JAVA

BASICS TO ADVANCED

THANK YOU SO MUCH!

prashant jain sir

COMLETE CODING BY PRASHANT SIR

INDEX

Chapter	Sub-Topics	Pages
1. Introduction to Java	1.1) Why you must learn Java? 1.2) What is a programming language? 1.3) What is an algorithm? 1.4) What is syntax? 1.5) History of JAVA 1.6) Magic of Byte Code 1.7) How Java boomed the internet? 1.8) BUZZ WORDS 1.9) Object Oriented Programming	1 - 3
2. Java Basics	2.1) Writing first class 2.2) Compiling and running 2.3) Anatomy of a class 2.4) JDK Vs. JVM Vs. JRE 2.5) Importance of the main method **CHALLENGES** **KEY POINTS**	4 - 6
3. Data types, Variables & Input	3.1) Variables? 3.2) Data types 3.3) Naming Conventions 3.4) Literals 3.5) Keywords 3.6) Escape sequences 3.7) User Input 3.8) Type conversion and casting	7 - 10
4. Operators and If-Else Statement	4.1) Assignment operator 4.2) Arithmetic operators 4.3) Order of operations 4.4) Shorthand operators 4.5) Unary operators 4.6) If-else 4.7) Relational operators	11 - 15

	4.8) Logical operators 4.9) Operator precedence 4.10) BitWise Operators	
5. While loop, Methods & Arrays	5.1) Comments 5.2) While Loop 5.3) What are Functions/Methods? 5.4) Return statement 5.5) Arguments vs. Parameters **CHALLENGES** 5.6) What is an Array? 5.7) 2D Arrays **ARRAY CHALLENGES**	16 - 33
6. Classes & Objects	6.1) POP Vs. OOP 6.2) Instance variables and methods 6.3) Declaring an object 6.4) Class vs. Object 6.5) This keyword, Static Keyword 6.6) Constructor, Code Blocks { } **CHALLENGES** 6.7) Stack Vs. Heap Memory 6.8) Primitive vs. Reference Types 6.9) Variable Scopes 6.10) Garbage collection & Finalize **KEY POINTS**	34 - 47
7. Control Statements, Math & String	7.1) Ternary Operator 7.2) Switch **CHALLENGES** 7.3) Loops (Do-While, For loop, Enhanced for loop) 7.4) Using Break & Continue 7.5) Recursion **CHALLENGES** 7.6) Random numbers & Math class 7.7) toString method 7.8) String class 7.9) StringBuffer vs. StringBuilder 7.10) Final Keyword	48 - 64

	CHALLENGES **KEY POINTS**	
8. Encapsulation & Inheritance	8.1) Intro to OOP Principle 8.2) Encapsulation 8.3) Import and packages 8.4) Access modifiers 8.5) Getter and Setter methods **CHALLENGES** 8.6) What is inheritance? 8.7) Types of inheritance 8.8) Object class 8.9) Equals and HashCode 8.10) Nested and Inner classes **CHALLENGES** **KEY POINTS**	65 - 80
9. Abstraction and Polymorphism	9.1) Abstraction 9.2) Abstract Keyword 9.3) Interfaces **CHALLENGES** 9.4) What is polymorphism? 9.5) References and objects 9.6) Method / constructor overloading 9.7) Super Keyword 9.8) Method/ Constructor overriding 9.9) Final keyword revisited 9.10) Pass by Value vs Pass by reference **CHALLENGES** **KEY POINTS**	81 - 95
10. Exception and File Handling	10.1) What is an Exception 10.2) Try-Catch 10.3) Types of Exceptions 10.4) Throw and throws keyword 10.5) Finally block 10.6) Custom exceptions **CHALLENGES** 10.7) FileWriter class	96 - 106

	10.8) FileReader **CHALLENGES** **KEY POINTS**	
11. Collections & Generics	11.1) Variable arguments 11.2) Wrapper classes 11.3) Collections library 11.4) List Interface 11.5) Queue Interface 11.6) Set Interface 11.7) Collections class **CHALLENGES** 11.8) Map Interface 11.9) Enums 11.10) Generics & Diamond Operators **CHALLENGES** **KEY POINTS**	107 - 123
12. Multi-threading & Executor Service	12.1) What is a Thread? 12.2) Creating a Thread (Extending Thread Class, Implementing Runnable Interface) 12.3) States of a Thread 12.4) Thread Priority 12.5) Join Method 12.6) Synchronize Keyword 12.7) Thread Communication **CHALLENGES** 12.8) Intro to Executor Service 12.9) Multiple Threads with executor 12.10) Returning Futures **CHALLENGES** **KEY POINTS**	124 - 145
13. Functional Programming	13.1) What is Functional programming 13.2) Lambda Expressions 13.3) What is a stream 13.4) Filtering & Reducing **CHALLENGES** 13.5) Functional Interfaces	146 - 158

	13.6) Method references 13.7) Functional vs. Structural Programming 13.8) Optional class **CHALLENGES** 13.9) Intermediate vs. Terminal Operations 13.10) Max, Min, Collect to list 13.11) Sort, Distinct, Map **CHALLENGES** **KEY POINTS**	

1. Introduction to Java

1.1) Why you must learn Java ?

- Most popular(Runs on more than 6B devices).
- Widely used(Web-Apps,Mobile Apps,Back-end,Enterprise Applications).
- Object oriented.
- Old but Gold.
- High paying jobs.
- Rich API's and good community support.

1.2) What is a programming language ?

Programming language is nothing but giving instructions to the computer.

Human instructions are given in high-level languages.These instructions are nothing but code.

Compiler converts this high-level language into low-level language(I.e., machine code which is of binary 010111….)

1.3) What is an algorithm?

Step-by-step instructions for solving a problem/ performing a task.The perfect example is that when you buy a packet of Maggie you can see the procedure to make it on the back of the pack, that is know to be the algorithm to make Maggie.

Algorithm is not specific to computer science.Any thing that can be written in step-by-step manner can be called as an algorithm.

1.4) What is syntax?

