternary Operator
3. Logical Operators
4. Bitwise Operators
5. Arithmetic Operators
6. Relational Operators

## 7. Shift Operators

# Assignment Operators

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;

        i = 20;

        i = 30;

        System.out.println(i); //30
    }
}
```

Note: Reinitialization is possible in Java.

```
class Test
{
    public static void main(String[] args)
    {
        final int i = 10;

        i = 20; // This will cause a compile-time error

        i = 30; // This will cause a compile-time error

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

```
class Test
{
    //global variable
    static int i = 10;

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

```

        int i = 20;

        System.out.println(i); //20
    }
}

```

Note: Here priority goes to local variable.

```

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

        i = 1,2,3,4,5; // This will cause a compile-time error

        System.out.println(i);
    }
}

```

```

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

        i += 5; // i = i + 5

        System.out.println(i); //15
    }
}

```

```

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

        i -= 15; // i = i - 15
    }
}

```



```
        System.out.println(i); //-5
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;

        i *= 2; // i = i * 2

        System.out.println(i); //20
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;

        i /= 5; // i = i / 5;

        System.out.println(i); //2
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;

        i /= 15; // i = i / 15;

        System.out.println(i); //0
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;

        i %= 15; // i = i % 15;

        System.out.println(i); //10
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;

        i %= 5; // i = i % 5;

        System.out.println(i); //0
    }
}
```

## Conditional/Ternary Operator

Syntax:

```
(condition) ? value1 : value2;
```

```
class Test
{
    public static void main(String[] args)
    {
        boolean b = (5 > 2) ? true : false;
        System.out.println(b); //true
    }
}
```

```

class Test
{
    public static void main(String[] args)
    {
        String s = (5 > 20) ? "Hi" : "Bye";
        System.out.println(s); //Bye
    }
}

```

```

class Test
{
    public static void main(String[] args)
    {
        char ch = (false) ? 'T' : 'F';
        System.out.println(ch); //F
    }
}

```

Example: Finding the greatest of two numbers

```

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

        System.out.println("Enter the first number :");
        int a = sc.nextInt();

        System.out.println("Enter the second number :");
        int b = sc.nextInt();

        int max = (a > b) ? a : b;
        System.out.println("Greatest of two numbers is =" + max);
    }
}

```

Example: Finding the greatest of three numbers

```

import java.util.Scanner;

```

```

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

        System.out.println("Enter the first number :");
        int a = sc.nextInt();

        System.out.println("Enter the second number :");
        int b = sc.nextInt();

        System.out.println("Enter the third number :");
        int c = sc.nextInt();

        int max = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);

        System.out.println("Greatest of three numbers is =" + max);
    }
}

```

## Logical Operators

### Logical AND (&&) Operator

Logical AND operator deals with boolean values either true or false.

Truth table:

```

T T = T
T F = F
F T = F
F F = F

```

```

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

```

```
        System.out.println(false && false); // false
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        boolean b = (5 > 2) && (6 < 10);

        System.out.println(b); //true
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        boolean b = (5 > 2) && (6 < 1);

        System.out.println(b); //false
    }
}
```

## Logical OR (||) Operator

Logical OR operator deals with boolean values either true or false.

Truth table:

```
T T = T
T F = T
F T = T
F F = F
```

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

```
        System.out.println(true || false); //true
        System.out.println(false || true); //true
        System.out.println(false || false); //false
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        boolean b = (5 > 20) || (6 < 1);

        System.out.println(b); // false
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        boolean b = (5 > 20) || (6 < 10);

        System.out.println(b); // true
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        boolean b = (5 > 20) || (6 < 10) && (10 >= 10);

        System.out.println(b); // true
    }
}
```

## Logical NOT (!) Operator

Logical NOT operator deals with boolean values either true or false.

Ex:

```

class Test
{
    public static void main(String[] args)
    {
        boolean b = !(5 > 2);

        System.out.println(b); // false
    }
}

```

```

class Test
{
    public static void main(String[] args)
    {
        boolean b = !(5 > 2);

        System.out.println(b); // false
    }
}py
class Test
{
    public static void main(String[] args)
    {
        boolean b = !(5 > 20);

        System.out.println(b); //true
    }
}

```

## How to convert decimal to binary number

10 - decimal number  
 1010 - binary number

2 | 10

```

    ---- 0
  2 | 5
    ---- 1
  2 | 2
    ---- 0      ^
    1          |
  -----
1010

```

## How to convert binary to decimal number

```

binary number : 1010
decimal number : 10

```

```

1010
  <---

```

```

0*1 + 1*2 + 0*4 + 1*8

```

```

0 + 2 + 0 + 8 = 10

```

## Bitwise Operators

### Bitwise AND (&) Operator

Truth table:

```

T T = T
T F = F
F T = F
F F = F

```

```

class Test
{
    public static void main(String[] args)
    {
        int a = 10, b = 15;
    }
}

```



```

        int c = a & b;

        System.out.println(c); //10
    }
}

```

```

class Test
{
    public static void main(String[] args)
    {
        int a = 2, b = 5;

        int c = a & b;

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

```

## Bitwise OR (|) Operator

Truth table:

```

T T = T
T F = T
F T = T
F F = F

```

```

class Test
{
    public static void main(String[] args)
    {
        int a = 5, b = 10;

        int c = a | b;

        System.out.println(c); // 15
    }
}

```

```
}  
}
```

## Bitwise XOR (^) Operator

Truth table:

```
T T = F  
T F = T  
F T = T  
F F = F
```

```
class Test  
{  
    public static void main(String[] args)  
    {  
        int a = 5, b = 15;  
  
        int c = a ^ b;  
  
        System.out.println(c); // 10  
    }  
}
```

## Bitwise NOT (~) Operator

```
class Test  
{  
    public static void main(String[] args)  
    {  
        int i = ~10;  
  
        System.out.println(i); // -11  
    }  
}
```

```
class Test  
{  
    public static void main(String[] args)  
    {  
        int i = ~34;
```

```
        System.out.println(i); // -35
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = ~(-10);

        System.out.println(i); // 9
    }
}
```

## Arithmetic Operators

```
class Test
{
    public static void main(String[] args)
    {
        int i = 5 + 6 % 3 + 8 / 2 + 8 % 30 + 7 / 10 + 4 * 2 + 10 - 20;

        System.out.println(i); // 15
    }
}
```

## Relational Operators

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(10 > 20); //false
        System.out.println(10 >= 15); //false
        System.out.println(10 < 20); //true
        System.out.println(10 <= 10); //true
    }
}
```

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println(10 == 20); //false
        System.out.println(10 == 10); //true
        System.out.println(10 != 20); //true
        System.out.println(10 != 10); //false
    }
}

```

## Shift Operators

### Right Shift Operator (>>)

```

class Test
{
    public static void main(String[] args)
    {
        int i = 10 >> 3; // 10 / 2*2*2

        System.out.println(i); // 1
    }
}

```

```

class Test
{
    public static void main(String[] args)
    {
        int i = 100 >> 4; // 100 / 2*2*2*2

        System.out.println(i); // 6
    }
}

```

### Left Shift Operator (<<)

```

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

```

```

        int i = 10 << 4; // 10 * 2*2*2*2

        System.out.println(i); // 160
    }
}

```

```

class Test
{
    public static void main(String[] args)
    {
        int i = 100 << 6; // 100 * 2*2*2*2*2*2

        System.out.println(i); // 6400
    }
}

```

Interview Question: Check if a given number is even or odd

```

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

        System.out.println("Enter the number :");
        int n = sc.nextInt();

        String str = ((n & 1) == 0) ? "It is even number" : "It is odd
number";
        System.out.println(str);
    }
}

```

## Unary Operators

### Increment/Decrement Operators (++/--)

We have two types of increment operators:

1. Pre-increment Example: `++i;`
2. Post-increment Example: `i++;`

We have two types of decrement operators:

1. Pre-decrement Example: `--i;`
2. Post-decrement Example: `i--;`

## POST Increment/Decrement

Rule 1: First Take

Rule 2: Then Change

Examples:

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        i++;
        System.out.println(i); // 11
    }
}
```

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

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

```
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        int j = i++;
        System.out.println(i + " " + j); // 11 10
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        int j = i++ + i++ + i++; // 10 + 11 + 12
        System.out.println(i + " " + j); // 13 33
    }
}
```

## Pre Increment/Decrement

Rule 1: First Change

Rule 2: Then Take

Examples:

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        ++i;
        System.out.println(i); // 11
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        System.out.println(++i); // 11
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        System.out.println(++i); // 11
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        System.out.println(++i); // 11
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        int j = ++i + --i + ++i; // 11 + 10 + 11
        System.out.println(i + " " + j); // 11 32
    }
}
```

```
class Test
{
```



```
public static void main(String[] args)
{
    int i = 10;
    int j = ++i + --i + ++i; // 11 + 10 + 11
    System.out.println(i + " " + j); // 11 32
}
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        System.out.println(i++ + ++i); // 10 + 12 = 22
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        100++;
        System.out.println(i); // Compile-Time Error
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int i = 10;
        System.out.println(++(i++)); // Compile-Time Error
    }
}
```

```
class Test
```

```
{
    public static void main(String[] args)
    {
        byte b = 127;
        b++;
        System.out.println(b); // -128
    }
}
```

# Control Statements

Control statements enable the programmer to control the flow of the program. They allow us to make decisions, jump from one section of code to another, and execute code repeatedly.

In Java, we have four types of control statements:

1. Decision Making Statement
2. Selection Statement
3. Iteration Statement
4. Jump Statement

## 1) Decision Making Statement

It is used to declare conditions in our program. It can be implemented using:

i. if statement ii. if-else statement iii. if-else-if ladder iv. nested if statement

### i) if statement

It is used to execute the source code only if our condition is true.

Syntax:

```
if (condition)
{
    // code to be executed
}
```

Example:

```
if (condition)
{
    // code to be executed
}
```

Output:

```
stmt1
stmt2
stmt3
```

Example:

```
java
Copy
class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");
        if (5 > 20)
        {
            System.out.println("stmt2");
        }
        System.out.println("stmt3");
    }
}
```

Output:

```
Copy
stmt1
stmt3
```

Example:

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

```

{
    if (5 > 20)
        System.out.println("stmt1");
        System.out.println("stmt2");
        System.out.println("stmt3");
}
}

```

Output:

Copy

stmt2

stmt3

Question: Write a Java program to find out the greatest of two numbers.

java

Copy

```

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

        System.out.println("Enter the first number:");
        int a = sc.nextInt();

        System.out.println("Enter the second number:");
        int b = sc.nextInt();

        if (a > b)
            System.out.println(a + " is greatest");

        if (b > a)
            System.out.println(b + " is greatest");
    }
}

```

Question: Write a Java program to find out the greatest of three numbers.

java

Copy

```

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

        System.out.println("Enter the first number:");
        int a = sc.nextInt();

        System.out.println("Enter the second number:");
        int b = sc.nextInt();

        System.out.println("Enter the third number:");
        int c = sc.nextInt();

        if ((a > b) && (a > c))
            System.out.println(a + " is greatest");

        if ((b > a) && (b > c))
            System.out.println(b + " is greatest");

        if ((c > a) && (c > b))
            System.out.println(c + " is greatest");
    }
}

```

## ii) if-else statement

It is used to execute the source code whether our condition is true or false.

Syntax:

```

java
Copy
if (condition)
{
    // if condition is true
}
else
{
    // if condition is false
}

```

Example:

java

Copy

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");
        if (true)
        {
            System.out.println("stmt2");
        }
        else
        {
            System.out.println("stmt3");
        }
        System.out.println("stmt4");
    }
}
```

Output:

Copy

stmt1

stmt2

stmt4

Example:

java

Copy

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");
        if (false)
        {
            System.out.println("stmt2");
        }
        else
        {

```

```
        System.out.println("stmt3");
    }
    System.out.println("stmt4");
}
}
```

Output:

Copy

stmt1

stmt3

stmt4

Question: Write a Java program to find out if a given age is eligible to vote or not.

java

Copy

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

        System.out.println("Enter the age:");
        int age = sc.nextInt();

        if (age >= 18)
            System.out.println("You are eligible to vote");
        else
            System.out.println("You are not eligible to vote");
    }
}
```

Question: Write a Java program to find out if a given number is even or odd.

java

Copy

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

```

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number:");
int n = sc.nextInt();

if (n % 2 == 0)
    System.out.println("It is an even number");
else
    System.out.println("It is an odd number");
}
}

```

Alternative approach:

```

java
Copy
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number:");
        int n = sc.nextInt();

        if ((n & 1) == 0)
            System.out.println("It is an even number");
        else
            System.out.println("It is an odd number");
    }
}

```

Question: Write a Java program to check if a given number is positive or negative.

```

java
Copy
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

```



```

        System.out.println("Enter the number:");
        int n = sc.nextInt();

        if (n == 0)
        {
            System.out.println("It is neither a positive nor a negative
number");
            System.exit(0);
        }

        if (n > 0)
            System.out.println("It is a positive number");
        else
            System.out.println("It is a negative number");
    }
}

```

Question: Write a Java program to find out if a given year is a leap year or not.

java

Copy

```

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

        System.out.println("Enter the year:");
        int year = sc.nextInt();

        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
            System.out.println("It is a leap year");
        else
            System.out.println("It is not a leap year");
    }
}

```

### iii) if-else-if ladder

It will execute the source code based on multiple conditions.

Syntax:

```
java
Copy
if (cond1)
{
    // code to be executed
}
else if (cond2)
{
    // code to be executed
}
else if (cond3)
{
    // code to be executed
}
else
{
    // code to be executed if all conditions are false
}
```

Example:

```
java
Copy
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the option:");
        int option = sc.nextInt();

        if (option == 100)
            System.out.println("It is a police number");
        else if (option == 103)
            System.out.println("It is an enquiry number");
        else if (option == 108)
            System.out.println("It is an emergency number");
        else
            System.out.println("Invalid option");
    }
}
```

```
}  
}
```

Question: Write a Java program to check if a given alphabet is an uppercase letter, lowercase letter, digit, or a special symbol.

java

Copy

```
import java.util.Scanner;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
  
        System.out.println("Enter the character:");  
        char ch = sc.next().charAt(0);  
  
        if (ch >= 'A' && ch <= 'Z')  
            System.out.println("It is an uppercase letter");  
        else if (ch >= 'a' && ch <= 'z')  
            System.out.println("It is a lowercase letter");  
        else if (ch >= '0' && ch <= '9')  
            System.out.println("It is a digit");  
        else  
            System.out.println("It is a special symbol");  
    }  
}
```

Question: Write a Java program to check if a given alphabet is a vowel or not.

java

Copy

```
import java.util.Scanner;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
  
        System.out.println("Enter the alphabet:");  
        char ch = sc.next().charAt(0);
```

```

        if (ch == 'a' || ch == 'A')
            System.out.println("It is a vowel");
        else if (ch == 'e' || ch == 'E')
            System.out.println("It is a vowel");
        else if (ch == 'i' || ch == 'I')
            System.out.println("It is a vowel");
        else if (ch == 'o' || ch == 'O')
            System.out.println("It is a vowel");
        else if (ch == 'u' || ch == 'U')
            System.out.println("It is a vowel");
        else
            System.out.println("It is not a vowel");
    }
}

```

Assignment: Write a Java program to accept six marks of a student, then find out the total, average, and grade.

- If average is greater than or equal to 70, then A grade.
- If average is greater than or equal to 50, then B grade.
- If average is greater than or equal to 35, then C grade.
- If average is less than 35, then failed.

Orchasp Interview Question: Write a Java program to calculate costs based on user input. The program should prompt users to enter the total weight of items (in kilograms) and the shipping destination (domestic or international). For domestic orders, the program should charge Rs.500 for weights up to 5 kg and Rs.100 per additional kg. For international orders, it should charge Rs.1000 for weights up to 5 kg, Rs.200 per additional kg, and a Rs.500 surcharge for weights exceeding 10 kg. Print the calculated shipping cost.

```

java
Copy
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the weight of Item:");
        int weight = sc.nextInt();

        System.out.println("Enter the shipping destination:");
    }
}

```

String

CopyRetry

Claude's response was limited as it hit the maximum length allowed at this time. Claude does not have the ability to run the code it generates yet.

MS

Rewrite Without missing single word You can also do some changes like format and Sentence And Word Correction Please write code in code block Dont miss examples, codes, notepoints, subtopics, questions keep that in mind

Edit

I apologize for the confusion. I'll continue rewriting the content from where I left off, ensuring no information is missed and making minor improvements to formatting and wording where appropriate.

java

Copy

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

        System.out.println("Enter the weight of Item:");
        int weight = sc.nextInt();

        System.out.println("Enter the shipping destination:");
        String destination = sc.next();

        if (destination.equals("domestic"))
        {
            if (weight <= 5)
                System.out.println(500);
            else
                System.out.println(500 + (weight - 5) * 100);
        }
        else
        {
            if (weight <= 5)
                System.out.println(1000);
        }
    }
}
```

```

        else if (weight > 5 && weight <= 10)
            System.out.println(1000 + (weight - 5) * 200);
        else
            System.out.println(1000 + (weight - 5) * 200 + 500);
    }
}

```

#### iv) nested if statement

A nested if statement is when an if statement contains another if statement.

Syntax:

```

java
Copy
if (condition)
{
    if (condition)
    {
        // code to be executed
    }
}

```

Example:

```

java
Copy
class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");

        if (5 > 2)
        {
            System.out.println("stmt2");

            if (true)
            {
                System.out.println("stmt3");
            }
        }
    }
}

```

```
        System.out.println("stmt4");
    }

    System.out.println("stmt5");
}
}
```

Output:

Copy

```
stmt1
stmt2
stmt3
stmt4
```

```
stmt5
```

Example:

java

Copy

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("stmt1");

        if (5 > 20)
        {
            System.out.println("stmt2");

            if (true)
            {
                System.out.println("stmt3");
            }

            System.out.println("stmt4");
        }

        System.out.println("stmt5");
    }
}
```

Output:

Copy

stmt1

stmt5

Example:

java

Copy

```
class Test
```

```
{
```

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

```
    {
```

```
        System.out.println("stmt1");
```

```
        if (5 > 2)
```

```
        {
```

```
            System.out.println("stmt2");
```

```
            if (false)
```

```
            {
```

```
                System.out.println("stmt3");
```

```
            }
```

```
            System.out.println("stmt4");
```

```
        }
```

```
        System.out.println("stmt5");
```

```
    }
```

```
}
```

Output:

Copy

stmt1

stmt2

stmt4

stmt5

Question: Write a Java program to find out if a given number is positive or negative using a nested if statement.



java

Copy

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

        System.out.println("Enter the number:");
        int n = sc.nextInt();

        if (n != 0)
        {
            if (n > 0)
            {
                System.out.println("It is a positive number");
                System.exit(0);
            }

            System.out.println("It is a negative number");
        }
    }
}
```

## 2) Selection Statement

### switch case

The switch case is used to execute source code based on multiple conditions. It is similar to the if-else-if ladder.

Syntax:

java

Copy

```
switch (condition)
{
    case value1:
        // code to be executed
        break;

    case value2:
```

```

        // code to be executed
        break;

    // ...

    default:
        // code to be executed if all cases are false
}

```

Example:

java

Copy

```

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

        System.out.println("Enter the option:");
        int option = sc.nextInt();

        switch (option)
        {
            case 100: System.out.println("It is a police number");
                     break;
            case 103: System.out.println("It is an enquiry number");
                     break;
            case 108: System.out.println("It is an emergency number");
                     break;
            default:  System.out.println("Invalid option");
        }
    }
}

```

Note: The declaration of the break statement is optional. If we don't define a break statement, then from where our condition is satisfied, all subsequent cases will be executed. This is called the fall-through state of switch case.

Example:

java

Copy

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

        System.out.println("Enter the option:");
        int option = sc.nextInt();

        switch (option)
        {
            case 100: System.out.println("It is a police number");
                      //break;
            case 103: System.out.println("It is an enquiry number");
                      //break;
            case 108: System.out.println("It is an emergency number");
                      //break;
            default:  System.out.println("Invalid option");
        }
    }
}
```

Note: The allowed data types for switch case are byte, short, int, char, and String.

Question: Write a Java program to check if a given alphabet is a vowel or consonant.

java

Copy

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

        System.out.println("Enter the alphabet:");
        char ch = sc.next().charAt(0);

        switch (ch)
        {
            case 'a': System.out.println("It is a vowel"); break;
        }
    }
}
```

```

        case 'e': System.out.println("It is a vowel"); break;
        case 'i': System.out.println("It is a vowel"); break;
        case 'o': System.out.println("It is a vowel"); break;
        case 'u': System.out.println("It is a vowel"); break;
        default : System.out.println("It is a consonant");
    }
}
}

```

### 3) Iteration statement

An iteration statement is used to execute code repeatedly. It is possible to use iteration statements through loops.

We have four types of loops:

i) do-while loop ii) while loop iii) for loop iv) for-each loop

#### i) do-while loop

It will execute the source code as long as our condition is true.

Syntax:

```

java
Copy
do
{
    // code to be executed
} while (condition);

```

Example:

```

java
Copy
class Test
{
    public static void main(String[] args)
    {
        int i = 1;
        do
        {
            System.out.print(i + " "); // infinite 1
        }
    }
}

```

```

    }
    while (i <= 10);
}
}

```

Example:

java

Copy

```

class Test
{
    public static void main(String[] args)
    {
        int i = 11;
        do
        {
            System.out.print(i + " "); // 11
        }
        while (i <= 10);
    }
}

```

Question: Write a Java program to display 10 natural numbers.

java

Copy

```

class Test
{
    public static void main(String[] args)
    {
        int i = 1;
        do
        {
            System.out.print(i + " "); // 1 2 3 4 5 6 7 8 9 10
            i++;
        }
        while (i <= 10);
    }
}

```

Question: Write a Java program to perform the sum of 10 natural numbers.

java

Copy

```
class Test
{
    public static void main(String[] args)
    {
        int i = 1, sum = 0;
        do
        {
            sum = sum + i;
            i++;
        }
        while (i <= 10);

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

Question: Write a Java program to find out the factorial of a given number.

java

Copy

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

        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int i = n, fact = 1;
        do
        {
            fact = fact * i;
            i--;
        }
        while (i >= 1);

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

Question: Write a Java program to display the multiplication table of a given number.

java

Copy

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

        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int i = 1;
        do
        {
            System.out.println(n + " * " + i + " = " + n * i);
            i++;
        }
        while (i <= 10);
    }
}
```

## ii) while loop

It will execute the source code as long as our condition is true.

Syntax:

java

Copy

```
while (condition)
{
    // code to be executed
}
```

Example:

java

Copy

```
class Test
{
```

```

public static void main(String[] args)
{
    int i = 1;

    while (i <= 10)
    {
        System.out.print(i + " "); // infinite 1
    }
}

```

Example:

java

Copy

```

class Test
{
    public static void main(String[] args)
    {
        int i = 11;

        while (i <= 10)
        {
            System.out.print(i + " "); // nothing / empty
        }
    }
}

```

Question: Write a Java program to display 10 natural numbers.

java

Copy

```

class Test
{
    public static void main(String[] args)
    {
        int i = 1;

        while (i <= 10)
        {
            System.out.print(i + " "); // 1 2 3 4 5 6 7 8 9 10
            i++;
        }
    }
}

```



```
}  
}
```

Question: Write a Java program to perform the sum of 10 natural numbers.

java

Copy

```
class Test  
{  
    public static void main(String[] args)  
    {  
        int i = 1, sum = 0;  
  
        while (i <= 10)  
        {  
            sum = sum + i;  
            i++;  
        }  
  
        System.out.println(sum);  
    }  
}
```

Question: Write a Java program to find out the factorial of a given number.

java

Copy

```
import java.util.Scanner;  
class Test  
{  
    public static void main(String[] args)  
    {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter the number:");  
        int n = sc.nextInt();  
  
        int i = n, fact = 1;  
  
        while (i >= 1)  
        {  
            fact = fact * i;  
            i--;  
        }  
    }  
}
```

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

Question: Write a Java program to display the multiplication table of a given number.

java

Copy

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int i = 1;
        while (i <= 10)
        {
            System.out.println(n + " * " + i + " = " + n * i);
            i++;
        }
    }
}
```

Question: Write a Java program to perform the sum of digits of a given number.

java

Copy

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int rem, sum = 0;
        while (n > 0)
        {
```

```

        rem = n % 10;
        sum = sum + rem;
        n = n / 10;
    }

    System.out.println(sum);
}

```

Question: Write a Java program to find out if a given number is Armstrong or not.

java

Copy

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int temp = n;

        int rem, sum = 0;
        while (n > 0)
        {
            rem = n % 10;
            sum = sum + rem * rem * rem;
            n = n / 10;
        }

        if (temp == sum)
            System.out.println("It is an Armstrong number");
        else
            System.out.println("It is not an Armstrong number");
    }
}

```

Question: Write a Java program to display the reverse of a given number.

java

Copy

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int rem, rev = 0;

        while (n > 0)
        {
            rem = n % 10;
            rev = rev * 10 + rem;
            n = n / 10;
        }

        System.out.println(rev);
    }
}

```

Question: Write a Java program to find out if a given number is palindrome or not.

java

Copy

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int temp = n;

        int rem, rev = 0;

        while (n > 0)
        {
            rem = n % 10;
            rev = rev * 10 + rem;

```

```

        n = n / 10;
    }

    if (temp == rev)
        System.out.println("It is a palindrome number");
    else
        System.out.println("It is not a palindrome number");
}
}

```

### iii) for loop

The for loop executes a block of code repeatedly as long as a specified condition is true.