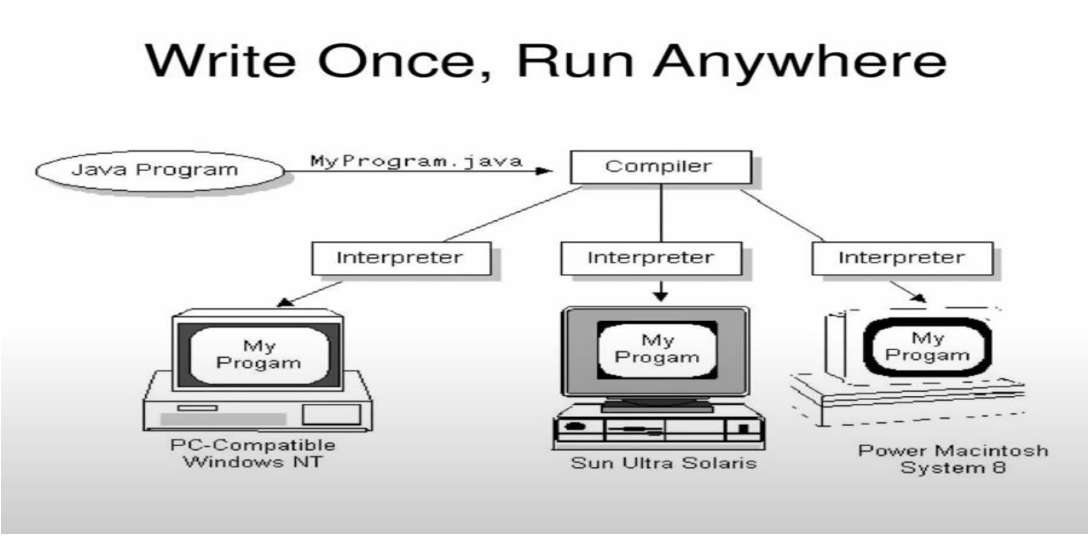
Rules of the language is known as the syntax and it must be followed for programming.

1.5) History of JAVA

Developed by James Gosling at Sun Microsystems in early 1990's.

Originally named as OAK later renamed as JAVA in 1995.

Principle:**"Write Once, Run Anywhere - WORA"(Cross platform compatibility)**



.java ➡ Compiler ➡ Byte code(Intermediate output)

.class file contains this byte code.

This byte code can be runned on any type of machine using JVM to produce machine code.

Byte code ➡ JVM ➡ Machine code

Java has backward compatibility which means code written in older versions typically runs on newer JVM's.

Java(Programming language) is named after the JAVA Island in Indonesia,Known for it's coffee.

coffee cup is the symbol of java which depicts that Our java is always fresh just like coffee.

Oracle acquired java in 2010.

1.6) Magic of Byte Code

Compiler converts source code into byte code.

JVM coverts byte code into machine code.

Byte code is not binary but machine code is.

JVM is machine specific means works and acts different for each machine.

Java has compiler+interpreter.

1.7) How java boomed the internet ?

WORA/WORE-Write once run everywhere, Java byte code can be executed on any machine that has JVM.

Java programs need not be rewritten for different systems.

JVM checks the byte code before executing for security concerns like memory corruption.

Supports sandboxing which means applets and applications runs in an restricted environment.

Java does not use pointers like in c/c++ to avoid memory related vulnerabilities.

1.8) BUZZ WORDS

1. Robust --> Strong and relaible.Doesn't crash and handles errors smoothly without breaking.
2. Multi threaded --> it's the ability of a CPU to execute multiple threads concurrently.
3. Architecture Neutral --> Compiled code/byte code can run on any device with a JVM regardless of the underlying hardware architecture.
4. Interpreted and High-performance --> JVM is an interpreter which means that byte code is converted into machine code using an interpreter.
5. Distributed --> Java programs can run on different computers which means that it is perfect for building apps that runs over the internet or in a network.

Interpreter executes the code line by line but compiler executes entire code at once.

1.9) Object Oriented Programming

Class: it's a blueprint.

Object: Instance of a class.

Properties: Variables inside a class that holds the characteristics of an object.

Methods: Functions inside a class that defines the behaviour of an object.

In the perspective of the mankind,

Class --> Human

Object --> name of the person

Properties--> Height,Weight and color, etc.,

Methods --> breathing,going to office, etc.,

2. Java Basics

Java belongs to oracle(present).Oracle bought entire sun microsystems.

2.1) Writing first class

```java
import java.lang.*;

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

Execution:

>javac firstProgram.java --> converts the .java file into byte code which is .class file.

>java firstProgram --> gives out the output.

2.2) Compiling and running

.java(Source file) ➡ Javac command(Compiler) ➡ .class(byte code) ➡ java command(JVM) 0101... ⟹ O/P.

2.3) Anatomy of a class

Public means anyone can access it.

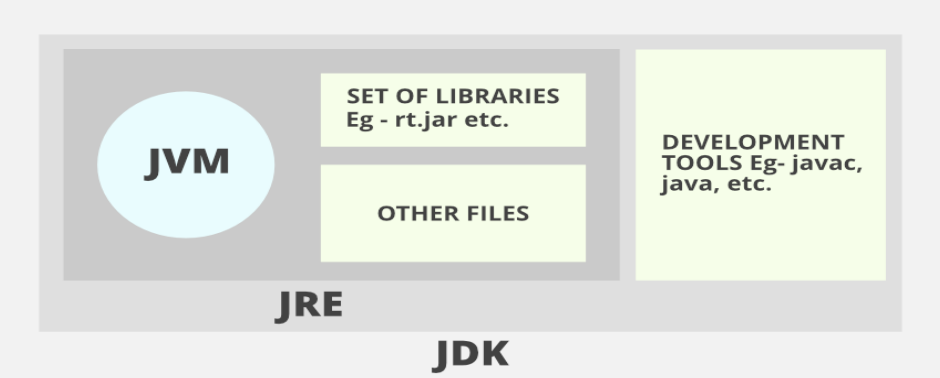
Name of the class should have first letter of the word as capital and should not have special characters like -,_,@,..

Main is the name of the method.

Void is the return type of the main method.

String args[] is the argument of the method main and it must be an array of strings and the array is called as args.

2.4) JDK Vs. JVM Vs. JRE



JDK is the SDK(software development kit) used to develop java applications.It is the superset of JRE.

JDK is used by the the developers like us to write the code.

JDK=(Developemental tools)+JRE

JRE=JVM+(set of libraries)+(other files)

JRE doesn't have tools and utilities for developers like compilers and it's a part of JDK but can be downloaded separately.

JVM is machine specific but javac is architecture independent.

JVM converts the byte code into machine code and ensures WORE(Write once run everywhere).

JVM can not not be downloaded seperately like JRE and JDK. If we want to run the byte code into a system we needed to download JRE/JDK.

2.5) Importance of the main method

Main method is the entry point of execution of every java program.Without main method JVM don't know where to start the execution of the program.

If there are 'n' no.of methods in a java program, main method will be executed first.

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

Main method cannot be duplicated and can not be overwritten.

Void is the return type of the main method.

Static methods can be called without needing to create an object for the class(need not to instantiate), where the method was present.We make the main method public because JVM needs to access the main method to execute the program.

CHALLENGES

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

// *
// * *
// * * *
// * * * *
// * * * * *
```