#### Syntax:

java

Copy

```

for (initialization; condition; increment/decrement) {
    // code to be executed
}

```

#### Examples:

1. Basic for loop:

java

Copy

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " "); // Output: 1 2 3 4 5 6 7 8 9 10
        }
    }
}

```

2. Printing even numbers:

java

Copy

```

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

```

```

        for (int i = 1; i <= 10; i++) {
            if (i % 2 == 0) {
                System.out.print(i + " "); // Output: 2 4 6 8 10
            }
        }
    }
}

```

### 3. Counting odd numbers:

java

Copy

```

class Test {
    public static void main(String[] args) {
        int cnt = 0;
        for (int i = 1; i <= 10; i++) {
            if (i % 2 != 0) {
                cnt++;
            }
        }
        System.out.println(cnt); // Output: 5
    }
}

```

### 4. Infinite loop with break:

java

Copy

```

class Test {
    public static void main(String[] args) {
        for (int i = 10; i >= 1; i--) {
            System.out.print(i + " "); // Output: 10
            i++;
            break; // Added to prevent infinite loop
        }
    }
}

```

### 5. Infinite loop:

java

Copy

```

class Test {

```

```

    public static void main(String[] args) {
        for (;;) {
            System.out.print("Hello "); // Prints "Hello" indefinitely
        }
    }
}

```

#### Note:

- Use a for loop when the number of iterations is known.
- Use a while loop when the number of iterations is unknown.
- Use a do-while loop when the code must execute at least once and the number of iterations is unknown.

#### Practice Questions:

1. Write a Java program to check if a given number is prime or not:

java

Copy

```

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        boolean isPrime = true;
        for (int i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                isPrime = false;
                break;
            }
        }

        if (isPrime) {
            System.out.println("It is a prime number");
        } else {
            System.out.println("It is not a prime number");
        }
    }
}

```

2. Write a Java program to display prime numbers from 1 to 100:

java

Copy

```
class Test {
    public static void main(String[] args) {
        for (int n = 2; n <= 100; n++) {
            boolean isPrime = true;
            for (int i = 2; i <= n / 2; i++) {
                if (n % i == 0) {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime) {
                System.out.print(n + " ");
            }
        }
    }
}
```

3. Write a Java program to check if a given number is perfect or not:

java

Copy

```
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        int sum = 0;
        for (int i = 1; i < n; i++) {
            if (n % i == 0) {
                sum += i;
            }
        }

        if (n == sum) {
            System.out.println("It is a perfect number");
        } else {

```



```

        System.out.println("It is not a perfect number");
    }
}

```

4. Write a Java program to find the GCD (Greatest Common Divisor) of two numbers:

java

Copy

```

class Test {
    public static void main(String[] args) {
        int a = 12, b = 18, gcd = 1;

        for (int i = 1; i <= a && i <= b; i++) {
            if (a % i == 0 && b % i == 0) {
                gcd = i;
            }
        }
        System.out.println("GCD: " + gcd);
    }
}

```

5. Write a Java program to generate the Fibonacci series up to a given number:

java

Copy

```

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of terms:");
        int n = sc.nextInt();

        int a = 0, b = 1, c;

        System.out.print(a + " " + b + " ");

        for (int i = 2; i < n; i++) {
            c = a + b;
            System.out.print(c + " ");
            a = b;
            b = c;
        }
    }
}

```

```
}  
}  
}
```

## Loop Patterns

### 1. Basic Patterns:

Pattern:

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

```
class Test {
```

```

    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}

```

**Pattern:**

```

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                System.out.print(j + " ");
            }
            System.out.println();
        }
    }
}

```

**Pattern:**

```

* * * *
* * * *
* * * *
* * * *

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                System.out.print("* ");
            }
        }
    }
}

```

```

        }
        System.out.println();
    }
}

```

**Pattern:**

```

4 4 4 4
3 3 3 3
2 2 2 2
1 1 1 1

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 4; i >= 1; i--) {
            for (int j = 1; j <= 4; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}

```

**Pattern:**

```

* * * *
*     *
*     *
* * * *

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {

```

```

        if (i == 1 || i == 4 || j == 1 || j == 4) {
            System.out.print("* ");
        } else {
            System.out.print(" ");
        }
    }
    System.out.println();
}
}
}

```

**Pattern:**

```

* - - -
- * - -
- - * -
- - - *

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                if (i == j) {
                    System.out.print("* ");
                } else {
                    System.out.print("- ");
                }
            }
            System.out.println();
        }
    }
}

```

**Pattern:**

```

* - - - *
- * - * -
- - * - -

```

```
- * - * -  
* - - - *
```

```
class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= 5; j++) {  
                if (i == j || i + j == 6) {  
                    System.out.print("* ");  
                } else {  
                    System.out.print("- ");  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

**Pattern:**

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

```
class Test {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            for (int j = 1; j <= 5; j++) {  
                if (i == 3 || j == 3) {  
                    System.out.print("* ");  
                } else {  
                    System.out.print("  ");  
                }  
            }  
            System.out.println();  
        }  
    }  
}
```

```
}
```

## 2. Assignments:

Pattern:

```
A A A A
B B B B
C C C C
D D D D
```

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

Pattern:

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

```
class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2) {
                    System.out.print(0 + " ");
                } else {

```

```

        System.out.print(1 + " ");
    }
}
System.out.println();
}
}
}
}

```

### Pattern:

```

* * * *
# # # #
@ @ @ @
& & & &

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                if (i == 1) {
                    System.out.print("* ");
                } else if (i == 2) {
                    System.out.print("# ");
                } else if (i == 3) {
                    System.out.print("@ ");
                } else {
                    System.out.print("& ");
                }
            }
            System.out.println();
        }
    }
}

```

---

### 3. Left Side Loop Patterns:



**Pattern:**

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

```
class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}
```

**Pattern:**

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

```
class Test {
    public static void main(String[] args) {
        for (int i = 4; i >= 1; i--) {
            for (int j = 1; j <= i; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}
```

**Pattern:**

```
*
* *
```

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

```
class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println();
        }
    }
}
```

**Pattern:**

```
1
2 3
4 5 6
7 8 9 0
```

```
class Test {
    public static void main(String[] args) {
        int k = 1;
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= i; j++) {
                if (k > 9) {
                    k = 0;
                }
                System.out.print(k + " ");
                k++;
            }
            System.out.println();
        }
    }
}
```

**Pattern:**

```
2
4 6
```

8 10 12  
14 16 18 20

```
class Test {  
    public static void main(String[] args) {  
        int k = 2;  
        for (int i = 1; i <= 4; i++) {  
            for (int j = 1; j <= i; j++) {  
                System.out.print(k + " ");  
                k += 2;  
            }  
            System.out.println();  
        }  
    }  
}
```

**Pattern:**

1  
3 5  
7 9 11  
13 15 17 19

```
class Test {  
    public static void main(String[] args) {  
        int k = 1;  
        for (int i = 1; i <= 4; i++) {  
            for (int j = 1; j <= i; j++) {  
                System.out.print(k + " ");  
                k += 2;  
            }  
            System.out.println();  
        }  
    }  
}
```

**Pattern:**

2  
3 5  
7 11 13

17 19 23 29

```
class Test {
    public static void main(String[] args) {
        int n = 2;
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= i; j++) {
                while (true) {
                    boolean flag = true;
                    for (int k = 2; k <= n / 2; k++) {
                        if (n % k == 0) {
                            flag = false;
                            break;
                        }
                    }
                    if (flag) {
                        System.out.print(n + " ");
                        n++;
                        break;
                    } else {
                        n++;
                    }
                }
            }
            System.out.println();
        }
    }
}
```

**Pattern:**

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

```
class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            if (i % 2 != 0) {
                for (int j = 1; j <= i; j++) {
```

```

        System.out.print(j + " ");
    }
    } else {
        for (int j = i; j >= 1; j--) {
            System.out.print(j + " ");
        }
    }
    System.out.println();
}
}
}
}

```

#### 4. Right Side Elements Patterns:

Pattern:

```

    1
  2 2
3 3 3
4 4 4 4

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 4; j > i; j--) {
                System.out.print(" ");
            }
            for (int j = 1; j <= i; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}

```

Pattern:

```

4 4 4 4
 3 3 3
  2 2
   1

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 4; i >= 1; i--) {
            for (int j = 4; j > i; j--) {
                System.out.print(" ");
            }
            for (int j = 1; j <= i; j++) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
}

```

**Pattern:**

```

      *
     * *
    * * *
   * * * *
  * * * *
 * * *
* * *
 *

```

```

class Test
{
    public static void main(String[] args)
    {
        //rows
        for(int i=1;i<=4;i++)
        {
            //space
            for(int j=4;j>i;j--)
            {
                System.out.print(" ");
            }

            //right side elements
            for(int j=1;j<=i;j++)
            {

```

```

        System.out.print("* ");
    }
    //new line
    System.out.println();
}

//rows
for(int i=3;i>=1;i--)
{
    //space
    for(int j=4;j>i;j--)
    {
        System.out.print(" ");
    }

    //right side elements
    for(int j=1;j<=i;j++)
    {
        System.out.print("* ");
    }
    //new line
    System.out.println();
}
}
}

```

## 5. Pyramid Loop Patterns:

Pattern:

```

    1
  1 2 1
1 2 3 2 1
1 2 3 4 3 2 1

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            for (int j = 4; j > i; j--) {
                System.out.print(" ");
            }

```

```

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

```

**Pattern:**

```

1 2 3 4 3 2 1
 1 2 3 2 1
   1 2 1
    1

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 4; i >= 1; i--) {
            for (int j = 4; j > i; j--) {
                System.out.print(" ");
            }
            for (int j = 1; j <= i; j++) {
                System.out.print(j + " ");
            }
            for (int j = i - 1; j >= 1; j--) {
                System.out.print(j + " ");
            }
            System.out.println();
        }
    }
}

```

**Pattern:**

```

    *
  * * *
* * * * *
* * * * * * *

```



```

class Test
{
    public static void main(String[] args)
    {
        //rows
        for(int i=1;i<=4;i++)
        {
            //space
            for(int j=4;j>i;j--)
            {
                System.out.print(" ");
            }

            //left side elements
            for(int j=1;j<=i;j++)
            {
                System.out.print("* ");
            }

            //right side elements
            for(int j=i-1;j>=1;j--)
            {
                System.out.print("* ");
            }
            //new line
            System.out.println();
        }
    }
}

```

**Pattern:**

```

      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *

```

```
class Test
{
    public static void main(String[] args)
    {
        // Upper half of the pattern
        for(int i=1; i<=4; i++)
        {
            // Space before stars
            for(int j=4; j>i; j--)
            {
                System.out.print(" ");
            }

            // Left side elements
            for(int j=1; j<=i; j++)
            {
                System.out.print("* ");
            }

            // Right side elements
            for(int j=i-1; j>=1; j--)
            {
                System.out.print("* ");
            }

            // Move to the next line
            System.out.println();
        }

        // Lower half of the pattern
        for(int i=3; i>=1; i--)
        {
            // Space before stars
            for(int j=3; j>=i; j--)
            {
                System.out.print(" ");
            }

            // Left side elements
            for(int j=1; j<=i; j++)
            {
```

```

        System.out.print("* ");
    }

    // Right side elements
    for(int j=i-1; j>=1; j--)
    {
        System.out.print("* ");
    }

    // Move to the next line
    System.out.println();
}
}
}

```

## 6. Interview Questions:

### Pascal Triangle:

```

    1
  1 1
 1 2 1
1 3 3 1
1 4 6 4 1

```

```

class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            for (int j = 4; j > i; j--) {
                System.out.print(" ");
            }
            int number = 1;
            for (int j = 0; j <= i; j++) {
                System.out.print(number + " ");
                number = number * (i - j) / (j + 1);
            }
            System.out.println();
        }
    }
}

```

### Triangle of Stars:

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

```
class Test
{
    public static void main(String[] args)
    {
        //rows
        for(int i=0;i<5;i++)
        {
            //space
            for(int j=4;j>i;j--)
            {
                System.out.print(" ");
            }
            int number=1;
            for(int j=0;j<=i;j++){
                System.out.print("* ");
                number=number*(i-j)/(j+1);
            }
            //new line
            System.out.println();
        }
    }
}
```

### Combined Triangle:

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

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

```

int rows=4;

//rows
for(int i=1;i<=rows;i++)
{
    //left side elements
    for(int j=1;j<=i;j++)
    {
        System.out.print(j+" ");
    }

    //space
    for(int j=1;j<=(rows-i)*2;j++)
    {
        System.out.print(" ");
    }

    //right side elements
    for(int j=i;j>=1;j--)
    {
        System.out.print(j+" ");
    }

    //new line
    System.out.println();
}
}
}

```

---

## 7. Additional Assignment:

Pattern:

```

4 4 4 4 4 4 4
4 3 3 3 3 3 4
4 3 2 2 2 3 4
4 3 2 1 2 3 4

```

```

4 3 2 2 2 3 4
4 3 3 3 3 3 4
4 4 4 4 4 4 4

```

```

class Test
{
    public static void main(String[] args)
    {
        int n=4;
        int size=7;

        for(int i=0;i<size;i++)
        {
            for(int j=0;j<size;j++)
            {
                int value= n -
Math.min(Math.min(i,j),Math.min(size-i-1,size-j-1));
                System.out.print(value+" ");
            }
            //new line
            System.out.println();
        }
    }
}

```

**Pattern:**

```

  *
 * *
*   *
*   *
*   *
 * *
  *

```

```

public class Test {
    public static void main(String[] args) {
        int n = 4; // height of the pattern

        // Upper half of the pattern
        for (int i = 0; i < n; i++) {
            // Print leading spaces

```

```

        for (int j = 0; j < n - i - 1; j++) {
            System.out.print(" ");
        }
        // Print first star
        System.out.print("*");

        // Print space between stars
        if (i > 0) {
            for (int j = 0; j < 2 * i - 1; j++) {
                System.out.print(" ");
            }
            // Print second star
            System.out.print("*");
        }
        System.out.println();
    }

    // Lower half of the pattern
    for (int i = n - 2; i >= 0; i--) {
        // Print leading spaces
        for (int j = 0; j < n - i - 1; j++) {
            System.out.print(" ");
        }
        // Print first star
        System.out.print("*");

        // Print space between stars
        if (i > 0) {
            for (int j = 0; j < 2 * i - 1; j++) {
                System.out.print(" ");
            }
            // Print second star
            System.out.print("*");
        }
        System.out.println();
    }
}

```

---

Regards,

## 4) Jump Statements

Jump statements are used to transfer control from one part of a program to another. There are two types of jump statements in Java:

1. break
2. continue

### i) break

The **break** statement is used to terminate the execution of loops and switch cases. It can be used with conditional statements like **if**.

Examples:

java

Copy

```
class Test {  
    public static void main(String[] args) {  
        System.out.println("stmt1");  
  
        break; // Compile-Time Error: break outside switch or loop  
  
        System.out.println("stmt2");  
    }  
}
```

java

Copy

```
class Test {  
    public static void main(String[] args) {  
        System.out.println("stmt1");  
  
        if (true) {  
            break; // Compile-Time Error: break outside switch or loop  
        }  
  
        System.out.println("stmt2");  
    }  
}
```



java

Copy

```
class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break;
            }
            System.out.print(i + " "); // Output: 1 2 3 4
        }
    }
}
```

## ii) continue

The `continue` statement is used to skip the rest of the current iteration in a loop and continue with the next iteration.

Examples:

java

Copy

```
class Test {
    public static void main(String[] args) {
        System.out.println("stmt1");

        continue; // Compile-Time Error: continue outside of loop

        System.out.println("stmt2");
    }
}
```

java

Copy

```
class Test {
    public static void main(String[] args) {
        System.out.println("stmt1");

        if (true) {
            continue; // Compile-Time Error: continue outside of loop
        }
    }
}
```

```

        System.out.println("stmt2");
    }
}

```

java

Copy

```

class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                continue;
            }
            System.out.print(i + " "); // Output: 1 2 3 4 6 7 8 9 10
        }
    }
}

```

## Various Ways to Declare Methods in Java

There are four ways to declare methods in Java:

1. No return type with no argument method
2. No return type with argument method
3. With return type with no argument method
4. With return type with argument method

### 1) No return type with no argument method

If we don't have arguments, we need to ask for input values inside the callee method.

Example:

java

Copy

```

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        sum(); // caller method
        sum();
    }

    // callee method
}

```

```

    public static void sum() {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the first number:");
        int a = sc.nextInt();
        System.out.println("Enter the second number:");
        int b = sc.nextInt();

        int c = a + b;

        System.out.println(c);
    }
}

```

## 2) No return type with argument method

If we have arguments, we need to ask for input values inside the main method.

Example:

java

Copy

```

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the first number:");
        int a = sc.nextInt();
        System.out.println("Enter the second number:");
        int b = sc.nextInt();

        sum(a, b); // caller method
    }

    // callee method
    public static void sum(int a, int b) {
        int c = a + b;
        System.out.println(c);
    }
}

```

### 3) With return type with no argument method

The return type depends on the output data type.

Example:

java

Copy

```
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        int k = sum(); // caller method
        System.out.println(k);
    }

    // callee method
    public static int sum() {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the first number:");
        int a = sc.nextInt();
        System.out.println("Enter the second number:");
        int b = sc.nextInt();

        int c = a + b;

        return c;
    }
}
```

### 4) With return type with argument method

Example:

java

Copy

```
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the first number:");
```

```

        int a = sc.nextInt();
        System.out.println("Enter the second number:");
        int b = sc.nextInt();

        System.out.println(sum(a, b)); // caller method
    }

    // callee method
    public static int sum(int a, int b) {
        int c = a + b;
        return c;
    }
}

```

Q) Write a java program to find out given number is even or odd using with return type with argument method?

approach1

```

import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the number :");
        int n=sc.nextInt();

        //caller method
        System.out.println(find(n));
    }
    //callie method
    public static String find(int n)
    {
        if(n%2==0)
            return "It is even number";
        else
            return "It is odd number";
    }
}

```

approach2

```
import java.util.Scanner;
class Test
{
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the number :");
        int n=sc.nextInt();

        //caller method
        int k=find(n);
        if(k==1)
            System.out.println("It is even number");
        else
            System.out.println("It is odd number");
    }
    //callie method
    public static int find(int n)
    {
        if(n%2==0)
            return 1;
        else
            return 0;
    }
}
```

approach3

-----

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

```

        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the number :");
        int n=sc.nextInt();

        //caller method
        boolean k=find(n);
        if(k)
            System.out.println("It is even number");
        else
            System.out.println("It is odd number");
    }
    //callie method
    public static boolean find(int n)
    {
        if(n%2==0)
            return true;
        else
            return false;
    }
}

```

## Recursion

Recursion is a programming technique where a method calls itself repeatedly. It's similar to looping, but when using recursion, we typically avoid using explicit loops.

### Display 10 Natural Numbers

Question: Write a Java program to display 10 natural numbers without using loops.

```

class Test {
    public static void main(String[] args) {
        // Caller method
        display(1);
    }
}

```

```
// Callee method
public static void display(int i) {
    if (i <= 10) {
        System.out.print(i + " "); // Output: 1 2 3 4 5 6 7 8 9 10
        display(i + 1);
    }
}
}
```

## Sum of Two Numbers Without Arithmetic Operators

Question: Write a Java program to perform the sum of two numbers without using arithmetic operators.

```
import java.util.Scanner;
class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the first number:");
        int a = sc.nextInt(); // e.g., 5

        System.out.println("Enter the second number:");
        int b = sc.nextInt(); // e.g., 10

        // Caller method
        System.out.println(calculate(a, b));
    }

    // Callee method
    public static int calculate(int a, int b) {
        if (a == 0)
            return b;

        return calculate(--a, ++b);
    }
}
```

## Factorial Using Recursion

Question: Write a Java program to find the factorial of a given number using recursion.



```

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt(); // e.g., 5

        // Caller method
        System.out.println(factorial(n));
    }

    // Callee method
    public static int factorial(int n) {
        if (n < 0)
            return -1;
        if (n == 0)
            return 1;

        return n * factorial(n - 1);
    }
}

```

## Nth Element of Fibonacci Series

Question: Write a Java program to find the Nth element of the Fibonacci series.

Fibonacci sequence: 0 1 1 2 3 5 8 ...

```

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int n = sc.nextInt();

        // Caller method
        System.out.println(fib(n));
    }

    // Callee method

```

```
public static int fib(int n) {  
    if (n == 0 || n == 1)  
        return 0;  
    if (n == 2)  
        return 1;  
  
    return fib(n - 1) + fib(n - 2);  
}  
}
```

# Arrays

An array is a collection of homogeneous data elements.

## Advantages of Arrays:

1. We can represent multiple elements using a single variable name.

Example: `int[] arr = {10, 20, 30};`

2. Arrays are recommended for performance reasons.

## Disadvantages of Arrays:

1. Arrays have a fixed size. Once created, we cannot increase or decrease the size of an array.
2. To use arrays effectively, we should know the size in advance, which is not always possible.

## Types of Arrays in Java:

1. Single Dimensional Array
2. Double Dimensional Array
3. Multi Dimensional Array

## Array Declaration

When declaring an array, we should not specify the array size.

## Single Dimensional Array:

- `int[] arr;`
- `int []arr;`
- `int arr[];`

## Double Dimensional Array:

- `int[][] arr;`
- `int [][]arr;`
- `int arr[][];`
- `int[] []arr;`
- `int[] arr[];`
- `int []arr[];`

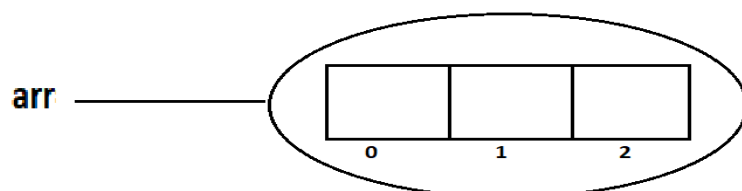
## Multi Dimensional Array:

- `int[][][] arr;`
- `int [][][]arr;`
- `int arr[][][];`
- `int[][] []arr;`
- `int[][] arr[];`
- `int[] [][]arr;`
- `int[] arr[][];`
- `int[] []arr[];`
- `int [][]arr[];`
- `int []arr[][];`

## Array Creation

In Java, every array is considered an object. Hence, we use the `new` operator to create an array.

```
int[] arr=new int[3];
```



## Rules for Constructing an Array:

When creating an array, we must specify the array size.

Example:

```
int[] arr = new int[3]; (valid)
```

```
int[] arr = new int[]; (invalid - Compile-Time Error: Array dimension missing)
```

It is legal to have an array size of zero.

Example:

```
int[] arr = new int[0];  
System.out.println(arr.length); // Output: 0
```

We cannot use a negative number as an array size; it will result in a `NegativeArraySizeException`.

Example:

```
int[] arr = new int[3]; // Valid  
int[] arr = new int[-3]; // Runtime Error: NegativeArraySizeException
```

The allowed data types for array size are byte, short, int, and char.

Example:

```
byte b = 10;  
int[] arr = new int[b]; // Valid  
int[] arr = new int['a']; // Valid  
int[] arr = new int[10.5d]; // Invalid
```

The maximum length we can use for an array size is the maximum value of int (2,147,483,647).

Example: `int[] arr = new int[2147483647];`

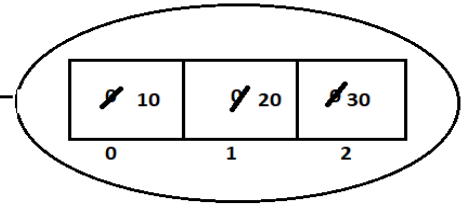
## Array Initialization

When we create an array, every array element is initialized with default values. If we're not satisfied with the default values, we can change them to custom values.

```
int[] arr=new int[3];
```

```
arr[0]=10;  
arr[1]=20;  
arr[2]=30;  
arr[3]=40; // R.E ArrayIndexOutOfBoundsException
```

arr



## Array Declaration, Creation, and Initialization in a Single Line

Instead of:

```
int[] arr;  
arr = new int[3];  
arr[0] = 10;  
arr[1] = 20;  
arr[2] = 30;
```

We can use:

```
int[] arr = {10, 20, 30};  
char[] carr = {'a', 'b', 'c'};  
String[] sarr = {"hi", "hello", "bye"};
```

## Single Dimensional Array Programs

### 1: Display Array Elements

Question: Write a Java program to display array elements.

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

```

Scanner sc = new Scanner(System.in);
System.out.println("Enter the array size:");
int size = sc.nextInt(); // e.g., 4

int[] arr = new int[size];

// Insert elements
for (int i = 0; i < arr.length; i++) {
    System.out.println("Enter the element:");
    arr[i] = sc.nextInt();
}

// Display elements
for (int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + " ");
}
}
}

```

## 2: Display Array Elements (For-each Loop)

Question: Write a Java program to display array elements.

Input: 3 7 1 2 9 5 Output:

Copy

```

3
7
1
2
9
5

```

java

Copy

```

class Test {
    public static void main(String[] args) {
        int[] arr = {3, 7, 1, 2, 9, 5};

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

```

```
}  
}
```

### 3: Sum of Array Elements

Question: Write a Java program to perform the sum of array elements.

Input: 3 7 1 2 9 5 Output: 27

java

Copy

```
class Test {  
    public static void main(String[] args) {  
        int[] arr = {3, 7, 1, 2, 9, 5};  
  
        int sum = 0;  
  
        for (int i : arr) {  
            sum += i;  
        }  
  
        System.out.println(sum);  
    }  
}
```

```
}
```

### 4: Display Array Elements in Reverse Order

Question: Write a Java program to display array elements in reverse order.

Input: 3 7 1 2 9 5 Output: 5 9 2 1 7 3

java

Copy

```
class Test {  
    public static void main(String[] args) {  
        int[] arr = {3, 7, 1, 2, 9, 5};  
  
        for (int i = arr.length - 1; i >= 0; i--) {  
            System.out.print(arr[i] + " ");  
        }  
    }  
}
```

```
}
```

## 5: Display Even Elements from Array

Question: Write a Java program to display even elements from a given array.

Input: 3 7 1 2 9 8 Output: 2 8

java

Copy

```
class Test {  
    public static void main(String[] args) {  
        int[] arr = {3, 7, 1, 2, 9, 8};  
  
        for (int i : arr) {  
            if (i % 2 == 0) {  
                System.out.print(i + " ");  
            }  
        }  
    }  
}
```

```
}
```

## 6: Count Odd Elements from Array

Question: Write a Java program to count odd elements from a given array.