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

// * * * * *
// * * * *
// * * *
// * *
// *
```

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

KEY POINTS

- .class file contains byte code and byte code is platform independent.
- Println() adds new line at the end of the line.
- JDK>JRE>JVM
- Main method must needed to be declared as public.
- Java was named after an island in Indonesia where coffee beans grow and java was first released in 1995.
- High level language(code) is not understood by java or any other languages.

3. Data types,Variables & Input

3.1) Variables?

These are the containers that are used to store data.

All the variables are not of same size.

Program stores in the computer storage and variables are stored in the memory.

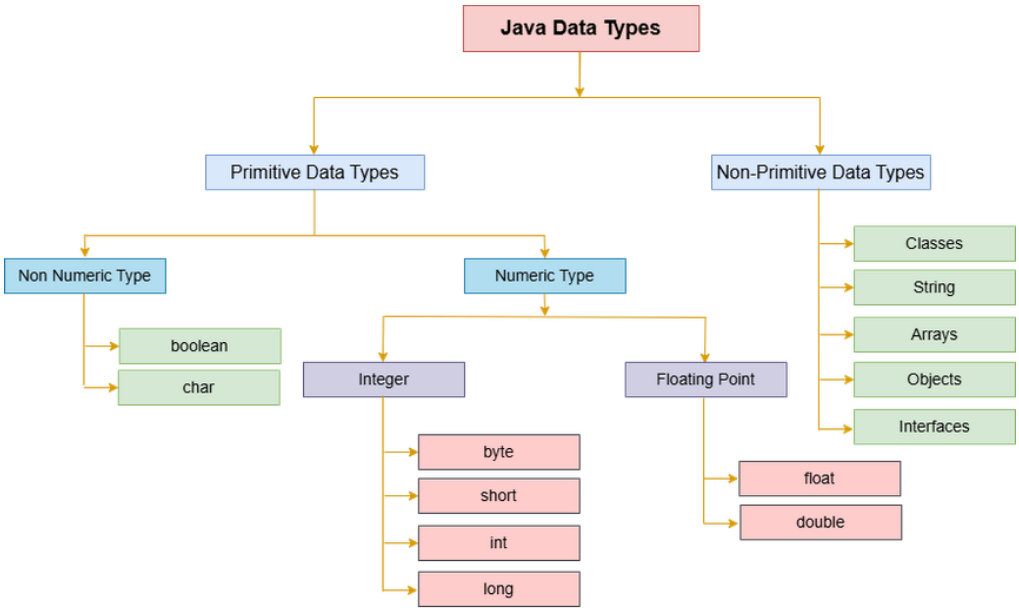
Memory means RAM. Storage means Hardisk.RAM is volatile but not hardisk.

int a=20;

int is the data type and a is the name of the variable,20 is the value assigned to the variable.

3.2) Data types

1 byte= 8 bits



Non-primitive data types doesn't have a fixed size and they needed to be created using the new keyword, where as primitive data types have the fixed size.

Data Type	Size (in bytes)
byte	1
short	2
int	4
long	8
float	4
double	8
char	2
boolean	1 (JVM-dependent)

Value assignment to the variables of the datatype float need to be specifies as F/f, otherwise it will be assumed as double and the same goes with long and need to be specified with either l or L at the end.

```
public class Variables {
    public static void main(String[] args) {
        int myInt = 5;
        float myFloat = 3.14f;
        byte myByte = 120;
        short myShort = 500;
        boolean isVegan = true;
        double myDouble = 5888.888;
        long myLong = 5874694855L;
    }
}
```

3.3) Naming Conventions

- ✓ Name of the variable is also known as the identifier.
- ✓ Spaces are not allowed.
- ✓ Only alphanumeric characters like [a-z],[A-Z],[0-9],$,_
- ✓ Should not start with digits.
- ✓ Keywords should not be used.
- ✓ Make sure that name of the variable is simple but also understandable when others try to understand our code.
- ✓ Java identifiers are case sensitive.Below is the example and the code is valid

```
public class Variables {
    public static void main(String[] args) {
        int myInt = 5;
        int myINT = 10;
    }
}
```

Java majorly uses camel case convention.

Classes in java starts with the capital letter then follows camel case convention.

Starting letter of the variable name must be lower case.Examples of the naming conventions are

camelCase ➡ start with a lowe case letter, capitalize each subsequent word. ex: myName, myAge

kebab-case ➡ start with lower case letter and separate each subsequent word with hyphen(-). ex: my-name,my-age

snake_case ➡ start with lower case letter and separate each subsequent word with underscore(_). ex: my_name,my_age

3.4) Literals

Values assigned to the variables are known as the literals.From the above code in datatypes,

5 is a int literal.

true is a boolean literal and so on…

Syntax of the variable declaration is data_type identifier=literal;

3.5) Keywords

Keywords have predefined meaning and they can not be used as

- Variable names
- Class name
- Method name
- Package name
- Any user-defined identifiers

3.6) Escape sequences

\ → this is known as escape sequence.whenever compiler finds this in the print statement it ignores the next character after this backslash.

Below is the code to print double quotes inside a string.

```
public class Escape {
    public static void main(String[] args) {
        System.out.println("My name is \"Hemanth\"");
    }
}
```

And the output will be

My name is “Hemanth”

If you want to display the backslash(\) itself in the print statement then use double backslash(\\).

\n → new line

\t → tab

\b → inserts backspace i.e sout(“hello\bmate”); gives hellmate.

\’ → to insert a single quote character in the text.

\” → to insert double quote character in the text.

\\ → to insert backslash character in the text.

3.7) User Input

```
import java.util.Scanner;

public class UserInput {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String name = s.nextLine();// input's entite line
        String last_name = s.next();// only inputs single word
        int age = s.nextInt();
        float salary=s.nextFloat();
        double taxation=s.nextDouble();
    }
}
```

3.8) Type conversion and casting

1) Implicit type conversion/Automatic type conversion

```
public class UserInput {
    public static void main(String[] args) {
        float f = 5;
        // 5 is int, on assigning to f the compiler automatically converts to float
        System.out.println(f); //5.0
    }
}
```

2) Explicit type conversion

```
public class UserInput {
    public static void main(String[] args) {
        float f = 5.0d;// 5.0d is double
        System.out.println(f); // on printing f the compiler throws error: incompatible types
    }
}
```

We have to declare explicitly to convert literals of large sized data types to literals of small sized data types.

```
public class UserInput {
    public static void main(String[] args) {
        // implicit
        long big = 45;//45
        float dec = 3;//3.0
        double d = 3.4f;//3.4000000953674316
        // explicit
        float eDec = (float) 3.4;//3.4
        long eBig = (long) 3.4;//3
        int eInt = (int) 3.4;//3
    }
}
```

4. Operators and if-else statement

4.1) Assignment operator

One file should have only one public class.

```
public class Chapter4 {
    public static void main(String[] args) {
        int num=5;//assigns 5(literal) to num(variable/identifier)
    }
}
```

Exercise of the assignment operator

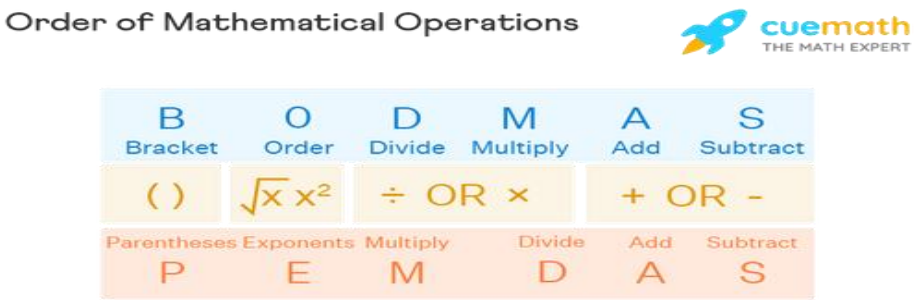
```
//to swap the values of a and b
temp=a;
a=b;
b=temp;
```

4.2) Arithmetic operators

Operator	Operation
+ (Addition)	z = x + y
- (Subtraction)	z = x - y
* (Multiplication)	z = x * y
/ (Division)	z = x / y
% (Modulus)	z = x % y

```
public class Chapter4 {

    public static void main(String[] args) {
        int a = 5;
        int b = 2;
        System.out.println(a + b); // 7
        System.out.println(a - b); // 3
        System.out.println(a / b); // 2
        System.out.println(a % b); // 1
        float x = 5;
        float y = 2;
        System.out.println(x + y); // 7.0
        System.out.println(x - y); // 3.0
        System.out.println(x / y); // 2.5
        System.out.println(x % y); // 1.0
    }
}
```

4.3) Order of operations



```java
public class Chapter4 {
    public static void main(String[] args) {
        System.out.println(9 / 3 / 3); // 1
        System.out.println(9 / (3 / 3)); // 9
        System.out.println(9 / (3 / 3 + 2)); // 3
        //as execution goes from left to right along with BODMAS/PEMDAS
    }
}
```

4.4) Shorthand operators

Operator	Operation	Equivalent to
=	num = 5	num = 5
+=	num+=5	num = num+5
-=	num-=5	num = num-5
*=	num*=5	num = num*5
/=	num/=5	num = num/5
%=	num%=5	num = num%5

4.5) Unary operators

Operator	Description	Example
-	Converts a positive value to a negative	x = -y
Pre Increment	Increment the value by 1 and then use it in our statement	x = ++y
Pre Decrement	Decrement the value by 1 and then use it in our statement	x = --y
Post Increment	Use current value in the statement and then increment it by 1	x = y++
Post Decrement	Use current value in the statement and then decrement it by 1	x = y--

```java
public class Chapter4 {
    public static void main(String[] args) {
        int a = 10;
        System.out.println(a++); // 10
        System.out.println(a);   // 11
        System.out.println(++a); // 12
        System.out.println(a);   // 12
        System.out.println(--a); // 11
        System.out.println(a);   // 11
        System.out.println(a--); // 11
        System.out.println(a);   // 10
    }
}
```

```java
public class Chapter4 {
    public static void main(String[] args) {
        float a = 3.4f;
        float b = 4.1f;
        System.out.println("Addition:" + a + b);   // 3.44.1 -->concatinates instead of add
        System.out.println("Addition" + (a + b));  // 7.5
        System.out.println("Mul:" + a * b); // 13.940001 -->works fine
        System.out.println(Math.pow(5, 2)); //25.0 as Math.pow() returns double

    }
}
```

4.6) If-else

```java
public class Chapter4 {
    public static void main(String[] args) {
        int age = 18;
        if (age < 13)
            System.out.println("You are a child.");
        else if (age < 18)
            System.out.println("You are a teenager.");
        else if (age < 60)
            System.out.println("You are an adult."); // This will be executed
        else
            System.out.println("You are a senior citizen.");
        // As there is only single statement under if and also under else, we omitted {}
        // but using {} is the best practice
    }
}
```

4.7) Relational operators

Order of arithmetic operators is greater than that of relational operators.