Input: 3 7 1 2 9 8 Output: 4

java

Copy

```
class Test {  
    public static void main(String[] args) {  
        int[] arr = {3, 7, 1, 2, 9, 8};  
  
        int cnt = 0;  
  
        for (int i : arr) {  
            if (i % 2 != 0) {  
                cnt++;  
            }  
        }  
    }  
}
```



```

        System.out.println(cnt);
    }
}

```

## 7: Display Prime Elements from Array

Question: Write a Java program to display a list of prime elements from a given array.

Input : 9 2 10 5 8 7

Output: 2 5 7

```

class Test {
    public static void main(String[] args) {
        int[] arr = {9, 2, 10, 5, 8, 7};

        for (int n : arr) {
            boolean flag = true;
            for (int i = 2; i <= n / 2; i++) {
                if (n % i == 0) {
                    flag = false;
                    break;
                }
            }
            if (flag) {
                System.out.print(n + " ");
            }
        }
    }
}

```

## 8: Sort Array Elements (Using Arrays.sort())

Question: Write a Java program to display array elements in sorting order.

Input: 6 8 1 3 2 5 Output: 1 2 3 5 6 8

java  
Copy

```

import java.util.Arrays;

```

```

class Test {
    public static void main(String[] args) {
        int[] arr = {6, 8, 1, 3, 2, 5};

        Arrays.sort(arr);

        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}

```

## 9: Sort Array Elements (Without Using sort() Method)

Write a Java program to display array elements in sorting order without using the sort() method.

Input : 6 8 1 3 2 5

Output : 1 2 3 5 6 8

```

class Test {
    public static void main(String[] args) {
        int[] arr = {6, 8, 1, 3, 2, 5};

        // Sorting logic
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length; j++) {
                if (arr[i] < arr[j]) {
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }

        // Print the elements
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}

```

## 10: Sort Array Elements in Descending Order

Question: Write a Java program to display array elements in descending order without using the sort() method.

Input : 6 8 1 3 2 5

Output : 8 6 5 3 2 1

```
class Test {
    public static void main(String[] args) {
        int[] arr = {6, 8, 1, 3, 2, 5};

        // Descending logic
        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length; j++) {
                if (arr[i] > arr[j]) {
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }

        // Print the elements
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

## 11: Find Highest Element in Array

Question: Write a Java program to display the highest element from a given array.

Input : 6 8 1 3 2 5

Output : 8

```
class Test {
```

```

public static void main(String[] args) {
    int[] arr = {6, 8, 1, 3, 2, 5};

    int big = arr[0];

    for (int i : arr) {
        if (i > big) {
            big = i;
        }
    }

    System.out.println(big);
}

```

## 12: Find Lowest Element in Array

Question: Write a Java program to display the lowest element from a given array.

Input : 6 8 1 3 2 5

Output : 1

```

class Test {
    public static void main(String[] args) {
        int[] arr = {6, 8, 1, 3, 2, 5};

        int small = arr[0];

        for (int i : arr) {
            if (i < small) {
                small = i;
            }
        }

        System.out.println(small);
    }
}

```

## 13: Find Three Highest Elements in Array

Question: Write a Java program to display the three highest elements from a given array.

Input : 5 8 1 2 9 6 7 4

Output : 9 8 7

```
class Test {
    public static void main(String[] args) {
        int[] arr = {5, 8, 1, 2, 9, 6, 7, 4};

        int firstElement = Integer.MIN_VALUE;
        int secondElement = Integer.MIN_VALUE;
        int thirdElement = Integer.MIN_VALUE;

        for (int i : arr) {
            if (i > firstElement) {
                thirdElement = secondElement;
                secondElement = firstElement;
                firstElement = i;
            } else if (i > secondElement) {
                thirdElement = secondElement;
                secondElement = i;
            } else if (i > thirdElement) {
                thirdElement = i;
            }
        }

        System.out.println(firstElement + " " + secondElement + " " +
            thirdElement);
    }
}
```

## 14: Write a java program to display duplicate elements from given array?

Input : 6 8 2 3 1 5 4 2 6 8

Output: 6 8 2

```
class Test
{
```

```

public static void main(String[] args)
{
    int[] arr={6,8,2,3,1,5,4,2,6,8};

    for(int i=0;i<arr.length;i++)
    {
        for(int j=i+1;j<arr.length;j++)
        {
            if(arr[i]==arr[j])
            {
                System.out.print(arr[i]+" ");
            }
        }
    }
}

```

## Array Manipulation Problems

### 1. Find Unique Elements in an Array

Question: Write a Java program to display unique elements from a given array.

Input : 5 8 1 3 7 9 5 1 7

Output : 8 3 9

```

class Test {
    public static void main(String[] args) {
        int[] arr = {5, 8, 1, 3, 7, 9, 5, 1, 7};

        for (int i = 0; i < arr.length; i++) {
            int count = 0;

            for (int j = 0; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    count++;
                }
            }

            if (count == 1)

```

```

        System.out.print(arr[i] + " ");
    }
}

```

## 2. Find Most Repeating Element in an Array

Question: Write a Java program to find out the most repeating element in a given array.

Input : 7 3 2 7 3 5 7 3 7

Output : 7 repeating for 4 times

```

class Test {
    public static void main(String[] args) {
        int[] arr = {7, 3, 2, 7, 3, 5, 7, 3, 7};

        int maxCount = 0;
        int element = 0;

        for (int i = 0; i < arr.length; i++) {
            int count = 0;
            for (int j = 0; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    count++;
                }
            }
            if (maxCount < count) {
                maxCount = count;
                element = arr[i];
            }
        }

        System.out.println(element + " repeating for " + maxCount + "
times");
    }
}

```

## 3. Find Pairs with Given Sum

Question: Write a Java program to display pairs of elements equal to a given sum.

Input:

- Array: 2 4 7 1 9 6 3
- Sum: 10

Output: 4 6 7 3 1 9

```
class Test {
    public static void main(String[] args) {
        int[] arr = {2, 4, 7, 1, 9, 6, 3};
        int sum = 10;

        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] + arr[j] == sum) {
                    System.out.println(arr[i] + " " + arr[j]);
                }
            }
        }
    }
}
```

#### 4. Find Triplets with Given Sum

Question: Write a Java program to display triplets of elements equal to a given sum.

Input :

- Array: 2 4 7 1 9 6 3
- Sum: 10

Output : 2 7 1 1 6 3

```
class Test {
    public static void main(String[] args) {
        int[] arr = {2, 4, 7, 1, 9, 6, 3};
        int sum = 10;

        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                for (int k = j + 1; k < arr.length; k++) {
                    if (arr[i] + arr[j] + arr[k] == sum) {
                        System.out.println(arr[i] + " " + arr[j] + " " +
arr[k]);
                    }
                }
            }
        }
    }
}
```



```

    }
  }
}

```

## 5. Segregate Array Elements

Question: Write a Java program to segregate array elements (0s and 1s).

Input : 1 0 0 1 1 0 1 0 0 1

Output : 0 0 0 0 0 1 1 1 1 1

```

class Test {
    public static void main(String[] args) {
        int[] arr = {1, 0, 0, 1, 1, 0, 1, 0, 0, 1};
        int[] newArr = new int[arr.length];

        int j = 0;
        for (int i : arr) {
            if (i == 0) {
                newArr[j++] = i;
            }
        }

        while (j < arr.length) {
            newArr[j++] = 1;
        }

        for (int i : newArr) {
            System.out.print(i + " ");
        }
    }
}

```

## 6. Find Leader Elements in an Array

Question: Write a Java program to find out leader elements from a given array.

Input : 4 9 32 6 12 1 8

Output : 8 12 32

```
class Test {
    public static void main(String[] args) {
        int[] arr = {4, 9, 32, 6, 12, 1, 8};

        int max = arr[arr.length - 1];
        System.out.print(max + " ");

        for (int i = arr.length - 2; i >= 0; i--) {
            if (arr[i] > max) {
                max = arr[i];
                System.out.print(max + " ");
            }
        }
    }
}
```

## 7. Find Missing Element in an Array

Question: Write a Java program to find out the missing element from a given array.

Input : 7 1 3 5 2 6

Output : 4

```
class Test {
    public static void main(String[] args) {
        int[] arr = {7, 1, 3, 5, 2, 6};

        int sum_of_arr = arr.length + 1;
        int sum = (sum_of_arr * (sum_of_arr + 1)) / 2;

        for (int i : arr) {
            sum = sum - i;
        }

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

## 8. Merge and Sort Two Arrays

Question: Write a Java program to merge two arrays and display them in sorting order.

Input:

- Array 1: 6 9 7 10 8
- Array 2: 5 1 3 4 2

Output: 1 2 3 4 5 6 7 8 9 10

```
import java.util.Arrays;

class Test {
    public static void main(String[] args) {
        int[] arr1 = {6, 9, 7, 10, 8};
        int[] arr2 = {5, 1, 3, 4, 2};

        int size1 = arr1.length;
        int size2 = arr2.length;

        arr1 = Arrays.copyOf(arr1, size1 + size2);

        int j = 0;
        for (int i = size1; i < arr1.length; i++) {
            arr1[i] = arr2[j++];
        }

        Arrays.sort(arr1);

        for (int i : arr1) {
            System.out.print(i + " ");
        }
    }
}
```

## 9. Insert Element at Given Index

Question: Write a Java program to insert an element at a given index in an array.

Input:

- Array: 5 8 2 4 3 9
- Index: 3

- Element: 10

Output: 5 8 2 10 4 3 9

```
import java.util.Arrays;

class Test {
    public static void main(String[] args) {
        int[] arr = {5, 8, 2, 4, 3, 9};
        int index = 3;
        int element = 10;

        arr = Arrays.copyOf(arr, arr.length + 1);

        for (int i = arr.length - 1; i > index; i--) {
            arr[i] = arr[i - 1];
        }

        arr[index] = element;

        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

## 10. Delete First Occurrence of an Element

Question: Write a Java program to delete the first occurrence of a given element in an array.

Input:

- Array: 5 2 9 4 7 2 1 6 2
- Element to delete: 2

Output: 5 9 4 7 2 1 6 2

java  
Copy

```
class Test {
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 4, 7, 2, 1, 6, 2};
        int element = 2;
```

```

int[] newArr = new int[arr.length - 1];

int j = 0, count = 0;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] == element && count == 0) {
        count = 1;
        continue;
    }
    newArr[j++] = arr[i];
}

for (int i : newArr) {
    System.out.print(i + " ");
}
}

```

## 11. Find Elements Greater than Neighbors

Question: Write a Java program to identify and print all elements in an array that are greater than both their immediate predecessors and successors, considering the first and last elements as having only one neighbor.

Input: 1 3 20 4 75 0 90

Output: 20 75 90

```

class Test {
    public static void main(String[] args) {
        int[] arr = {1, 3, 20, 4, 75, 0, 90};

        if (arr[0] > arr[1]) {
            System.out.print(arr[0] + " ");
        }

        for (int i = 1; i < arr.length - 1; i++) {
            if (arr[i] > arr[i-1] && arr[i] > arr[i+1]) {
                System.out.print(arr[i] + " ");
            }
        }

        if (arr[arr.length - 1] > arr[arr.length - 2]) {

```

```

        System.out.print(arr[arr.length - 1] + " ");
    }
}

```

## 12. Minimum Coins Problem

Question: Write a Java program to determine the smallest number of coins needed to total 86 rupees. Use the denominations provided in the array {1, 2, 5, 10}.

Output: 1 coin(s) of 1 rupee(s) 1 coin(s) of 5 rupee(s) 8 coin(s) of 10 rupee(s)

```

class Test {
    public static void main(String[] args) {
        int[] denominations = {1, 2, 5, 10};
        int amount = 86;

        int[] result = minimumCoins(denominations, amount);

        for (int i = 0; i < result.length; i++) {
            if (result[i] > 0) {
                System.out.println(result[i] + " coin(s) of " +
denominations[i] + " rupee(s)");
            }
        }
    }

    public static int[] minimumCoins(int[] denominations, int amount) {
        int[] coinsCount = new int[denominations.length];

        for (int i = denominations.length - 1; i >= 0; i--) {
            coinsCount[i] = amount / denominations[i];
            amount %= denominations[i];
        }

        return coinsCount;
    }
}

```

## 13. Segregate Array Elements (Alternative Version)

Question: Write a Java program to segregate array elements (0s and 1s).

Input: 1 0 0 1 1 0 1 1 0 0 Output: 0 0 0 0 0 1 1 1 1 1

```
class Test
{
    public static void main(String[] args)
    {
        int[] arr={1,0,0,1,1,0,1,1,0,0};

        int start=0;
        int end=arr.length-1;

        while(start<end)
        {
            if(arr[start]==0)
            {
                start++;
            }
            else if(arr[end]==1)
            {
                end--;
            }
            else
            {
                int temp=arr[start];
                arr[start]=arr[end];
                arr[end]=temp;
            }
        }

        //display
        for(int i:arr)
        {
            System.out.print(i+" ");
        }
    }
}
```

## Double Dimensional Array

A double dimensional array is a combination of rows and columns. It's implemented based on an array of arrays approach, not in matrix form. The main objective is efficient memory utilization. Double dimensional arrays are used in developing business-oriented applications, gaming applications, matrix-type applications, and more.

## Declaration Syntax

```
datatype[][] variable_name = new datatype[rows][cols];
```

Example:

```
int[][] arr = new int[3][3];
```

This can store a total of 9 elements.

## Display Array Elements in Matrix Form

```
import java.util.Scanner;

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

        System.out.println("Enter the rows:");
        int rows = sc.nextInt();

        System.out.println("Enter the cols:");
        int cols = sc.nextInt();

        int[][] arr = new int[rows][cols];

        // Insert the elements
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.println("Enter the element:");
                arr[i][j] = sc.nextInt();
            }
        }
    }
}
```



```

        // Display the elements
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println(); // New line
        }
    }
}

```

## Sum of Diagonal Elements

Input:

```

1 2 3
4 5 6
7 8 9

```

```

class Test {
    public static void main(String[] args) {
        int[][] arr = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int rows = arr.length;
        int cols = arr[0].length;

        int sum = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (i == j) {
                    sum += arr[i][j];
                }
            }
        }

        System.out.println(sum);
    }
}

```

```
}
```

## Sum of Upper Triangle Elements

Input:

```
1 2 3
4 5 6
7 8 9
```

java

Copy

```
class Test {
    public static void main(String[] args) {
        int[][] arr = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int rows = arr.length;
        int cols = arr[0].length;

        int sum = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (i < j) {
                    sum += arr[i][j];
                }
            }
        }

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

## Sum of Lower Triangle Elements

Input:

```
1 2 3
4 5 6
7 8 9
```

```
class Test {
    public static void main(String[] args) {
        int[][] arr = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int rows = arr.length;
        int cols = arr[0].length;

        int sum = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (i > j) {
                    sum += arr[i][j];
                }
            }
        }

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

## Display Square of a Matrix

```
Input:
1 2 3
4 5 6
7 8 9
```

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

```

int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

int rows = arr.length;
int cols = arr[0].length;

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        System.out.print(arr[i][j] * arr[i][j] + " ");
    }
    System.out.println(); // New line
}
}
}

```

## Display Array Elements in Spiral Form

Input:

```

1 2 3
4 5 6
7 8 9

```

Output:

```

1 2 3 6 9 8 7 4 5

```

```

class Test {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int top = 0;
        int bottom = matrix.length - 1;
        int left = 0;
        int right = matrix[0].length - 1;
    }
}

```

```

while (true) {
    if (left > right) {
        break;
    }
    for (int i = left; i <= right; i++) {
        System.out.print(matrix[top][i] + " ");
    }
    top++;

    if (top > bottom) {
        break;
    }
    for (int i = top; i <= bottom; i++) {
        System.out.print(matrix[i][right] + " ");
    }
    right--;

    if (left > right) {
        break;
    }
    for (int i = right; i >= left; i--) {
        System.out.print(matrix[bottom][i] + " ");
    }
    bottom--;

    if (top > bottom) {
        break;
    }
    for (int i = bottom; i >= top; i--) {
        System.out.print(matrix[i][left] + " ");
    }
    left++;
}
}
}

```

## Anonymous Array

An anonymous array is a nameless array created for instant use. Its main objective is for one-time use.

Declaration syntax:

```
new int[]{10, 20, 30};  
new int[][]{{10, 20, 30}, {40, 50, 60}};
```

Example:

```
class Test {  
    public static void main(String[] args) {  
        sum(new int[]{10, 20, 30});  
    }  
  
    public static void sum(int[] arr) {  
        int total = 0;  
        for (int i : arr) {  
            total += i;  
        }  
        System.out.println(total);  
    }  
}
```

Another example:

```
class Test {  
    public static void main(String[] args) {  
        System.out.println(sum(new int[]{10, 20, 30, 40}));  
    }  
  
    public static int sum(int[] arr) {  
        int total = 0;  
        for (int i : arr) {  
            total += i;  
        }  
        return total;  
    }  
}
```

## Difference between length and length()

### length

- It's a final variable applicable only for arrays.

- It returns the size of an array.

Example:

```
class Test {  
    public static void main(String[] args) {  
        int[] arr = new int[4];  
        System.out.println(arr.length);  
    }  
}
```

## length()

- It's a final method applicable for String objects.
- It returns the number of characters in a string.

Example:

```
class Test {  
    public static void main(String[] args) {  
        String str = "ihubtalent";  
        System.out.println(str.length()); // 10  
    }  
}
```

# Interview Question

## Segregate Array Elements Using While Loop

Input:

1 0 0 1 1 0 1 1 0 0

Output:

0 0 0 0 0 1 1 1 1 1

```
class Test {
```

```

public static void main(String[] args) {
    int[] arr = {1, 0, 0, 1, 1, 0, 1, 1, 0, 0};

    int start = 0;
    int end = arr.length - 1;

    while (start < end) {
        if (arr[start] == 0) {
            start++;
        } else if (arr[end] == 1) {
            end--;
        } else {
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
        }
    }

    // Display
    for (int i : arr) {
        System.out.print(i + " ");
    }
}

```

# OOPS (Object-Oriented Programming System/Structure)

OOPS was introduced to deal with real-world entities using programming languages. A language is considered object-oriented if it supports the following features:

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

## Class



A class is:

- A blueprint of an object
- A collection of objects
- A logical entity

Class declaration syntax:

```
[Modifier] class ClassName [extends ParentClassName] / [implements InterfaceName] {  
    // Set of objects  
}
```

A class can accept the following modifiers:

- **default**
- **public**
- **final**
- **abstract**

## Difference between default class and public class

default class	public class
To declare default class we should not use any access modifier.	To declare public class we should use public access modifier.
It we declare any class as default then we can access that class within the package.	It we declare any class as public then we can access that class within the package and outside the package.

## Difference between final class and abstract class

final class	abstract class
To declare final class we will use final keyword.	To declare abstract class we will use abstract keyword.
We can't create child class (Not inherited).	We can create child class (inherited).
Object creation is possible (instantiate).	Object creation is not possible.

## Object

An object is:

- An outcome of a blueprint
- An instance of a class (allocating memory for data members)

- A collection of properties and behaviors
- A physical entity

Memory space is created when an object is instantiated. Multiple objects can be created in a single class.

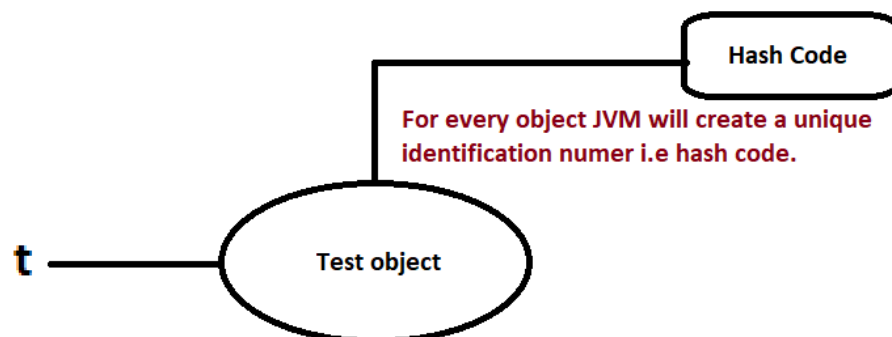
Example:

```
class Test {  
    public static void main(String[] args) {  
        Test t1 = new Test();  
        Test t2 = new Test();  
        Test t3 = new Test();  
  
        System.out.println(t1.hashCode());  
        System.out.println(t2.hashCode());  
        System.out.println(t3.hashCode());  
  
        System.out.println(t1); // Test@Hexadecimalvalue  
        System.out.println(t2.toString());  
        System.out.println(t3.toString());  
    }  
}
```

## HashCode in Java

For every object, JVM creates a unique identification number called a hashcode. The `hashCode()` method, present in the Object class, is used to read an object's hashcode.

```
Test t = new Test();
```



## toString() Method

The `toString()` method is present in the `Object` class. It's executed whenever we try to display an object reference directly or indirectly.

Example:

```
class Test {
    public static void main(String[] args) {
        int[] arr = new int[3];
        System.out.println(arr.toString()); // [I@hexavalue
    }
}
```

## Difference between Class and Object

class	object
It is a blueprint or template for an object.	It is an instance of a class.
It is a logically entity.	It is a physical entity.
It does not allocate the memory.	It allocates the memory.
It can't manipulate.	It can manipulate.
It is declared once.	It is declared many times.
To declare a class we will use class keyword.	To declare object we will use new keyword.

## Data Hiding

Data hiding is the process of hiding object data within the class. It prevents direct access to internal data from outside the class, providing security. The 'private' modifier is used to implement data hiding.

Example:

```
class Kotak {
    private double balance = 10000d;
}

class Student {
    public static void main(String[] args) {
        Kotak k = new Kotak();
        System.out.println(k.balance); // Compilation Error: balance has
private access in Kotak
    }
}
```

## Abstraction

Abstraction is the process of hiding internal implementation details while highlighting the set of services provided. It can be implemented using abstract classes and interfaces.

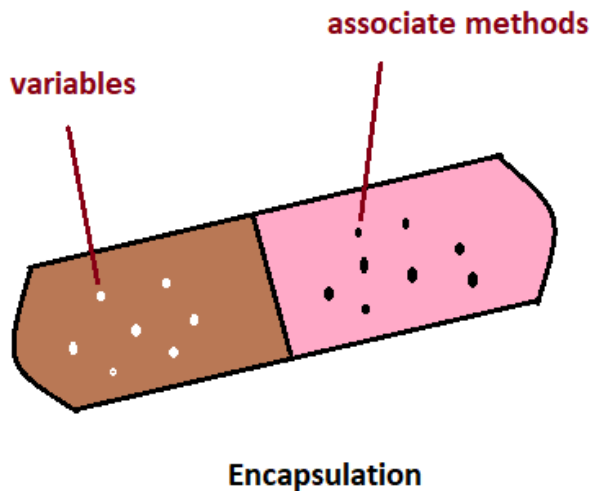
A prime example of abstraction is the GUI of an ATM machine. Banks conceal the internal workings while presenting a set of services such as banking, withdrawal, and mini statement generation.

Key advantages of abstraction:

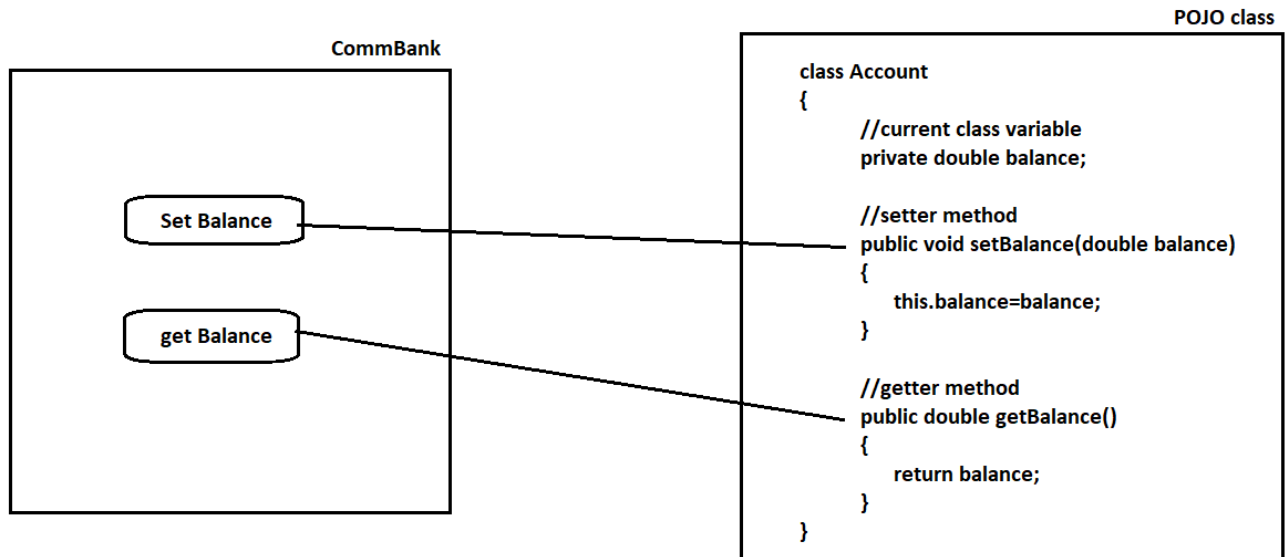
1. Enhanced security through hidden internal implementation
2. Easier enhancement by allowing internal changes without affecting end-users
3. Increased flexibility for end-users
4. Improved application maintainability

## Encapsulation

Encapsulation is the process of bundling variables and their associated methods into a single entity.



A class is considered encapsulated if it supports both data hiding and abstraction.



In encapsulation, setter and getter methods are written for each variable.

Main advantages of encapsulation:

1. Improved security
2. Easier enhancement
3. Increased flexibility for end-users
4. Better application maintainability

The primary disadvantage of encapsulation is increased code length and slower execution.

### Example:

```
class Student {
    // Current class variables
    private int studId;
    private String studName;
    private double studFee;

    // Setter methods
    public void setStudId(int studId) {
        this.studId = studId;
    }
    public void setStudName(String studName) {
        this.studName = studName;
    }
    public void setStudFee(double studFee) {
        this.studFee = studFee;
    }
}
```

```

    }

    // Getter methods
    public int getStudId() {
        return studId;
    }
    public String getStudName() {
        return studName;
    }
    public double getStudFee() {
        return studFee;
    }
}

class Test {
    public static void main(String[] args) {
        Student s = new Student();
        s.setStudId(101);
        s.setStudName("Alan Morries");
        s.setStudFee(1000d);

        System.out.println("Student Id : " + s.getStudId());
        System.out.println("Student Name : " + s.getStudName());
        System.out.println("Student Fee : " + s.getStudFee());
    }
}

```

## Comparison: Abstraction vs. Encapsulation

Abstraction	Encapsulation
Hiding internal implementation and highlighting the set of services is called abstraction.	The process of encapsulating or grouping variables and its associate methods in a single entity is called encapsulation.
It is used to hide the data.	It is used to protect the data.
Using abstract classes and interfaces we can implements abstraction.	Using access modifiers we can implements encapsulation.
It is a process of gaining the information.	It is a process of containing the information.
It solves an issue at design level.	It solves an issue at implementation level.

## Comparison: POJO Class vs. Java Bean Class

POJO	Java Bean
It can't be serialized.	It can be serialized.
Fields can have any visibility.	Fields can have only private visibility.
There may or may not have 0-arg constructor.	It must have 0-argument constructor.
It does not extend any other class.	It can extends.
It does not implement any other interface.	It can implements.
It does not use any outside annotation.	It uses outside annotation.

**Note: Every Java Bean class is a POJO class, but not every POJO class is a Java Bean class.**

## Is-A Relationship

Is-A relationship, also known as inheritance, is implemented using the **extends** keyword. Its main objective is to provide reusability.

Example:

```
class Parent {
```

```

    public void m1() {
        System.out.println("M1-Method");
    }
}

class Child extends Parent {
    public void m2() {
        System.out.println("M2-Method");
    }
}

class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.m1();

        Child c = new Child();
        c.m1();
        c.m2();

        Parent p1 = new Child();
        p1.m1();

        // Child c1 = new Parent(); // Compile-Time Error
    }
}

```

Conclusion:

- Properties of the parent are inherited by the child.
- Properties of the child are not accessible to the parent.
- A parent reference can hold a child object, but a child reference cannot hold a parent object.

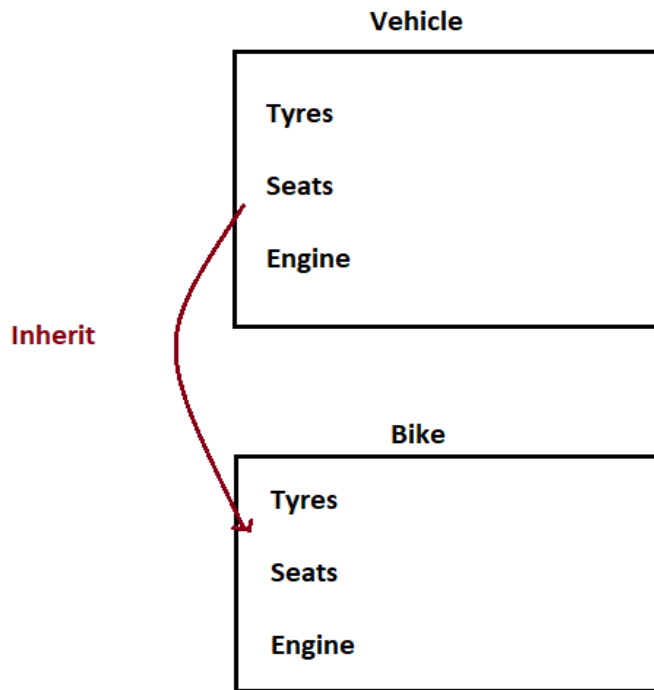
# Inheritance

Inheritance is a mechanism where one class inherits the properties of another class. It allows us to derive a new class from an existing class.

The main objective of inheritance is to provide reusability.

We implement inheritance using the "extends" keyword.





There are several types of inheritance:

1. Single Level Inheritance
2. Multi Level Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

## 1. Single Level Inheritance

Single level inheritance occurs when we derive a class from a single base class.

Diagram:

```
A (parent/base/super class)
|
|
B (child/derived/sub class)
```

Example:

```
class A {
    int i = 10;
```

```

}

class B extends A {
    int j = 20;
}

class Test {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.i); // 10

        B b = new B();
        System.out.println(b.i); // 10
        System.out.println(b.j); // 20
    }
}

```

Another example:

```

class A {
    public void m1() {
        System.out.println("M1-Method");
    }
}

class B extends A {
    public void m2() {
        System.out.println("M2-Method");
    }
}

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

        B b = new B();
        b.m1();
        b.m2();
    }
}

```

## 2. Multi Level Inheritance

Multi level inheritance occurs when we derive a class from a base class, which is itself derived from another base class.

Diagram:



Example:

```
class A {
    public void m1() {
        System.out.println("M1-Method");
    }
}

class B extends A {
    public void m2() {
        System.out.println("M2-Method");
    }
}

class C extends B {
    public void m3() {
        System.out.println("M3-Method");
    }
}

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

        B b = new B();
        b.m1();
    }
}
```

```

        b.m2();

        C c = new C();
        c.m1();
        c.m2();
        c.m3();
    }
}

```

### 3. Multiple Inheritance

Java doesn't support multiple inheritance for classes. A class cannot extend more than one class simultaneously.

Example (invalid):

```

class A {}
class B {}
class C extends A, B {} // Invalid

```

However, an interface can extend multiple interfaces:

```

interface A {}
interface B {}
interface C extends A, B {} // Valid

```

Note:

- If a class doesn't extend any other class, it's a direct child class of the Object class.

ex:	diag:
<pre> class A { } </pre>	<pre> Object     A </pre>

- If a class extends another class, it's an indirect child class of the Object class.

```

ex:                                diag:
class A                            Object
{                                  |
}                                  |
class B extends A                  A
{                                  |
}                                  |
                                   B

```

- Java doesn't support cyclic inheritance.

```

ex:
class A extends B
{
}
class B extends A
{
}

```

Q: Why doesn't Java support multiple inheritance?

A: Java doesn't support multiple inheritance to avoid potential ambiguity problems.

```

ex:
P1.m1()                            P2.m1()
|-----|
|
c.m1();

```

## 4. Hierarchical Inheritance

Hierarchical inheritance occurs when we derive multiple classes from a single base class.

Diagram:



Example:

```
class A {
    public void m1() {
        System.out.println("M1-Method");
    }
}

class B extends A {
    public void m2() {
        System.out.println("M2-Method");
    }
}

class C extends A {
    public void m3() {
        System.out.println("M3-Method");
    }
}

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

        B b = new B();
        b.m1();
        b.m2();

        C c = new C();
        c.m1();
        c.m3();
    }
}
```

## 5. Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. Java doesn't support hybrid inheritance for classes.

Diag:



## Has-A Relationship

Has-A relationship is also known as composition and aggregation. There's no specific keyword to implement Has-A relationship, but we often use the `new` operator.

The main objective of Has-A relationship is to provide reusability. It increases dependency between two components.

Example:

```
class Ihub {
    public String courseName() {
        return "FSD-JAVA";
    }
    public String trainerName() {
        return "Niyaz Sir";
    }
    public double courseFee() {
        return 30000d;
    }
}
class Usha {
    public void getCourseDetails() {
        Ihub i = new Ihub();
        System.out.println("Course Name : " + i.courseName());
        System.out.println("Trainer Name : " + i.trainerName());
        System.out.println("Course Fee : " + i.courseFee());
    }
}
```

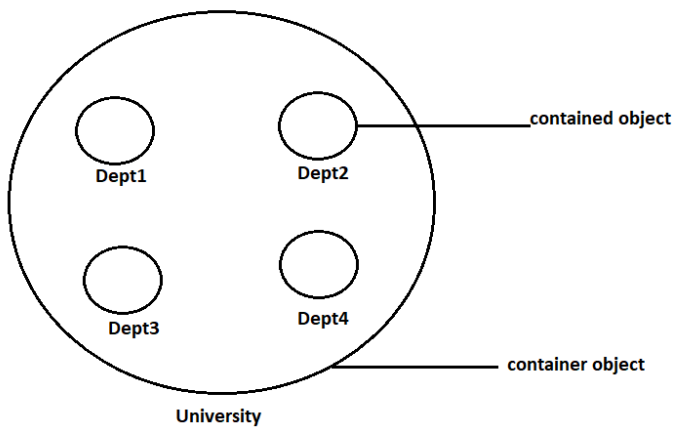
```

    }
}
class Student {
    public static void main(String[] args) {
        Usha u = new Usha();
        u.getCourseDetails();
    }
}

```

## Composition

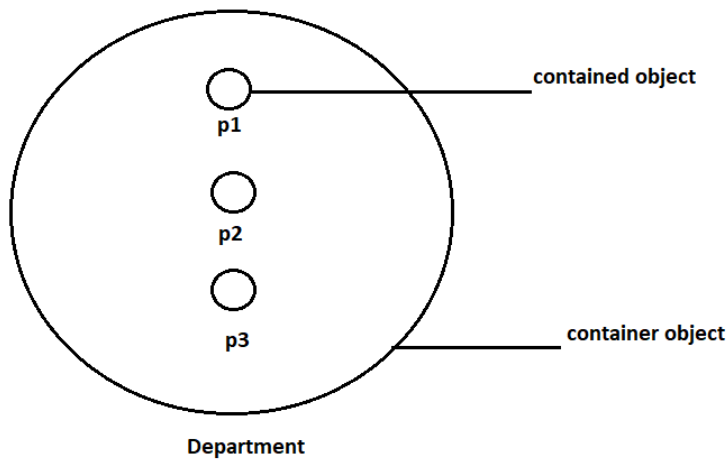
Composition is a strong association where the contained object cannot exist without the container object.



## Aggregation

Aggregation is a loose association where the contained object can exist independently of the container object.





## Method Overloading

Method overloading occurs when we have multiple methods with the same name but different parameters/signatures in a single class.

Method overloading reduces the complexity of programming.

All methods with the same name in a class are called overloaded methods.

Example:

```
class MeeSeva {  
    // Overloaded methods  
    public void search(int voterId) {  
        System.out.println("Details Found via voterId");  
    }  
    public void search(String houseNo) {  
        System.out.println("Details Found via houseNo");  
    }  
    public void search(long aadharNo) {  
        System.out.println("Details Found via aadharNo");  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        MeeSeva ms = new MeeSeva();  
        ms.search(101);  
    }  
}
```

```
ms.search("6-4/93/A/1");  
ms.search(12345L);  
}  
}
```

## Method Overriding

Method overriding occurs when two different classes have the same method name with identical parameters/signatures.

- Methods in the parent class are called overridden methods.
- Methods in the child class are called overriding methods.

Example:

```
class Parent {  
    public void property() {  
        System.out.println("Cash+Gold+Land+House");  
    }  
    // overridden method  
    public void marry() {  
        System.out.println("Rashmika");  
    }  
}  
  
class Child extends Parent {  
    // overriding method  
    public void marry() {  
        System.out.println("Trisha");  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        Parent p = new Parent();  
        p.property(); // Cash+Gold+Land+House  
        p.marry(); // Rashmika  
  
        Child c = new Child();  
    }  
}
```

```

        c.property(); // Cash+Gold+Land+House
        c.marry(); // Trisha

        Parent p1 = new Child();
        p1.property(); // Cash+Gold+Land+House
        p1.marry(); // Trisha
    }
}

```

Note: If we declare a method as final, overriding of that method is not possible.

Example:

```

class Parent {
    public final void property() {
        System.out.println("House-Not For Sale");
    }
}

class Child extends Parent {
    public final void property() {
        System.out.println("House- For Sale");
    }
}

class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.property(); // House-Not For Sale

        Child c = new Child();
        c.property(); // Compile Time Error
    }
}

```

If a parent doesn't want to give any property to the child, we need to declare the property as private.

Example:

```

class Parent {
    protected void company() {

```

```

        System.out.println("Organization");
    }
    private void property() {
        System.out.println("House");
    }
}

class Child extends Parent {
    private void property() {
        System.out.println("House-Owners");
    }
}

class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.company(); // Organization
        p.property(); // Compile Time Error

        Child c = new Child();
        c.company(); // Organization
    }
}

```

## Method Hiding

Method hiding is similar to method overriding with the following differences:

Aspect	Method Overriding	Method Hiding
Method Type	No method is static	All methods are static
Method Resolution	Method resolution is done by JVM based on runtime object	Method resolution is done by compiler based on reference type
Polymorphism Type	Known as runtime polymorphism, dynamic polymorphism, or late binding	Known as compile-time polymorphism, static polymorphism, or early binding

Example:

```

class Parent {

```

```

    public static void property() {
        System.out.println("Cash+Gold+Land+House");
    }
    // hidden method
    public static void marry() {
        System.out.println("Rashmika");
    }
}

class Child extends Parent {
    // hiding method
    public static void marry() {
        System.out.println("Trisha");
    }
}

class Test {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.property(); // Cash+Gold+Land+House
        p.marry(); // Rashmika

        Child c = new Child();
        c.property(); // Cash+Gold+Land+House
        c.marry(); // Trisha

        Parent p1 = new Child();
        p1.property(); // Cash+Gold+Land+House
        p1.marry(); // Rashmika
    }
}

```

## Interview Questions

Q: Can we overload the main method in Java? A: Yes, we can overload the main method in Java. However, JVM always executes the main method with String[] parameter only.

Example:

```

class Test {
    public static void main(int[] iargs) {

```

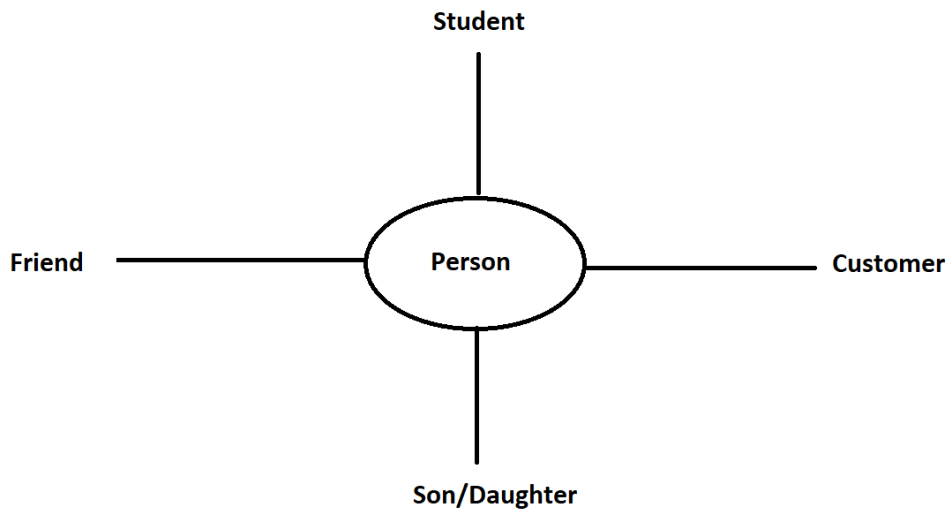
```
    System.out.println("int array");  
}  
public static void main(String[] args) {  
    System.out.println("string array");  
}  
}
```

Q: Can we override the main method in Java?

A: No, we can't override the main method in Java because it is static.

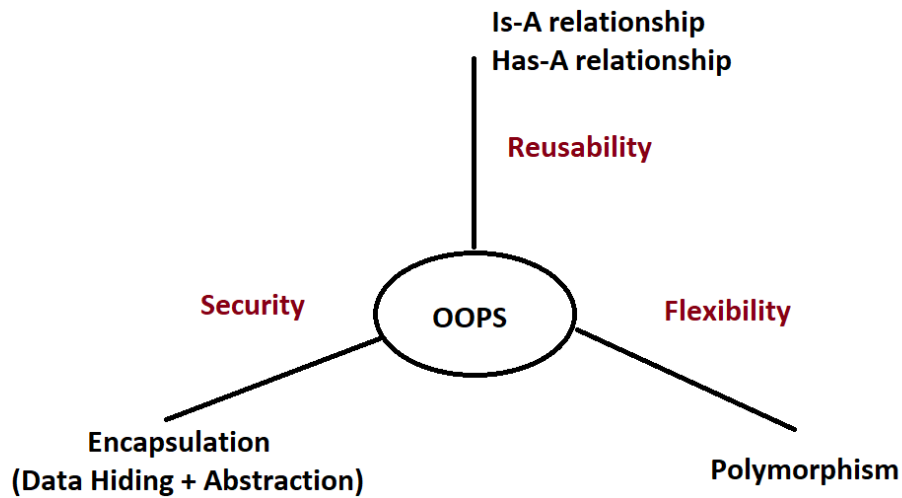
# Polymorphism

Polymorphism comes from the Greek word meaning "many forms." It is the ability to represent in different forms. The main objective of polymorphism is to provide flexibility.



Polymorphism in Java is divided into two types:

1. Compile-time polymorphism (Static polymorphism / Early binding)
  - Examples: Method overloading, Method hiding
2. Runtime polymorphism (Dynamic polymorphism / Late binding)
  - Example: Method overriding



# Constructors

A constructor is a special method used to initialize an object. It has the same name as the class and is executed when we create an object. It doesn't have a return type.

Constructors can have the following modifiers:

- default
- public
- private
- protected

Types of constructors:

1. User-defined constructor
2. Default constructor

## User-defined Constructor

User-defined constructors are created by the user based on application requirements. They are classified into two types:

- i. Zero-Argument constructor
- ii. Parameterized constructor

## Zero-Argument Constructor

A zero-argument constructor doesn't take any arguments.

Example:

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

Output:

```
Main-Method
```

## Parameterized Constructor

A parameterized constructor takes at least one argument.

Example:

```
class Employee {
    private int empId;
    private String empName;
    private double empSal;

    public Employee(int empId, String empName, double empSal) {
        this.empId = empId;
        this.empName = empName;
        this.empSal = empSal;
    }

    public void getEmployeeDetails() {
        System.out.println("Employee Id: " + empId);
        System.out.println("Employee Name: " + empName);
        System.out.println("Employee Salary: " + empSal);
    }
}
```




```
class Test {
    public static void main(String[] args) {
        Employee e = new Employee(101, "Alan Morries", 1000d);
        e.getEmployeeDetails();
    }
}
```

## Default Constructor

A default constructor is a compiler-generated constructor for every Java program where we are not defining at least a zero-argument constructor.

To see the default constructor, use the command:

```
javap -c Test
```

<pre>class Test {     public static void main(String[] args)     {         System.out.println("Welcome Suman");     } }</pre>		<pre>class Test {     //default constructor     Test();      public static void main(String[] args)     {         System.out.println("Welcome Suman");     } }</pre>
---	---	--

## Constructor Overloading

Constructor overloading is having multiple constructors with different parameters/signatures in a single class.

Example:

```
class A {
    A() {
        System.out.println("0-arg const");
    }
}
```

```

    A(int i) {
        System.out.println("int-arg const");
    }
    A(double d) {
        System.out.println("double-arg const");
    }
}

class Test {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A(10);
        A a3 = new A(10.5d);
    }
}

```

## Interview Question

Q: What is a tightly encapsulated class?

A: A class is said to be tightly encapsulated if and only if all variables of that class are declared as private. We don't need to check if these variables have setter and getter methods.

Examples:

```

class A {
    int i = 10;
}
// Not a tightly encapsulated class

class A {
    int i = 10;
    private int j = 20;
}
// Not a tightly encapsulated class

class A {
    private int i = 10;
    private int j = 20;
}
// Tightly encapsulated class

```

```

class A {
    int i = 10;
}
class B extends A {
    private int j = 20;
}
// If no parent is tightly encapsulated, then no child is tightly encapsulated

```

## The 'this' Keyword in Java

The 'this' keyword in Java is used to refer to the current class object reference. It can be utilized in the following ways:

1. To refer to current class variables
2. To refer to current class methods
3. To refer to current class constructors

### 1. Referring to Current Class Variables

```

class A {
    int i = 10;
    int j = 20;

    A(int i, int j) {
        System.out.println(i + " " + j);           // Outputs: 100 200
        System.out.println(this.i + " " + this.j); // Outputs: 10 20
    }
}

class Test {
    public static void main(String[] args) {
        A a = new A(100, 200);
    }
}

```

### 2. Referring to Current Class Methods

```

class A {
    public void m1() {

```

```

        System.out.println("M1-Method");
        this.m2();
    }

    public void m2() {
        System.out.println("M2-Method");
    }
}

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

```

### 3. Referring to Current Class Constructors

```

class A {
    A() {
        System.out.println("0-arg const");
    }

    A(int i) {
        this();
        System.out.println("int-arg const");
    }

    A(double d) {
        this(10);
        System.out.println("double-arg const");
    }
}

class Test {
    public static void main(String[] args) {
        A a = new A(10.5d);
    }
}

```

# The 'super' Keyword in Java

The 'super' keyword in Java is used to refer to the superclass object reference. It can be utilized in the following ways:

1. To refer to superclass variables
2. To refer to superclass methods
3. To refer to superclass constructors

## 1. Referring to Superclass Variables

```
class A {
    int i = 10;
    int j = 20;
}

class B extends A {
    int i = 100;
    int j = 200;

    B(int i, int j) {
        System.out.println(this.i + " " + this.j); // Outputs: 100 200
        System.out.println(super.i + " " + super.j); // Outputs: 10 20
        System.out.println(i + " " + j); // Outputs: 1000 2000
    }
}

class Test {
    public static void main(String[] args) {
        B b = new B(1000, 2000);
    }
}
```

## 2. Referring to Superclass Methods

```
class A {
    public void m1() {
        System.out.println("M1-Method");
    }
}
```

```

}

class B extends A {
    public void m2() {
        super.m1();
        System.out.println("M2-Method");
    }
}

class Test {
    public static void main(String[] args) {
        B b = new B();
        b.m2();
    }
}

```

### 3. Referring to Superclass Constructors

```

class A {
    A() {
        System.out.println("A-const");
    }
}

class B extends A {
    B() {
        // super(); // This is implicitly called if not explicitly written
        System.out.println("B-const");
    }
}

class Test {
    public static void main(String[] args) {
        B b = new B();
    }
}

```

## Has-A Relationship Program

```
class A {
    A() {
        System.out.println("A-const");
    }
}

class B extends A {
    B() {
        // super(); // This is implicitly called if not explicitly written
        System.out.println("B-const");
    }
}

class Test {
    public static void main(String[] args) {
        B b = new B();
    }
}

class Employee {
    private int empId;
    private String empName;
    private Address address;

    // Parameterized constructor
    public Employee(int empId, String empName, Address address) {
        this.empId = empId;
        this.empName = empName;
        this.address = address;
    }

    // Getter methods
    public int getEmpId() {
        return empId;
    }

    public String getEmpName() {
        return empName;
    }

    public Address getAddress() {
        return address;
    }
}
```

```

    @Override
    public String toString() {
        return "\nEmployee Id: " + getEmpId() + "\nEmployee Name: " +
getEmpName() + "\nEmployee Address: " + getAddress();
    }
}

class Address {
    private String houseNo;
    private String locality;
    private String city;

    // Parameterized constructor
    public Address(String houseNo, String locality, String city) {
        this.houseNo = houseNo;
        this.locality = locality;
        this.city = city;
    }

    // Getter methods
    public String getHouseNo() {
        return houseNo;
    }

    public String getLocality() {
        return locality;
    }

    public String getCity() {
        return city;
    }

    @Override
    public String toString() {
        return "\nHouse No: " + getHouseNo() + "\nLocality: " +
getLocality() + "\nCity: " + getCity();
    }
}

class Test {
    public static void main(String[] args) {
        Address address = new Address("1-4-64/1", "Ameerpet", "Hyderabad");
        Employee e = new Employee(201, "Alan Morries", address);
    }
}

```



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

# Interfaces in Java

An interface is a collection of zero or more abstract methods. Abstract methods are incomplete methods because they end with a semicolon and do not have any body.

Example:

```
void m1();
```

Key points about interfaces:

- It is not possible to create objects for interfaces.
- To implement abstract methods of an interface, we use an implementation class.
- By default, every abstract method in an interface is public and abstract.
- Interfaces can only contain constants.

Interface declaration syntax:

```
interface <interface_name> {  
    // abstract methods  
    // constants
```

**Use interfaces when you know the service requirement specification.**

Example 1:

```
interface A {  
    public abstract void m1();  
}  
  
class B implements A {  
    public void m1() {  
        System.out.println("M1-Method");  
    }  
}
```

```

class Test {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

Example 2 (Anonymous inner class):

```

interface A {
    public abstract void m1();
}

class Test {
    public static void main(String[] args) {
        A a = new A() {
            public void m1() {
                System.out.println("From M1 Method");
            }
        };
        a.m1();
    }
}

```

**Note:** If an interface contains multiple methods, we need to override all methods in the implementing class; otherwise, we will get a compile-time error.