```java
public class Chapter4 {
    public static void main(String[] args) {
        System.out.println(2 + 3 > 5 + 2); // 5>7
        // the arithmetic operator(+) was evaluated first and then relational operator(>)
    }
}
```

Operator	Description
==	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!=	Not Equal to

4.8) Logical operators

It has lower priority than math and comparison operators.

Operator	Name	Description
&&	Logical AND	True if **both** conditions are true
||	Logical OR	True if **at least one** condition is true
!	Logical NOT	Reverses the result (true becomes false)

```java
public class Chapter4 {
    public static void main(String[] args) {
        int a = 10;
        int b = 9;
        int c = 5;
        int big = a > b ? (a > c ? a : c) : (b > c ? b : c); // Largest of 3 numbers
        System.out.println(big); // 10
    }
```

```
}
public class Chapter4 {
    public static void main(String[] args) {
        int year = 296;
        boolean isLeap = false;
        if (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0))
            isLeap = true;
        System.out.println(isLeap); // true
        // if year%4==0 and year%100!=0 then it is a leap year and also
        // if year%400==0 then also leap year
    }
}
```

4.9) Operator precedence

Precedence: It decides which operator should be used first when there are multiple operators in an expression.

Example: In 3 + 5 * 2 = 3 + 10 = 13.

Associativity: It tells us the order in which operators are used when there are multiple operators of the same kind.

Example (Left to Right): For 5 - 3 - 1 = 2 - 1 = 1.

Example (Right to Left): For a = b = 10 is same as a = (b = 10).

4.10) BitWise Operators

Used to perform operations on individual bits of binary values.

Operator	Name	Description	Example	Result
&	Bitwise AND	1 if both bits are 1	5 & 3	1
|	Bitwise OR	1 if at least one bit is 1	5 | 3	7
^	Bitwise XOR	1 if bits are different	5 ^ 3	6
~	Bitwise NOT (complement)	Flips all bits	~5	-6
<<	Left shift	Shifts bits to the left	5 << 1	10
>>	Right shift	Shifts bits to the right (keeps sign bit)	5 >> 1	2
>>>	Unsigned right shift	Shifts right, fills with 0 (ignores sign)	-5 >>> 1	A large positive value

```
public class Chapter4 {
    public static void main(String[] args) {
        int a = 5; // 0101
        int b = 3; // 0011
        System.out.println("a & b = " + (a & b)); // 1
        System.out.println("a | b = " + (a | b)); // 7
        System.out.println("a ^ b = " + (a ^ b)); // 6
        System.out.println("~a = " + (~a)); // -6
        System.out.println("a << 1 = " + (a << 1)); // 10
        System.out.println("a >> 1 = " + (a >> 1)); // 2
        System.out.println("-5 >>> 1 = " + (-5 >>> 1)); // Big positive number
    }
}
```

Below program is to check if the number is odd or even using only bit-wise operators

```java
import java.util.Scanner;
public class Chapter4 {
    public static void main(String[] args) {
        // check weather the numbeer is odd/even using only bitwise operators
        Scanner s = new Scanner(System.in);
        int a = s.nextInt();
        if ((a & 1) == 1) {
            System.out.println("odd");
        } else {
            System.out.println("even");
        }
        // All the even numbers ends with 0.If we do this (num&1) and we get 1 means num
        // also had 1 at the end which is odd
    }
}
```

5. While loop,Methods &Arrays

5.1) Comments

To add notes to the code which doesnot execute.

// → single line comment

/* */ → multi-line comment

/** */ → Java docs

5.2) While Loop

A while loop in Java repeatedly executes a block of code as long as a given condition is true.

Syntax:

Initialization;

While(condition){

Updation;

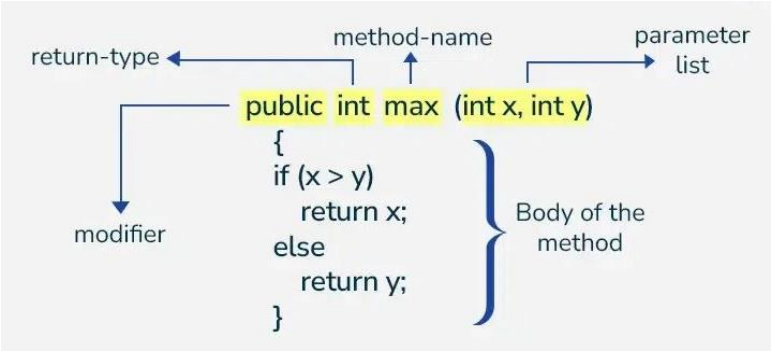
}

5.3) What are Functions/Methods?

Naming convention is same as of variables which is camelCase.

Block of reusable code.Follows DRY (Don't repeat yourself) principle.

Syntax of the method as folllows



Java can have multiple static methods.static methods can be called from any class without creating an object as long as they are public.

Body of the method is also known as method definition/function definition.

Main method will be executed first regardless of no.of methods, as main method is the entry point of execution.

Function calling / Invoking a method: In the above syntax the function call will be *max(a,b);*

If there are no parameters then the syntax would be *func_name();*

JVM calls main method but we need to call the methods we declared.

```
import java.util.Scanner;

public class Chapter5 {
    public static void greetins(String name) {
        System.out.println("Good morning " + name + " !");
    }
```

```
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter your name:");
        String name = s.next();
        greetins(name);
    }
}
```

5.4) Return statement

To send a value back from function.

Return statement ends the execution immediately.

Code jumping can be possible through function calls.

We can return only one value from a method, not multiple.You can wrap multiple values in an array, object, or a class to return them together.

Prefer return statement over global variables (declared inside a class and outside methods).

```
public class Chapter5 {
    static int i = 5;
    public static void main(String[] args) {
        System.out.println(i); // can be accessed without creating an object.
    }
}
```

```
public class Chapter5 {
    int i = 5;
    public static void main(String[] args) {
        Chapter5 c = new Chapter5();
        System.out.println(c.i); // can be accessed by creating an object.
    }
}
```

Return statement should be at end of the function, if it is in the middle anywhere in the function the code after the return statement in the function will not be executed.

A Java program with two methods: one returns a string, and the other returns an array.

```
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        String message = getGreeting();
        System.out.println(message);
        int[] numbers = getNumbers();
        for (int num : numbers) {
            System.out.print(num + " ");
        }
    }
    static String getGreeting() {
        return "Welcome to Java!";
    }
    static int[] getNumbers() {
        return new int[] { 10, 20, 30, 40, 50 };
    }
}
```

5.5) Arguments vs. Parameters

Parameters = placeholders

Arguments = actual values passed

```java
void greet(String name) { // "name" is a parameter
    System.out.println("Hello, " + name);
}
greet("Alice"); // "Alice" is an argument
```

Default parameters are not supported in java like python,c++. we need to use method over loading.Method overriding is only occurs in inheritence.