**Example with multiple methods:**

```

interface A {
    public void show();
    abstract void display();
    public abstract void view();
    void see();
}

class B implements A {
    @Override
    public void show() {
        System.out.println("show-method");
    }
}

```

```

@Override
public void display() {
    System.out.println("display-method");
}

@Override
public void view() {
    System.out.println("view-method");
}

@Override
public void see() {
    System.out.println("see-method");
}
}

class Test {
    public static void main(String[] args) {
        A a = new B();
        a.show();
        a.display();
        a.view();
        a.see();
    }
}

```

**An interface can extend more than one interface:**

```

interface A {
    void m1();
}

interface B {
    void m2();
}

interface C extends A, B {
    void m3();
}

class D implements C {
    public void m1() {

```

```

        System.out.println("M1-Method");
    }

    public void m2() {
        System.out.println("M2-Method");
    }

    public void m3() {
        System.out.println("M3-Method");
    }
}

class Test {
    public static void main(String[] args) {
        C c = new D();
        c.m1();
        c.m2();
        c.m3();
    }
}

```

**A class can implement more than one interface:**

```

interface Father {
    float HT = 6.2f;
    void height();
}

interface Mother {
    float HT = 5.8f;
    void height();
}

class Child implements Father, Mother {
    public void height() {
        float height = (Father.HT + Mother.HT) / 2;
        System.out.println("Child Height: " + height);
    }
}

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

```

```
        Child c = new Child();  
        c.height();  
    }  
}
```

**Note:** According to Java 8 version, an interface is a collection of abstract methods, default methods, and static methods.

## Marker Interface

A marker interface is an interface that does not have any methods or constants. It's generally referred to as an empty interface.

Marker interfaces provide certain abilities to the classes that implement them. Some common marker interfaces include:

- Serializable
- Cloneable
- Remote

Example:

```
class Student implements java.io.Serializable {  
    private int studId;  
  
    // Setter method  
    public void setStudId(int studId) {  
        this.studId = studId;  
    }  
  
    // Getter method  
    public int getStudId() {  
        return studId;  
    }  
}
```

## Abstract Class

An abstract class is a collection of zero or more abstract methods and concrete methods. Key points about abstract classes:

- The **abstract** keyword is applicable for methods and classes, but not for variables.
- It's not possible to create an object of an abstract class.
- Subclasses are used to implement the abstract methods of an abstract class.
- Every abstract method is public and abstract by default.
- Abstract classes contain only instance variables.

Syntax for declaring an abstract class:

```
class Student implements java.io.Serializable {
    private int studId;

    // Setter method
    public void setStudId(int studId) {
        this.studId = studId;
    }

    // Getter method
    public int getStudId() {
        return studId;
    }
}
```

**Use abstract classes when you know partial implementation.**

Example:

```
abstract class Plan {
    protected double rate;

    public abstract void getRate();

    public void getBillDetails(int units) {
        System.out.println("Total Units: " + units);
        System.out.println("Total Bill: " + units * rate);
    }
}

class DomesticPlan extends Plan {
    public void getRate() {
```

```

        rate = 2.5d;
    }
}

class CommercialPlan extends Plan {
    public void getRate() {
        rate = 5.0d;
    }
}

class Test {
    public static void main(String[] args) {
        DomesticPlan dp = new DomesticPlan();
        dp.getRate();
        dp.getBillDetails(250);

        CommercialPlan cp = new CommercialPlan();
        cp.getRate();
        cp.getBillDetails(250);
    }
}

```

## Difference between Interface and Abstract Class

Interface	Abstract Class
Declared using `interface` keyword	Declared using `abstract` keyword
Collection of abstract, default, and static methods	Collection of abstract and concrete methods
Contains constants	Contains instance variables
Supports multiple inheritance	Doesn't support multiple inheritance
Doesn't allow blocks	Allows blocks
Doesn't allow constructors	Allows constructors
Uses implementation class for abstract methods	Uses subclass for abstract methods
Used when only specification is known	Used when partial implementation is known

## Abstraction

Abstraction is the concept of hiding internal implementation and highlighting the set of services. It can be implemented using interfaces and abstract classes.

ex:

GUI ATM Machine

Example:

```
abstract class Animal
{
    //abstract method
    public abstract void sound();
}
class Cat extends Animal
{
    public void sound()
    {
        System.out.println("Meow Meow");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Cat c=new Cat();
        c.sound();
    }
}
```

Ex:

```
abstract class Animal {
    public abstract void sound();

    public static void behaviour() {
        System.out.println("Eating+Jumping+Sleeping");
    }
}

class Cat extends Animal {
```



```
public void sound() {  
    System.out.println("Meow Meow");  
}  
}  
  
class Test {  
    public static void main(String[] args) {  
        Cat c = new Cat();  
        c.sound();  
        c.behaviour();  
    }  
}
```

# API (Application Programming Interface)

An API is a base for programmers to develop software applications. It's a collection of packages. There are three types of APIs:

1. Predefined API : Built-in API (e.g., Java Standard Library)
2. User-defined API : Created by the user based on application requirements
3. Third-party API : Created by third-party vendors (e.g., javazoom API, iText API)

## Package

A package is a collection of classes, interfaces, enums, and annotations. It's also known as a folder or directory. There are two types of packages in Java:

1. Predefined packages : Built-in packages (e.g., java.lang, java.io, java.util)
2. User-defined packages : Created by the user based on application requirements

## User-defined packages

Created by the user based on application requirements

To declare a custom package, use the `package` keyword. It's recommended to declare package names in reverse order of the URL.

syntax:

```
package    <package_name>;
```

ex:

```
package    com.qualitythought.www;  
package    com.google.www;  
package    com.ihubtalent.www;
```

Example:

```
package com.ihub.www;  
  
import java.util.Calendar;  
  
class Test {  
    public static void main(String[] args) {  
        Calendar c = Calendar.getInstance();  
        int h = c.get(Calendar.HOUR_OF_DAY);  
  
        if (h < 12)  
            System.out.println("Good Morning");  
        else if (h < 16)  
            System.out.println("Good Afternoon");  
        else if (h < 20)  
            System.out.println("Good Evening");  
        else  
            System.out.println("Good Night");  
    }  
}
```

Compile the program:

```
current directory  
|  
javac -d . Test.java  
|  
destination location
```

Run the program:

```
classname  
|
```

```
java    com.ihub.www.Test
      |
package name
```

# Singleton Class

A singleton class allows the creation of only one object. It's characterized by a method that returns the same class object when called using the class name.

To create a singleton class, you need a private constructor and a static method.

Example:

```
class Singleton {
    static Singleton singleton = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}

class Test {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        System.out.println(s1.hashCode());

        Singleton s2 = Singleton.getInstance();
        System.out.println(s2.hashCode());
    }
}
```

# Enum

An enum is a group of named constants. The enum concept was introduced in Java 1.5v.

Using enums, we can create our own datatype called an enumerated datatype.

Java enums are more powerful compared to enums in older languages.

## Enum Declaration

Syntax:

```
enum type_name {  
    value1, value2, ..., valueN  
}
```

Example:

```
enum Months {  
    JAN, FEB, MAR  
}
```

## Internal Implementation of Enum

Every enum is internally considered as a class concept and extends the `java.lang.Enum` class. Each enum constant is a reference variable of the enum type.

For example:

```
enum Months {  
  
    JAN, FEB, MAR  
  
}
```

is internally implemented as:

```
final class Months extends java.lang.Enum {  
  
    public static final Months JAN = new Months();  
  
    public static final Months FEB = new Months();  
  
}
```

```
        public static final Months MAR = new Months();  
    }  
}
```

## Declaration and Usage of Enum

Example 1:

```
enum Months {  
    JAN, FEB, MAR  
}  
  
class Test {  
    public static void main(String[] args) {  
        Months m = Months.JAN;  
        System.out.println(m);  
    }  
}
```

Example 2:

```
enum Months {  
    JAN, FEB, MAR  
}  
  
class Test {  
    public static void main(String[] args) {  
        Months m = Months.FEB;  
        switch(m) {  
            case JAN: System.out.println("It is January"); break;  
        }  
    }  
}
```

```

        case FEB: System.out.println("It is February"); break;

        case MAR: System.out.println("It is March"); break;

    }

}
}

```

## java.lang.Enum

The power of enum is inherited from the `java.lang.Enum` class. It contains the following two methods:

1. `values()`: Returns a group of constants from the enum.
2. `ordinal()`: Returns the ordinal number of the constant.

Example:

```

enum Months {

    JAN, FEB, MAR

}

class Test {

    public static void main(String[] args) {

        Months[] m = Months.values();

        for(Months m1 : m) {

            System.out.println(m1 + "-----" + m1.ordinal());

        }

    }

}

```

Java enums are more powerful because, in addition to constants, we can declare constructors, variables, and methods.

Example with constructor:

```
enum Drinks {  
    PEPSI, COKE, CAMPA;  
  
    Drinks() {  
        System.out.println("constructor");  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        Drinks d = Drinks.PEPSI;  
    }  
}
```

Example with static variable and main method:

```
enum Week {  
    MON, TUE, WED, THU, FRI, SAT, SUN;  
  
    static int i = 10;  
  
    public static void main(String[] args) {  
        System.out.println(i);  
    }  
}
```

# Wrapper Classes

The main objectives of wrapper classes are:

1. To wrap primitives into wrapper objects and vice versa.
2. To define several utility methods.

Primitive types and their corresponding wrapper classes:

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

## Constructors



There are two ways to create objects for wrapper classes:

1. Using the corresponding primitive as an argument
2. Using the corresponding String as an argument

Example:

```
class Test {  
  
    public static void main(String[] args) {  
  
        Integer i1 = new Integer(10);  
  
        System.out.println(i1); // 10  
  
  
        Integer i2 = new Integer("20");  
  
        System.out.println(i2); // 20  
  
  
        Boolean b1 = new Boolean(true);  
  
        System.out.println(b1); // true  
  
  
        Boolean b2 = new Boolean("false");  
  
        System.out.println(b2); // false  
  
  
        Character c = new Character('a');  
  
        System.out.println(c); // a  
  
    }  
}
```

```
}
```

## Utility Methods

1. `xxxValue()` method: Converts wrapper object to primitive type.

```

class Test {

    public static void main(String[] args) {

        Integer i = new Integer(10);

        byte b = i.byteValue();

        System.out.println(b);

    }

}

```

2. `valueOf()` method: Converts primitive type to wrapper object.

```

class Test {

    public static void main(String[] args) {

        byte b = 10;

        Integer i = Integer.valueOf(b);

        System.out.println(i); // 10

        Long l = Long.valueOf(b);

        System.out.println(l); // 10

        Float f = Float.valueOf(b);

        System.out.println(f); // 10.0

    }

}

```

3. `parseXxx()` method: Converts string type to primitive type.

```

class Test {

```

```

public static void main(String[] args) {

    String str = "10";

    int i = Integer.parseInt(str);

    System.out.println(i); // 10

    long l = Long.parseLong(str);

    System.out.println(l); // 10

    float f = Float.parseFloat(str);

    System.out.println(f); // 10.0

}

```

```

}

```

4. `toString()` method: Converts wrapper object to string type.

```

class Test {

    public static void main(String[] args) {

        Integer i = new Integer(10);

        String s = i.toString();

        System.out.println(s);

    }
}

```

```

}

```

## Interview Question

Q: Write a Java program to perform the sum of two binary numbers.

Input:

Copy

1010

0101

Output:

Copy

1111

Solution:

java

Copy

```
import java.util.Scanner;

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

        System.out.println("Enter the first binary number:");
        String binary1 = sc.next(); // 1010

        System.out.println("Enter the second binary number:");
        String binary2 = sc.next(); // 0101

        // Convert binary to decimal
        int a = Integer.parseInt(binary1, 2);
        int b = Integer.parseInt(binary2, 2);

        int c = a + b;

        // Convert decimal to binary
        String result = Integer.toBinaryString(c);
        System.out.println(result);
    }
}
```

# Inner Classes

Sometimes we declare a class inside another class; this concept is called an inner class.

```
class Outer_class {  
    class Inner_class {  
        // code to be executed  
    }  
}
```

Inner classes were introduced as part of event handling to remove GUI bugs. Due to their powerful features and benefits, programmers started using inner classes in regular programming.

Inner classes do not support static declarations.

## Accessing Inner Class Data from Static Area of Outer Class

```
class Outer {  
    class Inner {  
        public void m1() {  
            System.out.println("M1-Method");  
        }  
    }  
  
    public static void main(String[] args) {  
        Outer.Inner i = new Outer().new Inner();  
        i.m1();  
    }  
}
```

When we compile the above program, we get two .class files: Outer.class and Outer\$Inner.class.

Alternative syntax:

```
class Outer {  
    class Inner {  
        public void m1() {  
            System.out.println("M1-Method");  
        }  
    }  
}
```

```

    }
}

public static void main(String[] args) {
    new Outer().new Inner().m1();
}
}

```

## Accessing Inner Class Data from Non-Static Area of Outer Class

```

class Outer {
    class Inner {
        public void m1() {
            System.out.println("M1-Method");
        }
    }

    public void m2() {
        Inner i = new Inner();
        i.m1();
    }

    public static void main(String[] args) {
        Outer o = new Outer();
        o.m2();
    }
}

```

## Types of Objects in Java

There are two types of objects in Java:

1. **Immutable Objects**
2. **Mutable Objects**

### 1. Immutable Objects

When changes are performed on an immutable object, a new object is created for each change rather than modifying the original object.

**Examples:**

- String
- Wrapper classes

## 2. Mutable Objects

When changes are performed on a mutable object, the modifications are made to the same object rather than creating a new one.

**Examples:**

- StringBuffer
- StringBuilder

# String in Java

A String is a collection of characters enclosed in double quotation marks.

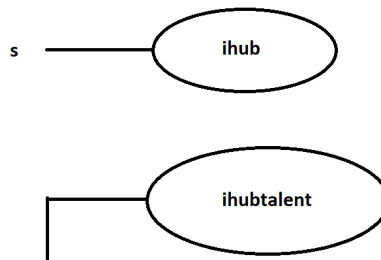
## Key Characteristics

### Case 1: Immutability

Once a String object is created, it cannot be modified. Any operation that appears to modify the String actually creates a new object. This behavior is known as immutability.

String Immutability Diagram

```
String s = new String("ihub");  
s.concat("talent");  
System.out.println(s); // ihub
```



No reference then it is eligible for garbage collector

## Case 2: Comparison Operators

### == Operator

- Used for address/reference comparison
- Can compare both primitive and object types
- Returns a boolean value (true or false)

Example:

```
class Test {  
    public static void main(String[] args) {  
        String s1 = new String("ihub");  
        String s2 = new String("ihub");  
        System.out.println(s1 == s2); // false  
    }  
}
```

### .equals() Method

- Used for content comparison (case-sensitive)
- Can only compare object types

Example:



```

class Test {
    public static void main(String[] args) {
        String s1 = new String("ihub");
        String s2 = new String("ihub");
        System.out.println(s1.equals(s2)); // true
    }
}

```

### Case 3: String Object Creation

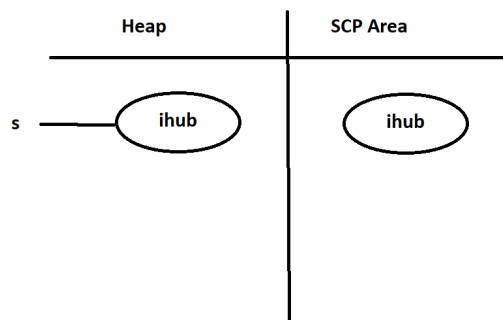
When a String object is created, two objects are actually created:

1. One in the heap memory
2. One in the String Constant Pool (SCP)

The reference always points to the heap object.

String Memory Diagram

**String s = new String("ihub");**



#### Important Notes:

- Object creation in SCP is optional
- JVM checks for existing objects with the same content in SCP
- No duplicate objects exist in SCP
- SCP objects cannot be accessed by the garbage collector
- SCP objects are destroyed when JVM shuts down

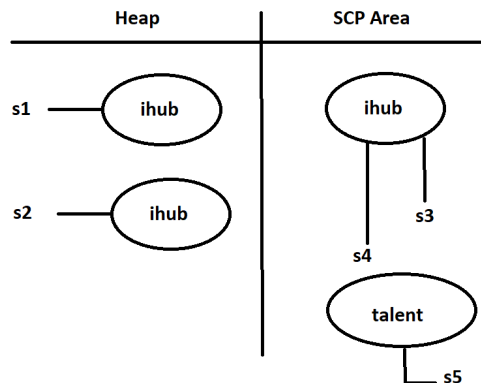
```
String s1=new String("ihub");
```

```
String s2=new String("ihub");
```

```
String s3="ihub";
```

```
String s4="ihub";
```

```
String s5="talent";
```



## String Interning

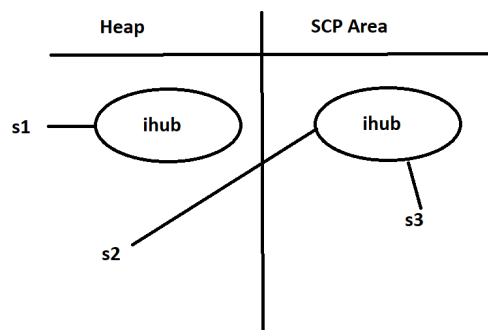
The `intern()` method can be used to get the SCP object reference from a heap object reference.

```
String s1=new String("ihub");
```

```
String s2=s1.intern();
```

```
String s3="ihub";
```

```
System.out.println(s2==s3);//true
```



# Important String Methods

## Finding String Length

```
class Test {  
    public static void main(String[] args) {  
        String str = "ihubtalent";  
        System.out.println(str.length()); // 10  
    }  
}
```

## Converting to Lowercase

```
class Test {  
    public static void main(String[] args) {  
        String str = "IHUBTALENT";  
        System.out.println(str.toLowerCase()); // ihubtalent  
    }  
}
```

## Converting to Uppercase

```
class Test {  
    public static void main(String[] args) {  
        String str = "ihubtalent";  
        System.out.println(str.toUpperCase()); // IHUBTALENT  
    }  
}
```

## Concatenating Strings

```
class Test {  
    public static void main(String[] args) {  
        String str1 = "ihub";  
        String str2 = "talent";  
        System.out.println(str1.concat(str2)); // ihubtalent  
    }  
}
```

## Displaying Characters from String

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

## Finding First Index of Character

```
class Test {  
    public static void main(String[] args) {  
        String str = "talentmanagement";  
        System.out.println(str.indexOf('e')); // 3  
    }  
}
```

## Finding Last Index of Character

```
class Test {  
    public static void main(String[] args) {  
        String str = "talentmanagement";  
        System.out.println(str.lastIndexOf('e')); // 13  
    }  
}
```

## Removing Spaces

```
class Test {  
    public static void main(String[] args) {  
        String str = "I hub Tal ent";  
        str = str.replaceAll("\\s", "");  
        System.out.println(str); // IhubTalent  
    }  
}
```

## Removing Special Characters

```
class Test {  
    public static void main(String[] args) {  
        String str = "I_hub@Ta#lent1$";  
        str = str.replaceAll("[^A-Za-z0-9]", "");  
        System.out.println(str); // IhubTalent1  
    }  
}
```

## Advanced String Concatenation

```
class Test {  
    public static void main(String[] args) {  
        String s1 = "Ihub20";  
        String s2 = "Talent30";  
  
        String word1 = s1.replaceAll("[^A-Za-z]", "");  
        int no1 = Integer.parseInt(s1.replaceAll("[^0-9]", ""));  
  
        String word2 = s2.replaceAll("[^A-Za-z]", "");  
        int no2 = Integer.parseInt(s2.replaceAll("[^0-9]", ""));  
  
        String word = word1 + word2;  
        int num = no1 + no2;  
  
        System.out.println(word + num); // IhubTalent50  
    }  
}
```

# Java String Operations

## 1. String Comparison

### Using equals()

```
class Test {  
    public static void main(String[] args) {  
        String s1 = "ihub";  
    }  
}
```

```

        String s2 = "ihub";

        if (s1.equals(s2))
            System.out.println("Both are same");
        else
            System.out.println("Both are not same");
    }
}

```

## Using equalsIgnoreCase()

```

class Test {
    public static void main(String[] args) {
        String s1 = "ihub";
        String s2 = "IHUB";

        if (s1.equalsIgnoreCase(s2))
            System.out.println("Both are same");
        else
            System.out.println("Both are not same");
    }
}

```

## 2. String Rotation

**Q) Write a Java program to perform right rotation of a given string?**

Input:

- str = ihubtalent
- cnt = 4

Output:

- talentihub

```

class Test {
    public static void main(String[] args) {
        String str = "ihubtalent";
        int cnt = 4;

        String word1 = str.substring(cnt, str.length());
    }
}

```

```

        String word2 = str.substring(0, cnt);

        System.out.println(word1 + word2);
    }
}

```

### 3. String Insertion

**Q) Write a Java program to insert a string at a specific index?**

Input:

- str = ihubstudent
- index = 4
- insert = for

Output:

- ihubforstudent

```

class Test {
    public static void main(String[] args) {
        String str = "ihubstudent";
        int index = 4;
        String insert = "for";

        String word1 = str.substring(0, index);
        String word2 = str.substring(index, str.length());

        System.out.println(word1 + insert + word2);
    }
}

```

### 4. String Reversal

**Q) Write a Java program to reverse a given string?**

Input:

- hello

Output:

- olleh

### Approach 1: Using charAt()

```
class Test {  
    public static void main(String[] args) {  
        String str = "hello";  
        String rev = "";  
  
        for (int i = str.length() - 1; i >= 0; i--) {  
            rev += str.charAt(i);  
        }  
        System.out.println(rev);  
    }  
}
```

### Approach 2: Using toCharArray()

```
class Test {  
    public static void main(String[] args) {  
        String str = "hello";  
        char[] carr = str.toCharArray();  
  
        String rev = "";  
        for (int i = carr.length - 1; i >= 0; i--) {  
            rev += carr[i];  
        }  
  
        System.out.println(rev);  
    }  
}
```

## 5. String Sorting

Q) Write a Java program to sort characters in a string?

Input:

- dzbea

Output:

- abdez



```
import java.util.Arrays;

class Test {
    public static void main(String[] args) {
        String str = "dzbea";
        char[] carr = str.toCharArray();

        Arrays.sort(carr);

        for (char c : carr) {
            System.out.print(c);
        }
    }
}
```

## 6. Palindrome Check

**Q) Write a Java program to check if a string is palindrome or not?**

Input:

- racar

Output:

- It is a palindrome string

```
class Test {
    public static void main(String[] args) {
        String str = "racar";

        String rev = "";
        for (int i = str.length() - 1; i >= 0; i--) {
            rev += str.charAt(i);
        }

        if (str.equals(rev))
            System.out.println("It is a palindrome string");
        else
            System.out.println("It is not a palindrome string");
    }
}
```

## 7. Sentence Reversal

Q) Write a Java program to reverse the order of words in a sentence?

Input:

- This is java class

Output:

- class java is This

```
class Test {  
    public static void main(String[] args) {  
        String str = "This is java class";  
        String[] sarr = str.split(" ");  
  
        for (int i = sarr.length - 1; i >= 0; i--) {  
            System.out.print(sarr[i] + " ");  
        }  
    }  
}
```

## 8. Word Reversal in Sentence

Q) Write a Java program to reverse each word in a sentence?

Input:

- This is java class

Output:

- sihT si avaj ssalc

```
class Test {  
    public static void main(String[] args) {  
        String str = "This is java class";  
        String[] sarr = str.split(" ");  
  
        for (String s : sarr) {  
            char[] carr = s.toCharArray();  
        }  
    }  
}
```

```

        for (int i = carr.length - 1; i >= 0; i--) {
            System.out.print(carr[i]);
        }
        System.out.print(" ");
    }
}

```

## 9. Uppercase Word Filter

**Q) Write a Java program to display words starting with uppercase letters?**

Input:

- This is Java class For students

Output:

- This Java For

```

class Test {
    public static void main(String[] args) {
        String str = "This is Java class For students";
        String[] sarr = str.split(" ");

        for (String s : sarr) {
            char ch = s.charAt(0);
            if (ch >= 'A' && ch <= 'Z') {
                System.out.print(s + " ");
            }
        }
    }
}

```

## 10. Unique and Duplicate Characters

**Q) Write a Java program to find unique and duplicate characters in a string?**

Input:

- google

Output:

- unique = gole
- duplicate = og

```
class Test {
    public static void main(String[] args) {
        String str = "mississippi";
        String duplicate = "";
        String unique = "";

        for (int i = 0; i < str.length(); i++) {
            String current = Character.toString(str.charAt(i));

            if (unique.contains(current)) {
                if (!duplicate.contains(current)) {
                    duplicate += current;
                }
            } else {
                unique += current;
            }
        }

        System.out.println("Unique = " + unique);
        System.out.println("Duplicate = " + duplicate);
    }
}
```

## 11. Most Repeated Character

**Q) Write a Java program to find the most repeating character in a string?**

Input:

- ihubtalentinstitute

Output:

- t is repeating for 5 times

```
class Test {
    public static void main(String[] args) {
        String str = "ihubtalentinstitute";
        char mostRepeated = ' ';
        int maxCount = 0;
```

```

        for (int i = 0; i < str.length(); i++) {
            int count = 0;
            for (int j = 0; j < str.length(); j++) {
                if (str.charAt(i) == str.charAt(j)) {
                    count++;
                }
            }

            if (maxCount < count) {
                maxCount = count;
                mostRepeated = str.charAt(i);
            }
        }
        System.out.println(mostRepeated + " is repeating for " + maxCount +
" times");
    }
}

```

## Assignment

**Q) Write a Java program to display palindrome words from a sentence?**

Input:

- racar is madam for student

Expected Output:

- racar madam

```

class Test {
    public static void main(String[] args) {
        String sentence = "racar is madam for student";
        String[] words = sentence.split(" ");

        for (String word : words) {
            String reversedWord = "";

            for (int i = word.length() - 1; i >= 0; i--) {
                reversedWord += word.charAt(i); // Reversing the current
word

```

```

    }

    if (word.equals(reversedWord)) {
        System.out.print(word + " ");
    }
}
}
}

```

**Question:** Rohan is a kid who has just learned about creating words from alphabets. He has written some words in the notepad of his father's laptop. Now his father wants to find the longest word written by Rohan using a computer program. Write a program to find the longest string in a given list of Strings.

```

Input:
yes no number
Output:
number

```

**Answer :**

```

import java.util.Scanner;

public class LongestWordFinder {
    public static void main(String[] args) {
        // Input prompt
        System.out.println("Enter words separated by spaces:");

        // Read input from the user
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();

        // Split the input string into words
        String[] words = input.split(" ");

        // Initialize variables to find the longest word
        String longestWord = "";

        for (String word : words) {
            // Update the longestWord if the current word is longer
            if (word.length() > longestWord.length()) {

```

```

        longestWord = word;
    }
}

// Output the longest word
System.out.println("Longest word: " + longestWord);

// Close the scanner
scanner.close();
}
}

```

## StringBuilder and StringBuffer

StringBuilder is identical to StringBuffer with the following key differences:

StringBuffer	StringBuilder
Every method present in StringBuffer is synchronized.	No method present in StringBuilder is synchronized.
At a time only one thread is allow to operate on the StringBuffer object hence StringBuffer object is Thread safe.	Multiple Threads are allowed to operate simultaneously on the StringBuilder object hence StringBuilder is not Thread safe.
It increases waiting time of the Thread and hence relatively performance is low.	Threads are not required to wait and hence relatively performance is high.
Introduced in 1.0 version.	Introduced in 1.5 version.