```java
public class Chapter5 {
    // Method with one parameter
    void greet(String name) {
        System.out.println("Hello, " + name);
    }
    // Overloaded method with no parameters (default value)
    void greet() {
        greet("Guest"); // calling with default value
    }
    public static void main(String[] args) {
        Chapter5 obj = new Chapter5();
        obj.greet(); // Output: Hello, Guest
        obj.greet("Alice"); // Output: Hello, Alice
    }
}
```

CHALLENGES

1) Print multiplication table

```java
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number for which you want to print the table:");
        int num = s.nextInt();
        mulTable(num);
    }
    public static void mulTable(int num) {
        int i = 1;
        while (i <= 10) {
            System.out.println(num + " * " + i + " = " + (num * i));
            i++;
        }
    }
}
/*
Enter the number for which you want to print the table:7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
*/
```

2) Print all odd numbers from 1 to specified number

```java
import java.util.Scanner;
```

```java
public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number upto which you want to add odd numbers:");
        int num = s.nextInt();
        oddSum(num);
    }
    public static void oddSum(int num) {
        int i = 1;
        int count = 0;
        while (i <= num) {
            if (i % 2 == 0)
                count += i;
            i++;
        }
        System.out.println(count);
    }
}
/*
 * Enter the number upto which you want to add odd numbers:100
 * 2550
 */
```

3) Factorial of a number

```java
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number to find factorial:");
        int num = s.nextInt();
        long res = fact(num);
        if (res == -1)
            System.out.println("cant find factorilas for -ve values");
        else
            System.out.println(res);
    }
    public static long fact(int num) {
        if (num > 0) {
            if (num == 0 || num == 1)
                return 1;
            else
                return num * fact(num - 1);
        } else {
            return -1;
        }
    }
}
/*
 * Enter the number to find factorial:5
 * 120
 */
```

Without recursive functions the code is below

```java
 public static long fact(int num) {
        long out = 1, i = 1;
        if (num > 0) {
            if (num == 0 || num == 1)
                return 1;
            else {
```

```
                while (i <= num) {
                    out *= i;
                    i++;
                }
                return out;
            }
        } else {
            return -1;
        }
    }
```

4) Sum of digits of an integer

```
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number:");
        int num = s.nextInt();
        sumOfDigits(num);
    }
    public static void sumOfDigits(int num) {
        int out = 0;
        while (num > 0) { // num>=0 makes you stuck in infinite loop becz at the end num is
                                always equals to zero.
            int rem = num % 10;
            out = out + rem;
            num = num / 10;
        }
        System.out.println("sum of digits: " + out);
    }
}
/*
 * Enter the number:100
 * sum of digits: 1
 */
```

5) LCM of 2 numbers

```
public class Lcm {
    public static void main(String[] args) {
        int val = findLcm(300, 550); // 33000
        System.out.println(val);
    }
    public static int findLcm(int a, int b) {
        int i = 1;
        while (true) {
            if ((a * i) % b == 0)
                return a * i;
            i++;
        }
    }
}
```

6) HCF of 2 numbers

```
public class Hcf {
    public static void main(String[] args) {
        System.out.println("HCF: " + hcf(20, 30)); // HCF: 10
    }
    public static int hcf(int a, int b) {
```

```
        int small = Math.min(a, b);
        while (small > 0) {
            if (a % small == 0 && b % small == 0)
                return small;
            small--;
        }
        return 0;
    }
}
```

7) HCF and LCM

$$lcm(a,b) = \frac{a * b}{hcf(a,b)}$$

GCD and GCF are same thing but different names as divisors and factors are same.

```
public class Chapter5 {
    public static void main(String[] args) {
        System.out.println("GCD:" + gcd(12, 18));
        lcm(7, 35);
    }
    public static int gcd(int num1, int num2) {
        int small = num1 < num2 ? num1 : num2, i = 1;
        while (small >= 1) {
            if (num1 % small == 0 && num2 % small == 0) {
                return small;
            }
            small--;
        }
        return -1;
    }
    public static void lcm(int num1, int num2) {
        System.out.println("LCM: " + (num1 * num2) / gcd(num1, num2));
    }
}
/*
 * GCD:6
 * LCM: 35
 */
```

8) Prime or not

```
public class Chapter5 {
    public static void main(String[] args) {
        isPrime(5);
    }
    public static void isPrime(int num) {
        int i = 1, cnt = 0;
        while (i <= num) {
            if (num % i == 0)
                cnt++;
            i++;
        }
        if (cnt == 2)
            System.out.println("prime"); // prime
        else
            System.out.println("not prime");
    }
}
```

9) Reverse of an integer

```
public class Chapter5 {
    public static void main(String[] args) {
        reverse(123);
    }
    public static void reverse(int num) {
        int rem = 0, reverse_num = 0;
        while (num > 0) {
            rem = num % 10;
            reverse_num = (reverse_num * 10) + rem;
            num = num / 10;
        }
        System.out.println("Reverse: " + reverse_num); // Reverse: 321
    }
}
```

10) Fibonacci series

```
public class Chapter5 {
    public static void main(String[] args) {
        fibb(10);
    }
    public static void fibb(int num) {
        int a = 0, b = 1, fib = 0;
        if (num == 1)
            System.out.println(0);
        else {
            while (num >= 1) {
                a = b;
                b = fib;
                fib = a + b;
                System.out.print(fib + " "); // 1 1 2 3 5 8 13 21 34 55
                num--;
            }
        }
    }
}
```

11) Amstrong number/narcissistic number

If it's an 3 digit number then sum of cubes of individual digits of that number, if it's a 4 digit number then sum of power 4 of individual digits of that number.