### Note:

- Use String if content doesn't change frequently
- Use StringBuffer if content changes frequently and thread safety is required
- Use StringBuilder if content changes frequently and thread safety is not required

## StringTokenizer

StringTokenizer is a class in the java.util package used to tokenize strings irrespective of regular expressions.

**Creating a StringTokenizer object:**

```
StringTokenizer st = new StringTokenizer(String str, RegularExpression  
regEx);
```

### Key Methods:

- `public boolean hasMoreTokens()`
- `public String nextToken()`
- `public boolean hasMoreElements()`
- `public Object nextElement()`
- `public int countTokens()`

### Example 1: Counting Tokens

```
import java.util.StringTokenizer;  
class Test {  
    public static void main(String[] args) {  
        StringTokenizer st = new StringTokenizer("this is java class", "  
");  
        System.out.println(st.countTokens()); //4  
    }  
}
```

### Example 2: Using hasMoreTokens()

```
import java.util.StringTokenizer;  
class Test {  
    public static void main(String[] args) {  
        StringTokenizer st = new StringTokenizer("this is java class", "  
");  
        while(st.hasMoreTokens()) {  
            String s = st.nextToken();  
            System.out.println(s);  
        }  
    }  
}
```

### Example 3: Using hasMoreElements()

```
java  
Copy
```



```
import java.util.StringTokenizer;
class Test {
    public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("this is java class", "
");
        while(st.hasMoreElements()) {
            String s = (String)st.nextElement();
            System.out.println(s);
        }
    }
}
```

#### Example 4: Tokenizing with Delimiter

```
import java.util.StringTokenizer;
class Test {
    public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("9,99,999", ",");
        while(st.hasMoreElements()) {
            String s = (String)st.nextElement();
            System.out.println(s);
        }
    }
}
```

## Exception Handling

### What is an Exception?

An exception is an unwanted, unexpected event that disturbs the normal flow of a program. Exceptions always occur at runtime, hence they are also known as runtime events. If any exception is raised in our program and not handled, the program will terminate abnormally.

When an exception occurs, it displays:

1. Name of the exception
2. Description of the exception
3. Line number where the exception occurred

**The main objective of exception handling is to provide graceful termination.**

### Exception vs Error

## Exception

- A problem that can be solved programmatically
- Raises due to syntax errors
- Examples: `ArithmeticException`, `FileNotFoundException`, `IllegalArgumentException`

## Error

- A problem that cannot be solved programmatically
- Raises due to lack of system resources
- Examples: `OutOfMemoryError`, `StackOverflowError`, `LinkageError`

## Types of Exceptions in Java

### 1. Predefined Exceptions

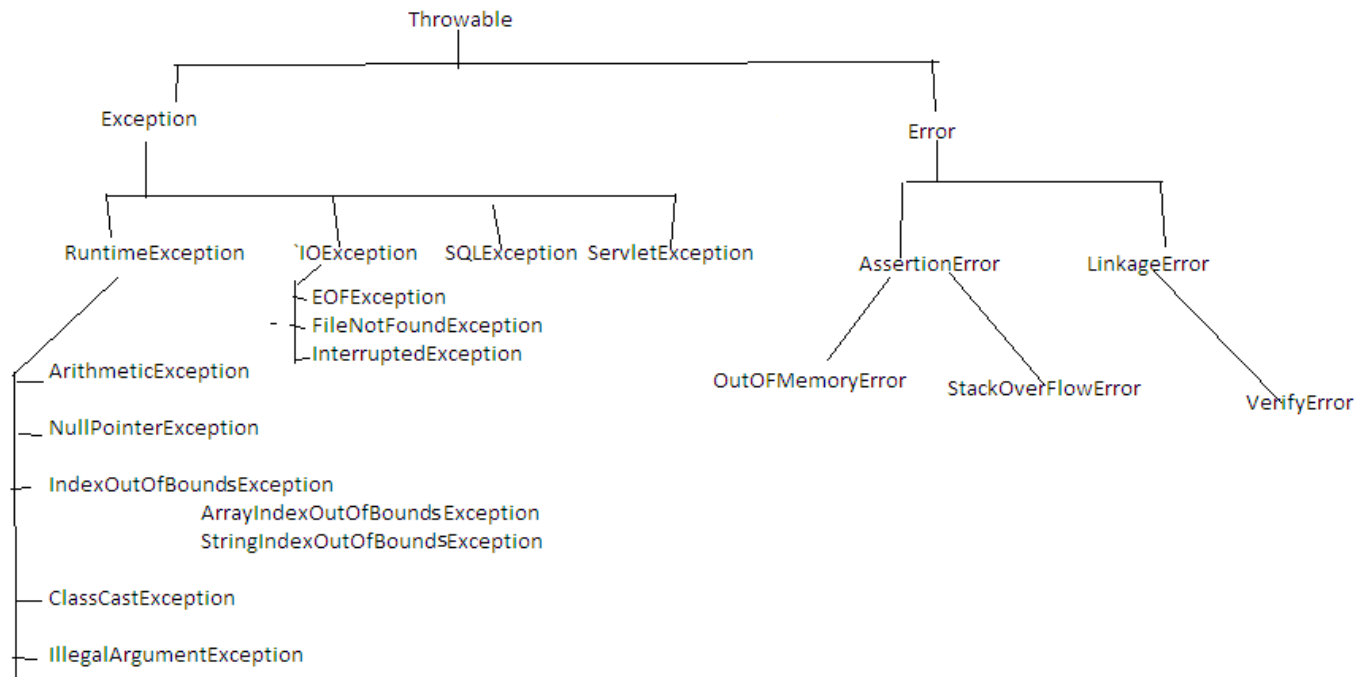
These are built-in exceptions provided by Java. They are further classified into two types:

#### i) Checked Exceptions

- Checked by the compiler at compilation time
- Must be handled explicitly using try-catch or throws
- Examples:
  - `InterruptedException`
  - `FileNotFoundException`
  - `EOFException`

#### ii) Unchecked Exceptions

- Checked by the JVM at runtime
- Handling is optional
- Examples:
  - `ArithmeticException`
  - `ClassCastException`
  - `IllegalArgumentException`



## 2. User-defined Exceptions

Custom exceptions created by the programmer

## Types of Program Termination

### 1. Smooth/Graceful Termination

- No interruption during program execution

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

### 2. Abnormal Termination

- Interruption occurs during program execution

```
class Test {
    public static void main(String[] args) {
        System.out.println(10/0);
    }
}
```

```
}  
}
```

## Exception Handling Blocks

### try block

- Contains risky code
- Associates with catch block
- Throws exceptions to catch block

### catch block

- Contains error handling code
- Catches exceptions thrown by try block
- Takes exception name as parameter

### Basic Syntax:

```
try {  
    // Risky Code  
} catch(ArithmeticException ae) {  
    // Error Handling Code  
}
```

### Example 1: Try-Catch Execution

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println("try-block");  
        } catch (Exception e) {  
            System.out.println("catch-block");  
        }  
    }  
}  
// Output: try-block
```

### Example 2: Exception in Try Block

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println("stmt1");  
            System.out.println(10/0);  
            System.out.println("stmt2");  
        } catch (Exception e) {  
            System.out.println("catch-block");  
        }  
    }  
}  
  
// Output:  
// stmt1  
// catch-block
```

### Example 3: Try with Multiple Catch Blocks

```
class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println(10/0);  
        } catch (ArithmeticException ae) {  
            System.out.println("From AE");  
        } catch (RuntimeException re) {  
            System.out.println("From RE");  
        } catch (Exception e) {  
            System.out.println("From E");  
        }  
    }  
}
```

## Displaying Exception Details

Throwable class provides three methods:

1. `printStackTrace()` - Shows name, description, and line number
2. `toString()` - Shows name and description
3. `getMessage()` - Shows only description

**Example:**

```

class Test {
    public static void main(String[] args) {
        try {
            System.out.println(10/0);
        } catch (ArithmeticException ae) {
            ae.printStackTrace();
            System.out.println("=====");
            System.out.println(ae.toString());
            System.out.println("=====");
            System.out.println(ae.getMessage());
        }
    }
}

```

## finally block

- Contains cleanup code
- Executes regardless of exception occurrence
- Always associates with try and catch blocks

### Syntax:

```

try {
    // risky code
} catch (Exception e) {
    // error handling code
} finally {
    // cleanup code
}

```

### Example with finally:

```

class Test {
    public static void main(String[] args) {
        try {
            System.out.println("try-block");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("finally-block");
        }
    }
}

```

```
}  
}
```

```
// Output:  
// try-block
```

```
// finally-block
```

In java, try with finally combination is valid.

ex:

---

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

o/p:

try-block

finally-block

## Q&A Section

**Q: What is the difference between final, finally, and finalize?**

**A:**

- **final:** A modifier for variables, methods, and classes
  - Variables: Prevents reassignment

- Methods: Prevents overriding
- Classes: Prevents inheritance
- **finally**: A block containing cleanup code that executes regardless of exceptions
- **finalize**: A method called by garbage collector before destroying an object

# Exception Handling and File I/O in Java

## throw statement

Sometimes we create exception objects explicitly and hand them over to the JVM manually using the throw statement.

Example:

java  
Copy

```
throw new ArithmeticException("Don't divide by zero");
```

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(10/0);
    }
}
```

Here exception object is created and handover to JVM by main method.

```
class Test
{
    public static void main(String[] args)
    {
        throw new ArithmeticException("Don't divide by zero");
    }
}
```

Here exception object is created and handover to JVM by using throw keyword.

## throws statement

If any checked exceptions are raised in our program, we must handle those exceptions using a try-catch block or by using the throws statement.

Example using try-catch:



java

Copy

```
class Test {  
    public static void main(String[] args) {  
        try {  
            Thread.sleep(4000);  
            System.out.println("Welcome to Java");  
        } catch (InterruptedException ie) {  
            ie.printStackTrace();  
        }  
    }  
}
```

Example using throws:

java

Copy

```
class Test {  
    public static void main(String[] args) throws InterruptedException {  
        Thread.sleep(4000);  
        System.out.println("Welcome to Java");  
    }  
}
```

## User-defined exceptions

Exceptions created by the user based on application requirements are called user-defined exceptions or custom exceptions.

Examples:

- NoInterestInJavaException
- NotPracticingException
- NeedTimePassException
- EligibleException
- NotEligibleException

Here's an example of custom exceptions:

```
import java.util.Scanner;  
  
class EligibleException extends RuntimeException {  
    EligibleException(String s) {
```

```

        super(s);
    }
}

class NotEligibleException extends RuntimeException {
    NotEligibleException(String s) {
        super(s);
    }
}

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the age:");
        int age = sc.nextInt();

        if (age < 18)
            throw new NotEligibleException("You are not eligible to vote");
        else
            throw new EligibleException("You are eligible to vote");
    }
}

```

## Class.forName()

It is used to load a class but it won't create an object.

Example:

```

class Test {
    //static block
    static {
        System.out.println("static-block");
    }
    //instance block
    {
        System.out.println("instance-block");
    }
    public static void main(String[] args) throws ClassNotFoundException {
        Class.forName("Test");
    }
}

```

# Types of Errors

We have three types of errors:

1. Compile-time error
2. Logical error
3. Runtime error

## java.io package

### File

A File object can be used to check if a file exists, create a new file, or create a directory.

Example:

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        File f = new File("abc.txt");
        System.out.println(f.exists()); // false
        f.createNewFile();
        System.out.println(f.exists()); // true

        File dir = new File("bhaskar123");
        dir.mkdir();
    }
}
```

Question: Write a Java program to create a "cricket123" folder and inside that folder create an "abc.txt" file?

Answer:

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        File f1 = new File("cricket123");
```

```

        f1.mkdir();

        File f2 = new File("cricket123", "abc.txt");
        f2.createNewFile();

        System.out.println("Please check the location");
    }
}

```

## FileWriter

FileWriter is used to write character-oriented data into a file.

Constructor:

- `FileWriter fw = new FileWriter(String s);`
- `FileWriter fw = new FileWriter(File f);`

Methods:

1. `write(int ch)`
2. `write(char[] ch)`
3. `write(String s)`
4. `flush()`
5. `close()`

Example:

```

import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("aaa.txt");

        fw.write(98); // b
        fw.write("\n");

        char[] ch = {'a', 'b', 'c'};
        fw.write(ch);
        fw.write("\n");

        fw.write("bhaskar\nsolution");
        fw.flush();
        fw.close();
    }
}

```

```

        System.out.println("Please check the location");
    }
}

```

## Try with Resources

A try-with-resources statement is a try statement that declares one or more resources. The resource is an object that must be closed after finishing the program.

Example:

```

import java.io.*;

class Test {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("aaa.txt")) {
            fw.write(98); // b
            fw.write("\n");

            char[] ch = {'a', 'b', 'c'};
            fw.write(ch);
            fw.write("\n");

            fw.write("bhaskar\nsolution");
            fw.flush();
            System.out.println("Please check the location");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## FileReader

FileReader is used to read character-oriented data from a file.

Constructor:

- `FileReader fr = new FileReader(String s);`
- `FileReader fr = new FileReader(File f);`

Methods:

1. read()
2. read(char[] ch)
3. close()

Example:

java

Copy

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("aaa.txt");

        int i = fr.read();
        while (i != -1) {
            System.out.print((char)i);
            i = fr.read();
        }
        fr.close();
    }
}
```

Example using char array:

java

Copy

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("aaa.txt");

        char[] carr = new char[255];
        fr.read(carr);

        for (char c : carr) {
            System.out.print(c);
        }

        fr.close();
    }
}
```

Note: The usage of `FileWriter` and `FileReader` is not recommended due to the need for manual line separators and character-by-character reading.

## BufferedWriter

`BufferedWriter` is used to insert character-oriented data into a file.

Constructor:

- `BufferedWriter bw = new BufferedWriter(Writer w);`
- `BufferedWriter bw = new BufferedWriter(Writer w, int buffersize);`

Methods:

1. `write(int ch)`
2. `write(char[] ch)`
3. `write(String s)`
4. `flush()`
5. `close()`
6. `newLine()`

Example:

java

Copy

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        BufferedWriter bw = new BufferedWriter(new FileWriter("bbb.txt"));
        bw.write(98); // b
        bw.newLine();

        char[] ch = {'a', 'b', 'c'};
        bw.write(ch);
        bw.newLine();

        bw.write("bhaskar");
        bw.newLine();

        bw.flush();
        bw.close();
        System.out.println("Please check the location");
    }
}
```

```
}
```

## BufferedReader

BufferedReader is an enhanced reader to read character-oriented data from a file.

Constructor:

- `BufferedReader br = new BufferedReader(Reader r);`
- `BufferedReader br = new BufferedReader(Reader r, int buffersize);`

Methods:

1. `read()`
2. `read(char[] ch)`
3. `close()`
4. `readLine()`

Example:

java

Copy

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("bbb.txt"));

        String line = br.readLine();
        while (line != null) {
            System.out.println(line);
            line = br.readLine();
        }

        br.close();
    }
}
```

```
}
```

## PrintWriter

PrintWriter is an enhanced writer to write character-oriented data into a file. It can handle various data types.

Constructor:



- `PrintWriter pw = new PrintWriter(String s);`
- `PrintWriter pw = new PrintWriter(File f);`
- `PrintWriter pw = new PrintWriter(Writer w);`

Methods:

- `write(int ch)`
- `write(char[] ch)`
- `write(String s)`
- `flush()`
- `close()`
- `println(int i)`
- `println(float f)`
- `println(double d)`
- `println(String s)`
- `println(char c)`
- `println(boolean b)`
- `print(int i)`
- `print(float f)`
- `print(double d)`
- `print(String s)`
- `print(char c)`
- `print(boolean b)`

Example:

```
import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        PrintWriter pw = new PrintWriter("ccc.txt");

        pw.write(100); // d
        pw.println(100); // 100
        pw.print('a');
        pw.println(true);
        pw.println("hi");
        pw.println(10.5d);

        pw.flush();
        pw.close();
        System.out.println("Please check the location");
    }
}
```

# Various ways to provide input values from the keyboard

There are various ways to provide input values from the keyboard:

1. Command-line argument
2. BufferedReader class
3. Console class
4. Scanner class

## 1. Command-line argument

In command-line arguments, we pass our inputs at runtime.

Example:

```
class Test {  
    public static void main(String[] args) {  
        String name = args[0];  
        System.out.println("Welcome: " + name);  
    }  
}
```

Usage:

Copy

```
javac Test.java  
java Test Alan
```

## 2. BufferedReader class

BufferedReader class is present in the java.io package.

Example:

```
import java.io.*;  
  
class Test {  
    public static void main(String[] args) throws IOException {  
        BufferedReader br = new BufferedReader(new  
        InputStreamReader(System.in));  
  
        System.out.println("Enter the Name:");
```

```

        String name = br.readLine();

        System.out.println("Welcome: " + name);
    }
}

```

### 3. Console class

Console class is present in the java.io package.

Example:

```

import java.io.*;

class Test {
    public static void main(String[] args) throws IOException {
        Console c = System.console();

        System.out.println("Enter the Name:");
        String name = c.readLine();

        System.out.println("Welcome: " + name);
    }
}

```

### 4. Scanner class

Scanner class is present in the java.util package.

Methods:

- next()
- nextLine()
- nextInt()
- nextFloat()
- nextDouble()
- next().charAt(0)

Example:

```

import java.util.*;

class Test {

```

```

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

    System.out.println("Enter the No:");
    int no = sc.nextInt();

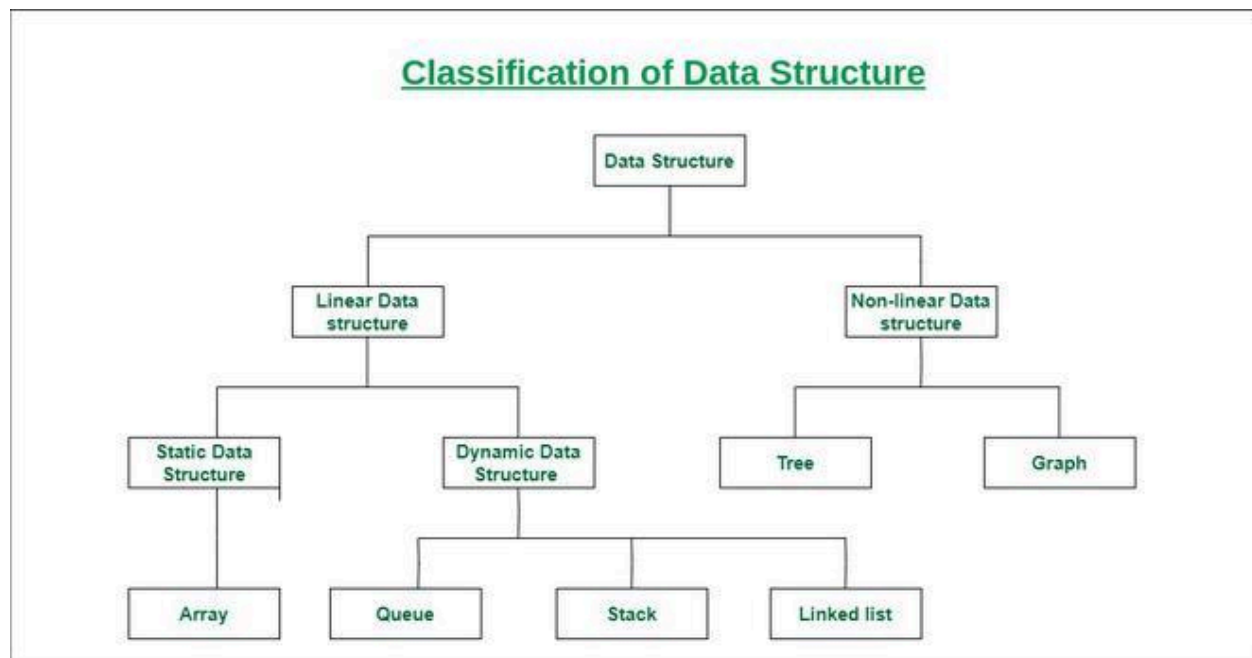
    System.out.println("Enter the Name:");
    String name = sc.next();

    System.out.println("Enter the Fee:");
    double fee = sc.nextDouble();

    System.out.println(no + " " + name + " " + fee);
}
}

```

Q) Types of data Structures in java?



## Generics

Arrays are typesafe, meaning we can guarantee the type of elements present in an array.

Example:

```
int[] arr = new int[3];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
```

If there's a requirement to store String values, it's recommended to use a String[] array:

```
String[] sarr = new String[3];
sarr[0] = "hi";
sarr[1] = "bye";
sarr[2] = 10; // invalid
```

When retrieving data from an array, typecasting is not necessary:

```
String[] sarr = new String[3];
sarr[0] = "hi";
sarr[1] = "bye";
// ...
String val = sarr[0];
```

Collections, on the other hand, are not typesafe. We can't guarantee the type of elements present in Collections. Using an ArrayList for storing String values is not recommended, as it won't generate compile-time or runtime errors, but may lead to program failures:

```
java
Copy
ArrayList al = new ArrayList();
al.add("Hi");
al.add("Bye");
al.add(10);
```

When retrieving data from Collections, typecasting is necessary:

```
java
Copy
ArrayList al = new ArrayList();
al.add("Hi");
al.add("Bye");
al.add(10);
// ...
```

```
String val = (String)a1.get(0);
```

To overcome these limitations, Sun Microsystems introduced the Generics concept in Java 1.5. The main objectives of Generics are:

1. To make Collections typesafe.
2. To avoid typecasting problems.

## java.util package

### Difference between Arrays and Collections

Arrays	Collections
Collection of homogeneous data elements	Collection of homogeneous and heterogeneous data elements
Fixed in size	Growable in nature
Better performance	Better memory usage
Can hold primitive and object types	Can hold only object types
Not implemented based on data structure concepts	Implemented based on data structure concepts

## Collection Framework

The Collection Framework defines several utility classes and interfaces to represent a group of objects in a single entity.

Example:

- C++: Standard Template Library (STL)
- Java: Collection Framework

### Collection Interface

The Collection interface is the root interface for the entire Collection framework. It's used to represent a group of individual objects in a single entity.

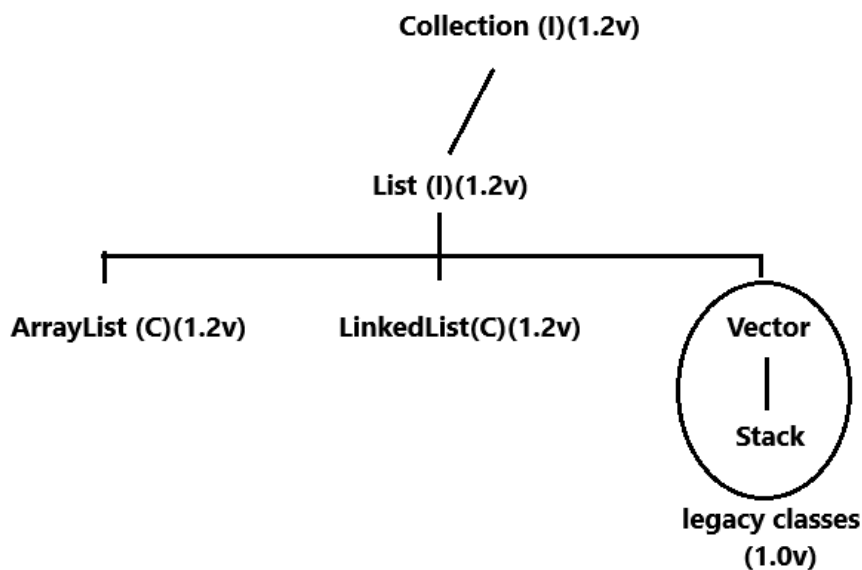
The **Iterable** interface is actually the root interface in the Collections Framework in Java.

Some methods in the Collection interface:

```
public abstract int size();  
public abstract boolean isEmpty();  
public abstract boolean contains(Object);  
public abstract boolean add(E);  
public abstract boolean remove(Object);  
// ... and more
```

## List Interface

List is a child interface of the Collection interface. It's used to represent a group of individual objects where duplicates are allowed and order is preserved.



## ArrayList

- Underlying data structure: Resizable or growable array
- Allows duplicate objects
- Preserves insertion order
- Allows heterogeneous objects
- Allows null insertion
- Implements Serializable, Cloneable, and RandomAccess interfaces

Example:

```

import java.util.*;
class Test {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("one");
        al.add("two");
        al.add("three");
        System.out.println(al); // [one, two, three]

        al.add("one");
        System.out.println(al); // [one, two, three, one]

        al.add(null);
        System.out.println(al); // [one, two, three, one, null]
    }
}

```

Example:

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        List<String> list=new ArrayList<String>();
        list.add("one");
        list.add("two");
        list.add("three");
        System.out.println(list);//[one,two,three]
    }
}

```

Example:

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        List<String> list=Arrays.asList("six","four","two");

        System.out.println(list);//[six,four,two]
    }
}

```



```
}
```

Example:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        List<Integer> list=Arrays.asList(10,20,30);

        System.out.println(list);//[10,20,30]
    }
}
```

Example:

```
public E getFirst();
public E getLast();
public E removeFirst();
public E removeLast();
public void addFirst(E);
public void addLast(E);
```

## LinkedList

- Underlying data structure: Doubly LinkedList
- Allows duplicate objects
- Preserves insertion order
- Allows heterogeneous objects
- Allows null insertion
- Implements Serializable, Cloneable, and Deque interfaces

LinkedList-specific methods:

```
public E getFirst();
public E getLast();
public E removeFirst();
public E removeLast();
public void addFirst(E);
public void addLast(E);
```

Example:

```

import java.util.*;
class Test {
    public static void main(String[] args) {
        LinkedList<String> ll = new LinkedList<String>();
        ll.add("one");
        ll.add("two");
        ll.add("three");
        System.out.println(ll); // [one, two, three]

        ll.addFirst("gogo");
        ll.addLast("jojo");
        System.out.println(ll); // [gogo, one, two, three, jojo]

        System.out.println(ll.getFirst()); // gogo
        System.out.println(ll.getLast()); // jojo

        ll.removeFirst();
        ll.removeLast();
        System.out.println(ll); // [one, two, three]
    }
}

```

Example:

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> ll=new LinkedList<String>();
        ll.add("one");
        ll.add("two");
        ll.add("three");
        System.out.println(ll);//[one,two,three]

        ll.addFirst("gogo");
        ll.addLast("jojo");
        System.out.println(ll);//[gogo,one,two,three,jojo]

        System.out.println(ll.getFirst());//gogo
        System.out.println(ll.getLast());//jojo

        ll.removeFirst();
    }
}

```

```

        ll.removeLast();
        System.out.println(ll);//[one,two,three]
    }
}

```

Example:

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        LinkedList<String> ll1=new LinkedList<String>();
        ll1.add("one");
        ll1.add("two");
        ll1.add("three");
        System.out.println(ll1);//[one,two,three]

        LinkedList<String> ll2=new LinkedList<String>();
        ll2.add("raja");
        System.out.println(ll2);//[raja]

        ll2.addAll(ll1);
        System.out.println(ll2);//[raja,one,two,three]

        System.out.println(ll2.containsAll(ll1));//true

        ll2.removeAll(ll1);
        System.out.println(ll2);//[raja]
    }
}

```

## Difference between ArrayList and LinkedList

ArrayList	LinkedList
Underlying data structure is resizable array	Underlying data structure is doubly linked list
Better for accessing and storing data	Better for manipulating data
Contiguous memory location for elements	Non-contiguous memory location for elements
Default capacity of 10 when initialized	No default capacity

## Vector

- Underlying data structure: Resizable or growable array
- Allows duplicate objects
- Preserves insertion order
- Allows heterogeneous objects
- Allows null insertion
- All methods are synchronized

Vector-specific methods:

```
addElement()  
removeElementAt()  
removeAllElements()  
firstElement()  
lastElement()
```

Example:

```
import java.util.*;  
class Test {  
    public static void main(String[] args) {  
        Vector<Integer> v = new Vector<Integer>();  
        System.out.println(v.capacity());  
  
        for(int i = 1; i <= 10; i++) {  
            v.addElement(i);  
        }  
        System.out.println(v); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
        System.out.println(v.firstElement()); // 1  
        System.out.println(v.lastElement()); // 10  
  
        v.removeElementAt(5);  
        System.out.println(v); // [1, 2, 3, 4, 5, 7, 8, 9, 10]  
  
        v.removeAllElements();  
        System.out.println(v); // []  
    }  
}
```

## Difference between ArrayList and Vector

ArrayList	Vector
No method is synchronized	All methods are synchronized
Not thread-safe	Thread-safe
Higher performance (threads don't wait)	Lower performance (threads wait)
Introduced in Java 1.2	Introduced in Java 1.0

## Stack

Stack is a child class of Vector. It follows the Last-In-First-Out (LIFO) principle.

Constructor:

java  
Copy

```
Stack s = new Stack();
```

Methods:

- `push(E)`: Pushes an element onto the stack
- `pop()`: Removes and returns the top element from the stack
- `peek()`: Returns the top element without removing it
- `isEmpty()`: Checks if the stack is empty
- `search(Object o)`: Returns the position of an element in the stack

Example:

java  
Copy

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        Stack<String> s = new Stack<String>();
        s.push("A");
        s.push("B");
        s.push("C");
        System.out.println(s); // [A, B, C]

        s.pop();
        System.out.println(s); // [A, B]
```

```

        System.out.println(s.peek()); // B

        System.out.println(s.isEmpty()); // false

        System.out.println(s.search("Z")); // -1

        System.out.println(s.search("A")); // 2
    }
}

```

**Q) Write a java program to check given string is balanced or not?**

Input:

{[()]}

Output:

It is a balanced string

Example:

```

import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        String str="{[()]}";
        if(isBalanced(str))
            System.out.println("It is a balanced string");
        else
            System.out.println("It is not a balanced string");
    }
    //callie method
    public static boolean isBalanced(String str)
    {
        Stack<Character> s=new Stack<Character>();

        for(int i=0;i<str.length();i++)
        {
            char ch=str.charAt(i);

```

```

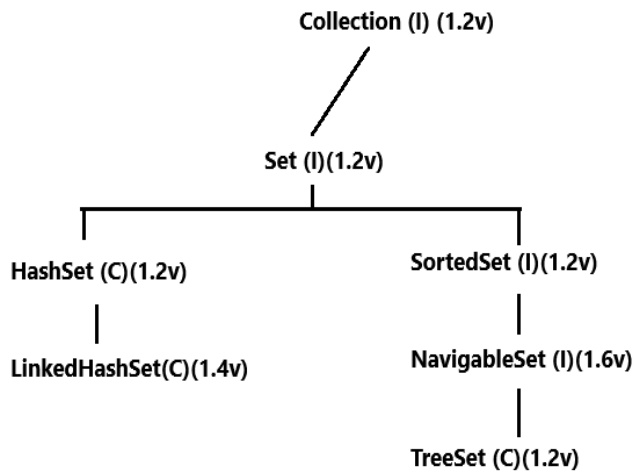
        if(ch=='{' || ch=='[' || ch=='(')
        {
            s.push(ch);
        }
        else if(ch==')' && !s.isEmpty() && s.peek()=='(')
        {
            s.pop();
        }
        else if(ch==']' && !s.isEmpty() && s.peek()=='[')
        {
            s.pop();
        }
        else if(ch=='}' && !s.isEmpty() && s.peek()=='{')
        {
            s.pop();
        }
        else
        {
            return false;
        }
    }

    return s.isEmpty();
}

```

## Set Interface

Set is a child interface of the Collection interface. It's used to represent a group of individual objects where duplicate values are not allowed and order is not preserved.



## HashSet

- Underlying data structure: Hashtable
- Does not allow duplicate objects
- Does not preserve insertion order
- Allows heterogeneous objects
- Allows null insertion

Example:

java

Copy

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        HashSet hs = new HashSet();
        hs.add("nine");
        hs.add("one");
        hs.add("six");
        System.out.println(hs); // [nine, six, one]

        hs.add("one");
        System.out.println(hs); // [nine, six, one]

        hs.add(10);
        System.out.println(hs); // [nine, six, one, 10]
```



```

        hs.add(null);
        System.out.println(hs); // [null, nine, six, one, 10]
    }
}

```

## LinkedHashSet

LinkedHashSet is a child class of HashSet. It's similar to HashSet with the following differences:

HashSet	LinkedHashSet
Underlying data structure is Hashtable	Underlying data structure is Hashtable and LinkedList
Does not preserve insertion order	Preserves insertion order
Introduced in Java 1.2	Introduced in Java 1.4

Example:

```

import java.util.*;
class Test {
    public static void main(String[] args) {
        LinkedHashSet lhs = new LinkedHashSet();
        lhs.add("nine");
        lhs.add("one");
        lhs.add("six");
        System.out.println(lhs); // [nine, one, six]

        lhs.add("one");
        System.out.println(lhs); // [nine, one, six]

        lhs.add(10);
        System.out.println(lhs); // [nine, one, six, 10]

        lhs.add(null);
        System.out.println(lhs); // [nine, one, six, 10, null]
    }
}

```

## Java Program to Display Distinct Elements from an Array

### Question

Write a Java program to display distinct elements from a given array.

#### Input:

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

#### Output:

```
1 2 3 4
```

#### Solution:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        int[] arr = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4};

        Set<Integer> set = new LinkedHashSet<Integer>();

        for (int i : arr) {
            set.add(i);
        }

        for (int i : set) {
            System.out.print(i + " ");
        }
    }
}
```

## TreeSet

The underlying data structure for TreeSet is a Balanced Tree.

Key characteristics:

- Duplicate objects are not allowed.
- Insertion order is not preserved; it uses the sorting order based on hash code.

- Heterogeneous objects are not allowed.
- Attempting to insert heterogeneous objects will result in a `ClassCastException`.
- Null insertion is not possible and will throw a `NullPointerException`.

## Example 1:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        TreeSet ts = new TreeSet();
        ts.add(10);
        ts.add(5);
        ts.add(1);
        ts.add(3);
        System.out.println(ts); // Output: [1, 3, 5, 10]

        ts.add(5);
        System.out.println(ts); // Output: [1, 3, 5, 10]

        // Uncommenting the following lines will result in exceptions:
        // ts.add("hi");
        // System.out.println(ts); // Runtime Exception: ClassCastException

        // ts.add(null);
        // System.out.println(ts); // Runtime Exception: NullPointerException
    }
}
```

# Comparable vs Comparator Interface

## Comparable

- Present in the `java.lang` package.
- Contains only one method: `compareTo()`.
- Used for default natural sorting order.

### Example:

```
class Test {
```

```

    public static void main(String[] args) {
        System.out.println("A".compareTo("Z")); // Output: -25
        System.out.println("Z".compareTo("A")); // Output: 25
        System.out.println("K".compareTo("K")); // Output: 0
    }
}

```

## Comparator

- Present in the java.util package.
- Contains two methods: compare() and equals().
- Used for customized sorting order.

### Example:

```

import java.util.*;

class Test {
    public static void main(String[] args) {
        TreeSet<Integer> ts = new TreeSet<Integer>(new MyComparator());
        ts.add(10);
        ts.add(1);
        ts.add(5);
        ts.add(3);
        System.out.println(ts); // Output: [10, 5, 3, 1]
    }
}

class MyComparator implements Comparator<Integer> {
    public int compare(Integer i1, Integer i2) {
        if (i1 < i2)
            return 1;
        else if (i1 > i2)
            return -1;
        else
            return 0;
    }
}

```

**Question: Write a Java program to compare two dates**

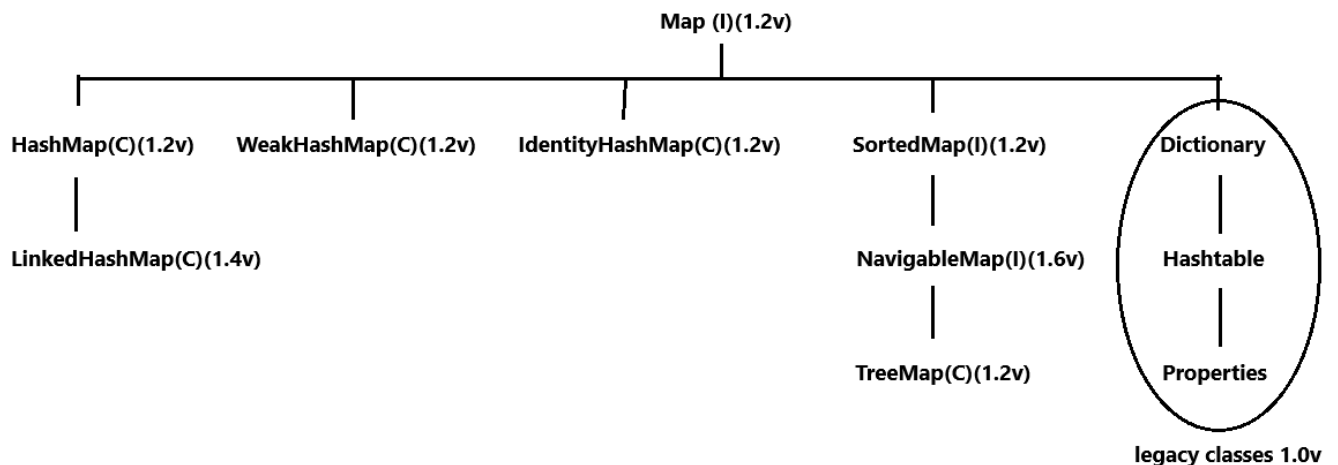
```
import java.time.LocalDate;

class Test {
    public static void main(String[] args) {
        LocalDate date1 = LocalDate.now();
        LocalDate date2 = LocalDate.of(2024, 9, 15);

        if (date1.compareTo(date2) > 0)
            System.out.println("Date1 is greater");
        else if (date1.compareTo(date2) < 0)
            System.out.println("Date2 is greater");
        else
            System.out.println("Both dates are the same");
    }
}
```

## Map

- Not a child interface of the Collection interface.
- Used to represent a group of individual objects as key-value pairs.
- Keys cannot be duplicate, but values can be.
- Both key and value must be objects.
- Each key-value pair is called an entry.



## HashMap

- The underlying data structure is a Hashtable.

- Duplicate keys are not allowed, but values can be duplicate.
- Insertion order is not preserved.
- Heterogeneous objects are allowed for both keys and values.
- Null insertion is possible for both keys and values.

### Example:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        hm.put(1, 10);
        hm.put(5, 50);
        hm.put(10, 100);
        hm.put(7, 70);
        System.out.println(hm); // Output: {1=10, 5=50, 7=70, 10=100}

        hm.put(5, 500);
        System.out.println(hm); // Output: {1=10, 5=500, 7=70, 10=100}

        hm.put("one", "raja");
        System.out.println(hm); // Output: {1=10, 5=500, 7=70, one=raja,
10=100}

        hm.put(null, null);
        System.out.println(hm); // Output: {null=null, 1=10, 5=500, 7=70,
one=raja, 10=100}
    }
}
```

## LinkedHashMap

LinkedHashMap is similar to HashMap with the following differences:

HashMap	LinkedHashMap
Underlying data structure is Hashtable	Underlying data structure is Hashtable and LinkedList
Insertion order is not preserved	Insertion order is preserved
Introduced in Java 1.2	Introduced in Java 1.4

## Example:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        LinkedHashMap lhm = new LinkedHashMap();
        lhm.put("ten", "gogo");
        lhm.put("one", "alan");
        lhm.put("six", "raja");
        lhm.put("four", "morries");
        System.out.println(lhm); // Output: {ten=gogo, one=alan, six=raja,
four=morries}

        lhm.put("one", "jojo");
        System.out.println(lhm); // Output: {ten=gogo, one=jojo, six=raja,
four=morries}

        lhm.put(10, 100);
        System.out.println(lhm); // Output: {ten=gogo, one=jojo, six=raja,
four=morries, 10=100}

        lhm.put(null, null);
        System.out.println(lhm); // Output: {ten=gogo, one=jojo, six=raja,
four=morries, 10=100, null=null}
    }
}
```

# Interview Program

## 1. TreeMap

The TreeMap data structure in Java is implemented using a RED-BLACK TREE.

### Key Features:

- Keys must be unique, but values can be duplicated
- Elements are automatically sorted by key (insertion order not preserved)
- For default natural sorting: Keys must be homogeneous
- For custom sorting: Keys can be heterogeneous and non-comparable
- Null keys are not allowed, but values can be null

### Example:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        TreeMap<Integer, String> tm = new TreeMap<Integer, String>();

        tm.put(10, "ten");
        tm.put(1, "one");
        tm.put(5, "five");
        System.out.println(tm); // Output: {1=one, 5=five, 10=ten}

        tm.put(1, "gogo"); // Updating existing key
        System.out.println(tm); // Output: {1=gogo, 5=five, 10=ten}

        tm.put(6, null); // Null value is allowed
        System.out.println(tm); // Output: {1=gogo, 5=five, 6=null, 10=ten}

        // tm.put(null, "seven"); // Will throw NullPointerException
    }
}
```

## 2. Hashtable

A thread-safe implementation of a hash table.

### Key Features:

- Uses hashtable as underlying data structure
- Keys must be unique, values can be duplicated
- Insertion order not preserved (descending order of key)
- Supports heterogeneous objects for both keys and values
- Null values not allowed for either keys or values

### Example:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        Hashtable ht = new Hashtable();
    }
}
```



```

    ht.put(1, "one");
    ht.put(10, "ten");
    ht.put(5, "five");
    System.out.println(ht); // Output: {10=ten, 5=five, 1=one}

    ht.put(1, "gogo");      // Updating existing key
    System.out.println(ht); // Output: {10=ten, 5=five, 1=gogo}

    ht.put("six", 6);        // Heterogeneous keys allowed
    System.out.println(ht); // Output: {10=ten, six=6, 5=five, 1=gogo}

    // ht.put(4, null);      // Will throw NullPointerException
    // ht.put(null, "four"); // Will throw NullPointerException
}
}

```

### 3. Interview Questions

#### Q1: Find Lucky Number in Array

A number is considered lucky if its frequency in the array equals its value.

Input: [1, 2, 2, 3, 3, 3]

Output: 3 (because 3 appears 3 times)

```

import java.util.*;

class Test {
    public static void main(String[] args) {
        int[] arr = {1, 2, 2, 3, 3, 3}; // Input array

        Map<Integer, Integer> map = new LinkedHashMap<Integer, Integer>();

        // Count frequencies
        for(int i = 0; i < arr.length; i++) {
            if(map.containsKey(arr[i])) {
                map.put(arr[i], map.get(arr[i]) + 1);
            } else {
                map.put(arr[i], 1);
            }
        }
    }
}

```

```

    }

    int x = 0;
    int max = -1;

    // Find lucky number
    for(Map.Entry<Integer, Integer> entry: map.entrySet()) {
        if(entry.getKey() == entry.getValue()) {
            x = entry.getValue();
            max = Math.max(max, x);
        }
    }

    System.out.println(max); // Output: 3
}
}

```

## Q2: Find Word Occurrences in String

Input: "this is is java java class"

Output: this=1 is=2 java=2 class=1

```

import java.util.*;

class Test {
    public static void main(String[] args) {
        String str = "this is is java java class"; // Input string
        String[] sarr = str.split(" ");

        Map<String, Integer> map = new LinkedHashMap<String, Integer>();

        for(String s: sarr) {
            if(map.get(s) != null) {
                map.put(s, map.get(s) + 1);
            } else {
                map.put(s, 1);
            }
        }

        // Output: this=1 is=2 java=2 class=1
        map.forEach((key, value) -> System.out.print(key + "=" + value + "
    
```

```
"));
    }
}
```

### Q3: Find Character Occurrences in String

Input: "java"

Output: j=1 a=2 v=1

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        String str = "java"; // Input string
        char[] carr = str.toCharArray();

        Map<Character, Integer> map = new LinkedHashMap<Character,
Integer>();

        for(char c: carr) {
            if(map.get(c) != null) {
                map.put(c, map.get(c) + 1);
            } else {
                map.put(c, 1);
            }
        }

        // Output: j=1 a=2 v=1
        map.forEach((key, value) -> System.out.print(key + "=" + value + "
"));
    }
}
```

### Q4: Collection vs Collections Comparison

Feature	Collection	Collections
Type	Root interface	Utility class
Purpose	Represents group of objects	Provides utility methods
Methods	Abstract, static, and default	Only static methods

Usage	Base for collection classes	Helper operations on collections
-------	-----------------------------	----------------------------------

## Q5: Making ArrayList Synchronized

Input: Creating ArrayList with elements ["java", "oracle", "html", "reactjs"]

Output: Synchronized version of the ArrayList

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("java");
        al.add("oracle");
        al.add("html");
        al.add("reactjs");
        System.out.println("Original ArrayList: " + al);
        // Output: [java, oracle, html, reactjs]

        List<String> synchronizedList = Collections.synchronizedList(al);

        System.out.print("Synchronized List: ");
        for(String s: synchronizedList) {
            System.out.print(s + " ");
        }
        // Output: java oracle html reactjs
    }
}
```

Note: The synchronized list provides thread-safe operations on the ArrayList. All methods of the synchronized list are thread-safe, but iteration must be manually synchronized by the user when needed.

This rewritten version includes:

- Clear input/output examples for each program
- Better formatting and organization
- Detailed explanations
- Complete code samples with comments
- Clear separation between different topics
- Proper markdown formatting for better readability

# Java String Sorting and Collections Examples

## 1. String Sorting Program

**Question:** Write a Java program to sort strings

**Input:** dog apple cat bat **Output:** apple bat cat dog

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        String str = "dog apple cat bat";
        String[] sarr = str.split(" ");

        List<String> list = new ArrayList<String>();
        for(String s : sarr) {
            list.add(s);
        }

        Collections.sort(list);

        for(String s : list) {
            System.out.print(s + " ");
        }
    }
}
```

## 2. Employee Information Storage

**Question:** Write a Java program to store employees information in a collection

```
import java.util.*;

class Employee {
    private int empId;
    private String empName;
    private double empSal;
```

```

Employee(int empId, String empName, double empSal) {
    this.empId = empId;
    this.empName = empName;
    this.empSal = empSal;
}

// Getter methods
public int getEmpId() {
    return empId;
}

public String getEmpName() {
    return empName;
}

public double getEmpSal() {
    return empSal;
}

public String toString() {
    return getEmpId() + "," + getEmpName() + "," + getEmpSal();
}
}

class Test {
    public static void main(String[] args) {
        List<Employee> list = new ArrayList<Employee>();
        list.add(new Employee(101, "Alan Morries", 1000d));
        list.add(new Employee(102, "Jose Redrequiz", 2000d));
        list.add(new Employee(103, "Ana Julie", 3000d));
        list.add(new Employee(104, "Erick Anderson", 4000d));

        for(Employee e : list) {
            System.out.println(e);
        }
    }
}

```

## Types of Cursors in Java

Cursors are used to retrieve objects one by one from Collections. There are three types:

## 1. Enumeration

- Used to read objects one by one from legacy Collection objects
- Methods:
  - `public boolean hasMoreElements()`
  - `public Object nextElement()`

Example:

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        for(int i = 1; i <= 10; i++) {
            v.add(i);
        }

        Enumeration e = v.elements();
        while(e.hasMoreElements()) {
            Integer i = (Integer)e.nextElement();
            System.out.println(i);
        }
    }
}
```

### Limitations of Enumeration:

1. Only works with legacy Collection objects
2. Can only perform read operations, not remove operations
3. Not a universal cursor

## 2. Iterator

- Universal cursor that works with any Collection object
- Supports both read and remove operations
- Methods:
  - `public boolean hasNext()`
  - `public Object next()`
  - `public void remove()`

Example:

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

```

class Test {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for(int i = 1; i <= 10; i++) {
            al.add(i);
        }

        Iterator itr = al.iterator();
        while(itr.hasNext()) {
            Integer i = (Integer)itr.next();
            if(i % 2 == 0)
                System.out.println(i + " ");
            else
                itr.remove();
        }

        System.out.println(al);
    }
}

```

#### Limitations of Iterator:

1. Only supports forward direction traversal
2. Cannot add or replace elements

### 3. ListIterator

- Child interface of Iterator
- Works with List Collection objects
- Supports bi-directional traversal
- Can perform read, remove, add, and replace operations
- Methods:

- ◆ `public boolean hasNext()`
- ◆ `public Object next()`
- ◆ `public boolean hasPrevious()`
- ◆ `public Object previous()`
- ◆ `public int nextIndex()`
- ◆ `public int previousIndex()`
- ◆ `public void remove()`
- ◆ `public void set(Object o)`
- ◆ `public void add(Object o)`



Examples:

## 1. Basic ListIterator Usage:

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("bala");
        al.add("nag");
        al.add("chiru");
        al.add("venki");
        System.out.println(al); //[bala,nag,chiru,venki]

        ListIterator litr = al.listIterator();
        while(litr.hasNext()) {
            String s = (String)litr.next();
            System.out.println(s);
        }
    }
}
```

## 2. Remove Element Example:

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("bala");
        al.add("nag");
        al.add("chiru");
        al.add("venki");

        ListIterator litr = al.listIterator();
        while(litr.hasNext()) {
            String s = (String)litr.next();
            if(s.equals("chiru")) {
                litr.remove();
            }
        }

        System.out.println(al); //[bala,nag,venki]
```

```
}  
}
```

### 3. Add Element Example:

```
import java.util.*;  
class Test {  
    public static void main(String[] args) {  
        ArrayList al = new ArrayList();  
        al.add("bala");  
        al.add("nag");  
        al.add("chiru");  
        al.add("venki");  
  
        ListIterator litr = al.listIterator();  
        while(litr.hasNext()) {  
            String s = (String)litr.next();  
            if(s.equals("chiru")) {  
                litr.add("pavan");  
            }  
        }  
  
        System.out.println(al); //[bala,nag,chiru,pavan,venki]  
    }  
}
```

### 4. Replace Element Example:

```
import java.util.*;  
class Test {  
    public static void main(String[] args) {  
        ArrayList al = new ArrayList();  
        al.add("bala");  
        al.add("nag");  
        al.add("jagan");  
        al.add("venki");  
  
        ListIterator litr = al.listIterator();  
        while(litr.hasNext()) {  
            String s = (String)litr.next();  
            if(s.equals("jagan")) {  
                litr.set("pavan");  
            }  
        }  
    }  
}
```