```
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number: ");
        int num = s.nextInt();
        System.out.println("Is Amstrong: " + amstrongOrNot(num));
    }
    public static boolean amstrongOrNot(int num) {
        int res = 0, rem, dummy1 = num, dummy = num, count = 0;
        while (num > 0) {
            num /= 10;
            count++;
        }
        while (dummy > 0) {
            rem = dummy % 10;
```

```
            res += Math.pow(rem, count);
            dummy /= 10;
        }
        return dummy1 == res ? true : false;
    }
}
```

12) Pattern printing

```
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no.of rows: ");
        int rows = s.nextInt();
        int i = 1;
        while (i <= rows) {
            int j = 0;
            while (i > j) {
                System.out.print("* ");
                j++;
            }
            i++;
            System.out.println();
        }
    }
}
 //Enter no.of rows: 10
 //*
 //* *
 //* * *
 //* * * *
 //* * * * *
 //* * * * * *
 //* * * * * * *
 //* * * * * * * *
 //* * * * * * * * *
 //* * * * * * * * * *
```

```
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no.of rows: ");
        int rows = s.nextInt();
        int i = rows;
        while (i >= 1) {
            int j = 1;
            while (i >= j) {
                System.out.print("* ");
                j++;
            }
            i--;
            System.out.println();
        }
    }
}
// Enter no.of rows: 7
// * * * * * * *
```

```
// * * * * * *
// * * * * *
// * * * *
// * * *
// * *
// *
```

```
import java.util.Scanner;

public class Chapter5 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no.of rows: ");
        int rows = s.nextInt();
        int i = rows;
        while (i >= 1) {
            int k = rows;
            while (k > i) {
                System.out.print("  ");
                k--;
            }
            int j = 1;
            while (i >= j) {
                System.out.print("* ");
                j++;
            }
            i--;
            System.out.println();
        }
    }
}
// Enter no.of rows: 7
// * * * * * * *
//   * * * * * *
//     * * * * *
//       * * * *
//         * * *
//           * *
//             *
```

```
public class pattern {
    public static void main(String[] args) {
        int i = 1;
        int rows = 5;
        while (i <= rows) {
            int k = rows;
            while ((k - i) > 0) {
                System.out.print("  ");
                k--;
            }
            int j = 0;
            while (j < i) {
                System.out.print("* ");
                j++;
            }
            i++;
            System.out.println();
        }
    }
}
```

```
// *
// * *
// * * *
// * * * *
// * * * * *
```

5.6) What is an Array?

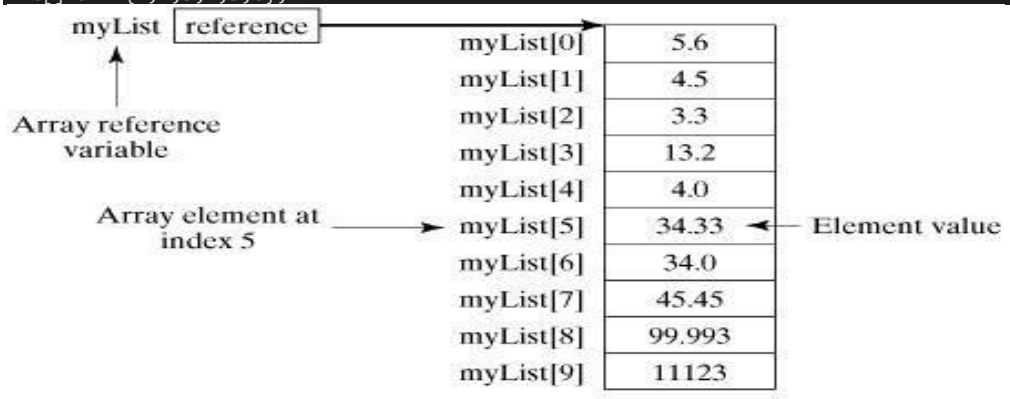
Arrays are used to store multiple values of same datatype in single variable.

Indexing starts with zero.

All the variables are stored in continuous memory locations.

Name of the variable addresses to the first element of the array.

```
int[] arr=new int[10]; //Declaration
int[] arr={1,2,3,4,5,6};
```



Once the size of the array is declared, can't be modified later.

If the array elements are predefined then no need to mention the size.

```
import java.util.Scanner;

public class Arrays {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        System.out.print("The array you entered: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
        System.out.print("Enter the search element: ");
        int se = s.nextInt();
        boolean isFound = false;
        for (int num : arr) {
            if (se == num) {
                isFound = true;
            }
        }
        System.out.println("is Found!: " + isFound);
    }
}
// Enter 1 th element:8
// Enter 2 th element:7
```

```
// Enter 3 th element:6
// Enter 4 th element:9
// Enter 5 th element:1
// The array you entered: 8 7 6 9 1
// Enter the search element: 6
// is Found!: true
```

5.7) 2D Arrays

In a school there are 60 students and there are 6 subjects for each student then the possible 2-D array to store marks of each student will be

int [][] stu_marks=new int[60][6];

2nd Student 1st subject marks can be accessed by *stu_marks[1][0]*

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Declaration of the 2D-array can done as follows

```
int[][] arr=new arr[2][3];
int[][] Arr={{1,2,3},{4,5,6}};
```

If the no.of columns for each row are not same then it is known as the heterogeneous array.

```
public class Array2d {
    public static void main(String[] args) {
        int[][] Arr = { { 1, 2, 3 }, { 4, 5, 6 } };
        System.out.println(Arr.length);     //2
        System.out.println(Arr[0].length);  //3
    }
}
```

```
// 2-D Array Traversal
public class Array2d {
    public static void main(String[] args) {
        int[][] Arr = { { 1, 2, 3 }, { 4, 5, 6 } };
        int i = 0;
        while (i < Arr.length) {
            int j = 0;
            while (j < Arr[i].length) {
                System.out.print(Arr[i][j] + " ");
                j++;
            }
            System.out.println();
            i++;
        }
    }

}
// 1 2 3
```

```
// 4 5 6
```

ARRAY CHALLENGES

1) Sum and avg. of all the elements in an array

```
import java.util.Scanner;

public class Sum_avg {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        int sum = 0, avg;
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
            sum += arr[i];
        }
        System.out.println("Sum: " + sum);
        System.out.println("Avg: " + (sum / arr.length));
    }
}
// Enter 1 th element:1
// Enter 2 th element:2
// Enter 3 th element:3
// Enter 4 th element:4
// Enter 5 th element:5
// Sum: 15
// Avg: 3
```

2) Find no.of occurances of an element in an array

```
import java.util.Scanner;

public class RepArrya {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        System.out.print("Enter the ele:");
        int ele = s.nextInt(), count = 0;
        for (int num : arr) {
            if (ele == num)
                count++;
        }
        System.out.println(ele + " repeated " + count + " times!");
    }
}
// Enter 1 th element:1
// Enter 2 th element:1
// Enter 3 th element:1
// Enter 4 th element:2
// Enter 5 th element:3
// Enter the ele:1
// 1 repeated 3 times!
```

3) Max and min elements of array