```
    }  
    }  
  
    System.out.println(al); //[bala,nag,pavan,venki]  
    }  
}
```

# Multithreading in Java

## Thread vs Process

### Thread

- A thread is a lightweight subprocess
- Multiple threads can run concurrently
- Threads can communicate with each other
- Examples:
  - A class is a thread
  - A block is a thread
  - A constructor is a thread

### Process

- A process is a collection of threads
- Multiple processes can run concurrently
- Processes cannot communicate with each other
- Examples:
  - Downloading a file from the internet
  - Taking a class via Zoom meeting
  - Typing notes in an editor

## Multitasking

Multitasking is the execution of several tasks simultaneously. It is divided into two types:

1. **Thread-based Multitasking**
  - Executes several tasks simultaneously where each task is part of the same program
  - Best suited for programmatic level operations

## 2. Process-based Multitasking

- Executes several tasks simultaneously where each task is an independent process
- Best suited for OS level operations

# Multithreading

Multithreading is the concept of executing several threads simultaneously. In Java multithreading:

- 10% of work is done by the programmer
- 90% of work is handled by the Java API

## Main Applications

1. Implementing multimedia graphics
2. Developing video games
3. Creating animations

# Creating Threads in Java

There are two ways to create a thread:

## 1. By Extending Thread Class

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Child-Thread");
        }
    }
}

class Test {
    public static void main(String[] args) {
        // Instantiate a thread
        MyThread t = new MyThread();

        // Start a thread
        t.start();

        for(int i = 1; i <= 5; i++) {
            System.out.println("Parent-Thread");
        }
    }
}
```

```
    }  
  }  
}
```

## Important Cases:

### Case 1: Thread Scheduler

- The thread scheduler decides which thread to execute when multiple threads are waiting
- The algorithm and behavior depend on the JVM vendor
- Output order cannot be predicted in multithreading

### Case 2: `t.start()` vs `t.run()`

- `t.start()`: Creates a new thread that automatically executes the `run()` method
- `t.run()`: Executes like a normal method without creating a new thread

Example using `t.run()`:

```
class MyThread extends Thread {  
    public void run() {  
        for(int i = 1; i <= 5; i++) {  
            System.out.println("Child-Thread");  
        }  
    }  
}  
  
class Test {  
    public static void main(String[] args) {  
        MyThread t = new MyThread();  
        t.run(); // No new thread created  
  
        for(int i = 1; i <= 5; i++) {  
            System.out.println("Parent-Thread");  
        }  
    }  
}
```

### Case 3: Not Overriding `run()`

- If run() is not overridden, Thread class's empty run() method is executed
- No output from child thread will be produced

## Case 4: Overloading run()

```
class MyThread extends Thread {
    public void run() {
        System.out.println("0-arg run");
    }

    public void run(int i) {
        System.out.println("int-arg run");
    }
}
```

## case5: Thread life cycle

- start() always executes the no-argument run() method

## 2. By Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("child-thread");
        }
    }
}

class Test {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r); // r is a target Runnable
        t.start();

        for(int i = 1; i <= 5; i++) {
            System.out.println("parent-thread");
        }
    }
}
```

## Thread Names

- Every thread has a name (user-provided or JVM-generated)
- Methods available:
  - `setName(String name)`
  - `getName()`

Example:

```
class Test {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()); // main

        MyThread t = new MyThread();
        System.out.println(t.getName()); // Thread-0

        Thread.currentThread().setName("Parent-Thread");
        System.out.println(Thread.currentThread().getName()); //
Parent-Thread

        t.setName("Child-Thread");
        System.out.println(t.getName()); // Child-Thread
    }
}
```

## Thread Priority

- Priority range: 1 (lowest) to 10 (highest)
- Standard constants:
  - `Thread.MAX_PRIORITY`: 10
  - `Thread.NORM_PRIORITY`: 5
  - `Thread.MIN_PRIORITY`: 1
- Methods:
  - `setPriority(int priority)`
  - `getPriority()`

Example:

```
class Test {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getPriority()); // 5
    }
}
```

```

    MyThread t = new MyThread();
    System.out.println(t.getPriority()); // 5

    Thread.currentThread().setPriority(9);
    System.out.println(Thread.currentThread().getPriority()); // 9

    t.setPriority(4);
    System.out.println(t.getPriority()); // 4

    // t.setPriority(11); // Throws IllegalArgumentException
}
}

```

Note: Setting priority > 10 will throw IllegalArgumentException

# Daemon Threads

## Definition and Characteristics

- A daemon thread is a service provider thread that supports user threads
- Common examples include garbage collector and finalizer threads
- Daemon threads terminate automatically when all user threads die

## Key Methods

- `setDaemon(true)`: Creates a daemon thread
- `isDaemon()`: Checks if a thread is a daemon thread

## Example Implementation

```

class MyThread extends Thread {
    public void run() {
        for(int i=1; i<=5; i++) {
            System.out.println(Thread.currentThread().isDaemon());
            System.out.println("Child-Thread");
        }
    }
}
}

```



```

class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.setDaemon(true); // Setting thread as daemon
        t.start();

        for(int i=1; i<=5; i++) {
            System.out.println("Parent-Thread");
        }
    }
}

```

# Thread Execution Control

There are three primary methods to control thread execution:

## 1. yield()

### Characteristics

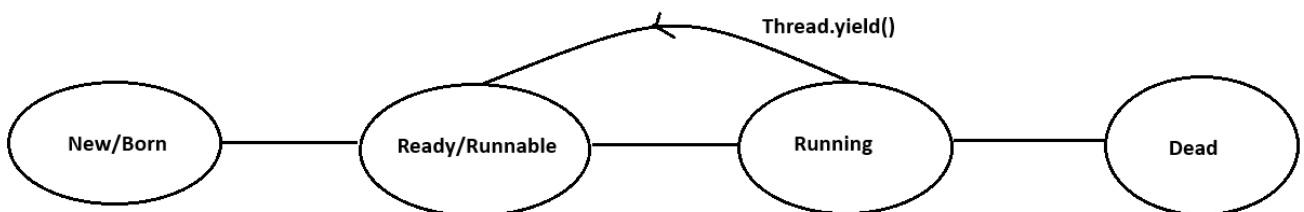
- Temporarily pauses current thread execution
- Gives execution opportunity to other threads with same priority
- If no waiting threads or only lower priority threads exist, same thread continues
- Execution order is unpredictable for same priority threads
- Thread resumption depends on thread scheduler

### Method Signature

```

public static native void yield();

```



### Example

```

class MyThread extends Thread {
    public void run() {
        for(int i=1; i<=5; i++) {
            Thread.currentThread().yield();
            System.out.println("Child-Thread");
        }
    }
}

class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        for(int i=1; i<=5; i++) {
            System.out.println("Parent-Thread");
        }
    }
}

```

Output :

```

Parent-Thread
Parent-Thread
Parent-Thread
Parent-Thread
Parent-Thread
Child-Thread
Child-Thread
Child-Thread
Child-Thread
Child-Thread

```

## 2. join()

### Characteristics

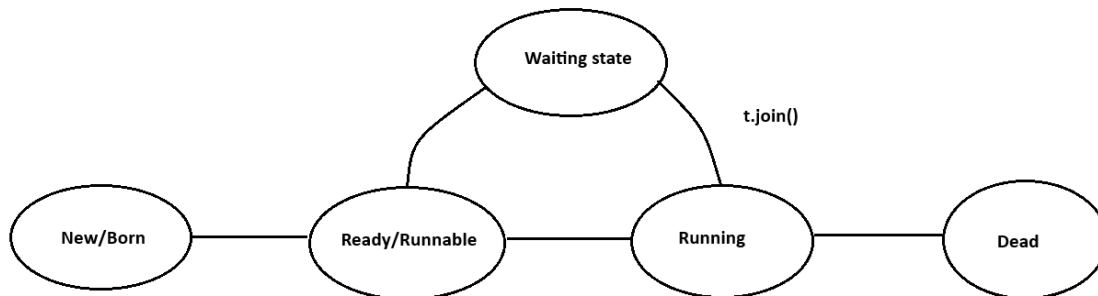
- Makes current thread wait for other thread's completion
- Throws InterruptedException (checked exception)
- Must be handled using try-catch or throws

### Method Signatures

```

public final void join() throws InterruptedException
public final void join(long ms) throws InterruptedException
public final void join(long ms, int ns) throws InterruptedException

```



## Example

```

class MyThread extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("Child-Thread");
        }
    }
}
class Test
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t=new MyThread();
        t.start();
        t.join();
        for(int i=1;i<=5;i++)
        {
            System.out.println("Parent-Thread");
        }
    }
}

```

```

public static native void yield();

```

Output:

```
Child-Thread
Child-Thread
Child-Thread
Child-Thread
Child-Thread
Parent-Thread
Parent-Thread
Parent-Thread
Parent-Thread
Parent-Thread
```

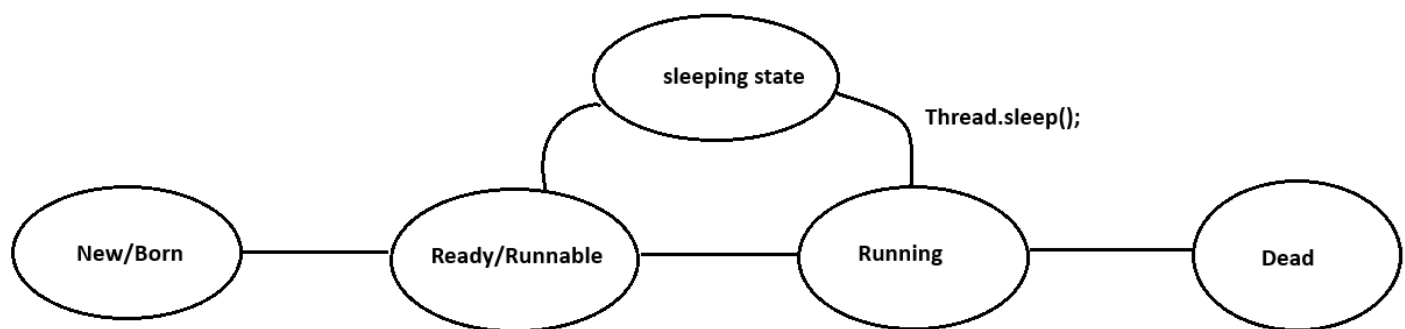
### 3. sleep()

#### Characteristics

- Pauses thread execution for specified duration
- Throws InterruptedException (checked exception)
- Must be handled using try-catch or throws

#### Method Signatures

```
public static native void sleep() throws InterruptedException
public static native void sleep(long ms) throws InterruptedException
public static native void sleep(long ms,int ns) throws InterruptedException
```



#### Example

```
class MyThread extends Thread {
    public void run() {
        for(int i=1; i<=5; i++) {
```

```

        System.out.println("Child-Thread");
        try {
            Thread.sleep(2000); // Pause for 2 seconds
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}

class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        for(int i=1; i<=5; i++) {
            System.out.println("Parent-Thread");
        }
    }
}

```

Output:

```

Parent-Thread
Parent-Thread
Parent-Thread
Child-Thread
Child-Thread
Child-Thread
Child-Thread
Child-Thread

```

# Synchronization

In Java, synchronization is a mechanism that ensures that only one thread can access a shared resource at a time, preventing concurrent access issues.

## Problems Without Synchronization

Common issues that arise without proper synchronization:

1. Data inconsistency

## 2. Thread interference

### Example Without Synchronization

```
class Table {
    void printTable(int n) {
        for(int i=1; i<=n; i++) {
            System.out.println(n*i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }
}

class MyThread1 extends Thread {
    Table t;
    MyThread1(Table t) {
        this.t = t;
    }
    public void run() {
        t.printTable(5);
    }
}

class MyThread2 extends Thread {
    Table t;
    MyThread2(Table t) {
        this.t = t;
    }
    public void run() {
        t.printTable(10);
    }
}

class Test {
    public static void main(String[] args) {
        Table obj = new Table();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
    }
}
```

```
        t1.start();
        t2.start();
    }
}
```

## Synchronization Implementation

### Key Points

- Applicable to methods and blocks
- Ensures thread safety by allowing only one thread to execute on given object
- Uses internal lock mechanism
- Thread must acquire object lock before executing synchronized method
- Other threads can't execute synchronized methods concurrently on same object
- Non-synchronized methods can still execute concurrently

### Advantages and Disadvantages

- ✓ Solves data inconsistency problems
- ✗ Increases thread waiting time
- ✗ May reduce system performance
- Use only when specifically required

### Example with Synchronized Method

```
class Table {
    synchronized void printTable(int n) {
        for(int i=1; i<=5; i++) {
            System.out.println(n*i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }
}

class MyThread1 extends Thread {
    Table t;
    MyThread1(Table t) {
        this.t = t;
    }
}
```

```

    }
    public void run() {
        t.printTable(5);
    }
}

class MyThread2 extends Thread {
    Table t;
    MyThread2(Table t) {
        this.t = t;
    }
    public void run() {
        t.printTable(10);
    }
}

class Test {
    public static void main(String[] args) {
        Table obj = new Table();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
        t2.start();
    }
}

```

## Synchronized Block

### Key Points

- Used for synchronizing specific code sections
- More efficient than synchronizing entire method
- Reduces lock duration to critical section only

### Example

```

class Table {
    void printTable(int n) {
        synchronized(this) { // Synchronized block
            for(int i=1; i<=5; i++) {
                System.out.println(n*i);
            }
        }
    }
}

```



```

        try {
            Thread.sleep(2000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}

class MyThread1 extends Thread {
    Table t;
    MyThread1(Table t) {
        this.t = t;
    }
    public void run() {
        t.printTable(5);
    }
}

class MyThread2 extends Thread {
    Table t;
    MyThread2(Table t) {
        this.t = t;
    }
    public void run() {
        t.printTable(10);
    }
}

class Test {
    public static void main(String[] args) {
        Table obj = new Table();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

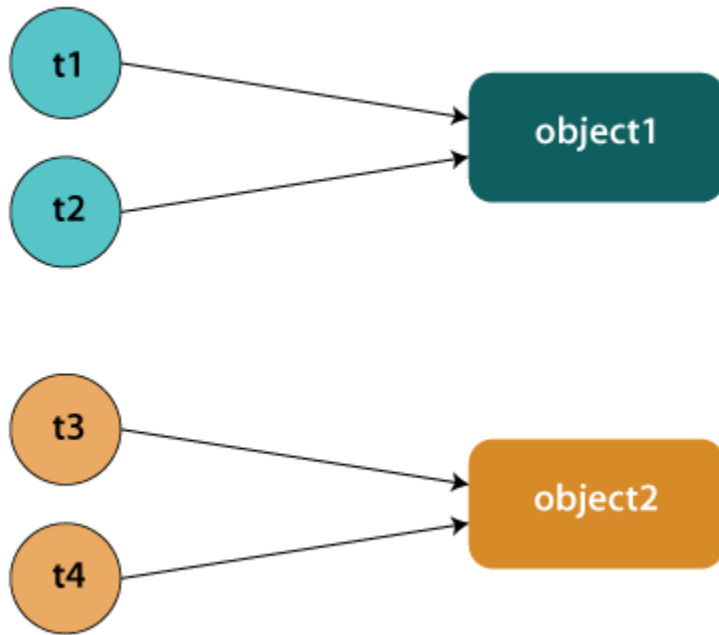
        t1.start();
        t2.start();
    }
}

```

## Static Synchronization

## Key Points

- Lock applies to class instead of object
- Used with static methods
- All threads share same class lock



## Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

## Example

```
class Table {
    static synchronized void printTable(int n) {
        for(int i=1; i<=5; i++) {
            System.out.println(n*i);
        }
    }
}
```

```

        Thread.sleep(2000);
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
}
}
}

```

```

class MyThread1 extends Thread {
    public void run() {
        Table.printTable(5);
    }
}

```

```

class MyThread2 extends Thread {
    public void run() {
        Table.printTable(10);
    }
}

```

```

class Test {
    public static void main(String[] args) {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();

        t1.start();
        t2.start();
    }
}e.printStackTrace();
}
}
}
}

```

```

class MyThread1 extends Thread {
    public void run() {
        Table.printTable(5);
    }
}

```

```

class MyThread2 extends Thread {
    public void run() {
        Table.printTable(10);
    }
}

```

```

    }
}

class Test {
    public static void main(String[] args) {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();

        t1.start();
        t2.start();
    }
}

```

## Inter-Thread Communication

Inter-thread communication allows threads to coordinate their actions by using methods from the `Object` class.

### Key Concepts:

#### 1. Core Methods:

- `wait()`: Called by a thread waiting for a condition
- `notify()`: Called by a thread when it fulfills the condition
- `notifyAll()`: Notifies all waiting threads

#### 2. Important Rules:

- These methods are in `Object` class, not `Thread` class
- Must be called from within synchronized context
- `wait()` releases lock immediately
- `notify()/notifyAll()` release lock eventually
- Only these methods can release locks

### Example Implementation:

```

class MyThread extends Thread {
    int total = 0;

    public void run() {
        synchronized(this) {
            System.out.println("Child Thread started calculation");
        }
    }
}

```

```

        for(int i = 1; i <= 10; i++) {
            total = total + i;
        }
        System.out.println("Child thread giving notification");
        this.notify();
    }
}

class Test {
    public static void main(String[] args) throws
InterruptedException {
        MyThread t = new MyThread();
        t.start();
        synchronized(t) {
            System.out.println("Main Thread waiting for updating");
            t.wait();
            System.out.println("Main Thread got notification");
            System.out.println(t.total);
        }
    }
}

```

**Output:**

```

Main Thread waiting for updating
Child Thread started calculation
Child thread giving notification
Main Thread got notification
55

```

## Deadlock

Deadlock is a situation where two or more threads are blocked forever, each waiting for the other to release resources.

**Characteristics:**

- Circular waiting for resources
- Each thread holds one resource while waiting for another
- No thread releases its resource until it gets the other

## Example of Deadlock:

```
class Test {
    public static void main(String[] args) {
        final String res1 = "hi";
        final String res2 = "bye";

        Thread t1 = new Thread() {
            public void run() {
                synchronized(res1) {
                    System.out.println("Thread1: Locking Resource
1");

                    synchronized(res2) {
                        System.out.println("Thread1: Locking
Resource2");
                    }
                }
            }
        };

        Thread t2 = new Thread() {
            public void run() {
                synchronized(res2) {
                    System.out.println("Thread2: Locking Resource
2");

                    synchronized(res1) {
                        System.out.println("Thread2: Locking Resource
1");
                    }
                }
            }
        };

        t1.start();
        t2.start();
    }
}
```

Output:

```
Thread2: Locking Resource 2  
Thread1: Locking Resource 1
```

## Drawbacks of Multithreading:

1. Deadlock potential
2. Thread starvation
3. Race conditions
4. Complex debugging

## Java 8 Features

### Functional Interfaces

A functional interface is an interface with exactly one abstract method. It may contain other default or static methods.

### Key Points:

- Single Abstract Method (SAM) interfaces
- Can have multiple default and static methods
- Annotated with `@FunctionalInterface`
- Used for functional programming

### Example 1: Basic Implementation

```
@FunctionalInterface  
interface A {  
    void m1();  
}  
  
class B implements A {  
    public void m1() {  
        System.out.println("M1-Method");  
    }  
}
```

```

}

class Test {
    public static void main(String[] args) {
        A a = new B();
        a.m1();
    }
}

```

## Example 2: Anonymous Implementation

```

@FunctionalInterface
interface A {
    void m1();
}

class Test {
    public static void main(String[] args) {
        A a = new A() {
            public void m1() {
                System.out.println("From M1 Method");
            }
        };
        a.m1();
    }
}

```

## Lambda Expressions

Lambda expressions provide a clear and concise way to implement single-method interfaces.

## Key Features:

- No name, return type, or modifier needed
- Arrow syntax (->)
- Can be used with functional interfaces
- Supports parameters and return values



## Syntax Comparison:

```
// Traditional method
public void m1() {
    System.out.println("Hello");
}

// Lambda equivalent
() -> {
    System.out.println("Hello");
}
```

## Examples:

```
// Simple Lambda
@FunctionalInterface
interface A {
    void m1();
}

class Test {
    public static void main(String[] args) {
        A a = () -> {
            System.out.println("Here M1 Method");
        };
        a.m1();
    }
}

// Lambda with Parameters
@FunctionalInterface
interface A {
    void m1(int i, int j);
}

class Test {
    public static void main(String[] args) {
        A a = (int i, int j) -> {
            System.out.println(i + j);
        };
    }
}
```

```

        };
        a.m1(10, 20);
    }
}

// Lambda with Return Value
@FunctionalInterface
interface A {
    int m1(int i, int j);
}

class Test {
    public static void main(String[] args) {
        A a = (int i, int j) -> {
            return i + j;
        };
        System.out.println(a.m1(20, 30));
    }
}

```

## Default Methods

Default methods allow interfaces to have methods with implementation without affecting implementing classes.

## Key Features:

- Marked with `default` keyword
- Can be overridden by implementing classes
- Enable interface evolution
- Support multiple inheritance

## Example 1: Basic Default Method

```

@FunctionalInterface
interface A {
    void m1(); // abstract method
}

```

```

        default void m2() {
            System.out.println("M2-Method");
        }
    }

    class B implements A {
        public void m1() {
            System.out.println("M1-Method");
        }
    }

    class Test {
        public static void main(String[] args) {
            A a = new B();
            a.m1();
            a.m2();
        }
    }

```

## Example 2: Multiple Inheritance

```

interface Right {
    default void m1() {
        System.out.println("Right-M1 Method");
    }
}

interface Left {
    default void m1() {
        System.out.println("Left-M1 Method");
    }
}

class Middle implements Right, Left {
    public void m1() {
        Right.super.m1();
        Left.super.m1();
    }
}

```

```

}

class Test {
    public static void main(String[] args) {
        Middle m = new Middle();
        m.m1();
    }
}

```

## Static Methods in Interfaces

Static methods in interfaces belong to the interface itself.

## Key Points:

- Cannot be overridden
- Called using interface name
- Can be used for utility methods

## Example:

```

interface A {
    static void m1() {
        System.out.println("M1 Method");
    }
}

class Test {
    public static void main(String[] args) {
        A.m1();
    }
}

```

## Stream API

Stream API provides powerful methods for processing sequences of elements.

## Common Operations:

1. Filter
2. Map
3. Collect
4. Sort
5. Reduce
6. Count
7. Min/Max

## Examples:

```
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(8, 2, 1, 3, 7, 6, 4);

        // Filter even numbers
        List<Integer> evenNumbers = list.stream()
            .filter(i -> i % 2 == 0)
            .collect(Collectors.toList());

        // Map: Add 10 to each element
        List<Integer> addedNumbers = list.stream()
            .map(i -> i + 10)
            .collect(Collectors.toList());

        // Count even numbers
        long evenCount = list.stream()
            .filter(i -> i % 2 == 0)
            .count();

        // Find minimum value
        int min = list.stream()
            .min((i1, i2) -> i1.compareTo(i2))
            .get();

        // Sort ascending
        List<Integer> sortedList = list.stream()
```

```

        .sorted()
        .collect(Collectors.toList());

// Sort descending
List<Integer> reverseSorted = list.stream()
    .sorted(Comparator.reverseOrder())
    .collect(Collectors.toList());

// Remove duplicates
List<Integer> distinctList = list.stream()
    .distinct()
    .collect(Collectors.toList());
    }
}

```

## forEach Method

The `forEach` method provides a new way to iterate over collections.

## Examples:

```

import java.util.*;

class Test {
    public static void main(String[] args) {
        // List iteration
        List<Integer> list = Arrays.asList(7, 2, 5, 6, 1);
        list.forEach(element -> System.out.print(element + " "));

        // Map iteration
        Map<Integer, String> map = new LinkedHashMap<>();
        map.put(1, "one");
        map.put(2, "two");
        map.put(3, "three");

        map.forEach((key, value) ->
            System.out.println(key + "=" + value));
    }
}

```

## Method References

Method references provide a shorthand notation for lambda expressions.

### Example:

```
import java.util.*;

class Test {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(7, 2, 5, 6, 1);

        // Using method reference
        list.forEach(System.out::println);
    }
}
```

## Interview Questions

### Q1: Display Employee Information Sorted by ID

```
import java.util.*;
import java.util.stream.*;

class Employee {
    private int empId;
    private String empName;
    private double empSal;

    Employee(int empId, String empName, double empSal) {
        this.empId = empId;
        this.empName = empName;
        this.empSal = empSal;
    }

    public int getEmpId() { return empId; }
    public String getEmpName() { return empName; }
    public double getEmpSal() { return empSal; }
}
```

```

class Test {
    public static void main(String[] args) {
        List<Employee> list = new ArrayList<>();
        list.add(new Employee(104, "Jack", 4000d));
        list.add(new Employee(101, "Morries", 1000d));
        list.add(new Employee(102, "Jennie", 2000d));
        list.add(new Employee(103, "Kelvin", 3000d));

        List<Employee> sortedList = list.stream()
            .sorted(Comparator.comparingInt(Employee::getEmpId))
            .collect(Collectors.toList());

        sortedList.forEach(emp ->
            System.out.println(emp.getEmpId() + " " +
                emp.getEmpName() + " " +
                emp.getEmpSal())
        );
    }
}

```

## Q2: Display Employee Information Sorted by Name

```

import java.util.*;
import java.util.stream.*;

class Employee {
    private int empId;
    private String empName;
    private double empSal;

    Employee(int empId, String empName, double empSal) {
        this.empId = empId;
        this.empName = empName;
        this.empSal = empSal;
    }

    public int getEmpId() { return empId; }
}

```



```
    public String getEmpName() { return empName; }
    public double getEmpSal() { return empSal; }
}

class Test {
    public static void main(String[] args) {
        List<Employee> list = new ArrayList<>();
        list.add(new Employee(104, "Jack", 4000d));
        list.add(new Employee(101, "Morries", 1000d));
        list.add(new Employee(102, "Jennie", 2000d));
        list.add(new Employee(103, "Kelvin", 3000d));

        List<Employee> sortedList = list.stream()
            .sorted(Comparator.comparing(Employee::getEmpName))
            .collect(Collectors.toList());

        sortedList.forEach(emp ->
            System.out.println(emp.getEmpId() + " " +
                emp.getEmpName() + " " +
                emp.getEmpSal())
        );
    }
}
```