```
import java.util.Scanner;
```

```java
public class Maxminarray {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        int max = arr[0], min = arr[0];
        for (int num : arr) {
            if (num > max)
                max = num;
            else if (num < min)
                min = num;
        }
        System.out.println("Max: " + max + ", Min: " + min);
    }
}
// Enter 1 th element:5
// Enter 2 th element:8
// Enter 3 th element:9
// Enter 4 th element:4
// Enter 5 th element:7
// Max: 9, Min: 4
```

4) Array sorted or not

```java
import java.util.Scanner;

public class Sortedornot {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        int flag = 0, i = 0;
        do {
            if (arr[i] < arr[i + 1])
                flag++;
            i++;
        } while (i < arr.length - 2);
        if (flag == arr.length - 2)
            System.out.println("sorted!");
        else
            System.out.println("not Sorted!");
    }
}
// Enter 1 th element:5
// Enter 2 th element:1
// Enter 3 th element:9
// Enter 4 th element:4
// Enter 5 th element:6
// not Sorted!
```

5) Return array after deleting specififc element

```java
import java.util.Scanner;
```

```java
public class Deletingele {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        System.out.print("Enter the element you want to delete: ");
        int se = s.nextInt(), i = 0, index = -1;
        while (i < arr.length) {
            if (se == arr[i])
                index = i;
            i++;
        }
        if (index == -1)
            System.out.println("Element not found!");
        else {
            arr[index] = 0;
            for (int j = index; j < arr.length - 1; j++) {
                arr[j] = arr[j + 1];
            }
            arr[arr.length - 1] = 0;
            for (int num : arr)
                System.out.print(num + " ");
        }
    }
}
// Enter 1 th element:5
// Enter 2 th element:8
// Enter 3 th element:1
// Enter 4 th element:6
// Enter 5 th element:4
// Enter the element you want to delete: 1
// 5 8 6 4 0
```

6) Reverse an array

```java
import java.util.Scanner;

public class Reversing_array {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        int[] new_arr = new int[arr.length];
        int i = arr.length - 1;
        while (i >= 0) {
            int j = arr.length - 1 - i;
            new_arr[i] = arr[j];
            i--;
        }
        System.out.print("Entered array: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.print("\nReversed array: ");
        for (int num : new_arr) {
```

```
            System.out.print(num + " ");
        }
    }
}
// Enter 1 th element:5
// Enter 2 th element:6
// Enter 3 th element:9
// Enter 4 th element:4
// Enter 5 th element:3
// Entered array: 5 6 9 4 3
// Reversed array: 3 4 9 6 5
```

The below can also be the solution

```
import java.util.Scanner;

public class Reverse_chatgpt {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
        }
        System.out.print("Entered array: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
        for (int i = 0; i < arr.length / 2; i++) {
            int temp = arr[i];
            arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = temp;
        }
        System.out.print("\nReversed array: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
// Enter 1 th element:1
// Enter 2 th element:2
// Enter 3 th element:5
// Enter 4 th element:8
// Enter 5 th element:4
// Entered array: 1 2 5 8 4
// Reversed array: 4 8 5 2 1
```

7) Array is palindrome or not

```
import java.util.Scanner;

public class PalindromArray {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int[] arr = new int[5];
        int[] copy = new int[5];
        for (int i = 0; i < arr.length; i++) {
            System.out.print("Enter " + (i + 1) + " th element:");
            arr[i] = s.nextInt();
            copy[i] = arr[i];
        }
        boolean isPalindrome = true;
```

```
        for (int i = 0; i < arr.length / 2; i++) {
            int temp = arr[i];
            arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = temp;
        }
        int i = 0;
        while (i < arr.length) {
            if (arr[i] != copy[i])
                isPalindrome = false;
            i++;
        }
        if (isPalindrome == true) {
            System.out.println("Palindrome array");
        } else {
            System.out.println("Not palindrome array");
        }
    }
}
// Enter 1 th element:1
// Enter 2 th element:2
// Enter 3 th element:3
// Enter 4 th element:2
// Enter 5 th element:1
// Palindrome array
```

8) Merge 2 sorted arrays

```
import java.util.Scanner;

public class Sortingarrays {
    public static void main(String[] args) {
        int arr1[] = { 5, 5, 7, 1, 2 };
        int arr2[] = { 45, 22, 01, 69, -4 };
        int[] final_arr = combiningAndSorting(arr1, arr2);
        System.out.print("\nsorted array: ");// sorted array: -4 1 1 2 5 5 7 22 45 69
        for (int num : final_arr) {
            System.out.print(num + " ");
        }
    }
    public static int[] combiningAndSorting(int[] arr1, int[] arr2) {
        int len = arr1.length + arr2.length;
        int[] final_arr = new int[len];
        for (int i = 0; i < arr1.length; i++) {
            final_arr[i] = arr1[i];
        }
        for (int i = arr1.length, j = 0; i < final_arr.length; i++, j++) {
            final_arr[i] = arr2[j];
        }
        // bubble sort
        int i = 0;
        while (i < final_arr.length) {
            int j = 0;
            while (j < final_arr.length - 1 - i) {
                if (final_arr[j] > final_arr[j + 1]) {
                    int temp = final_arr[j];
                    final_arr[j] = final_arr[j + 1];
                    final_arr[j + 1] = temp;
                }
                j++;
            }
            i++;
```

```
        }
        return final_arr;
    }
}
```

9) Search an element in 2-D array

```
public class SearchingIn2d {
    public static void main(String[] args) {
        int[][] arr = { { 1, 2, 3 }, { 4, 5, 6 } };
        int se = 2;
        int i = 0;
        int di = 0, dj = 0;
        while (i < arr.length) {
            int j = 0;
            while (j < arr[0].length) {
                if (arr[i][j] == se) {
                    di = i;
                    dj = j;
                    break;
                }
                j++;
            }
            i++;
        }
        if (di == 0 && dj == 0)
            System.out.println("ele not found");
        else
            System.out.println("ele found at [" + di + "][" + dj + "]");// ele found at [0][1]
    }
}
```

10) Sum and avg. All the elements in 2-D array

```
public class Sumandavgof2d {
    public static void main(String[] args) {
        int[][] arr = { { 1, 2, 3 }, { 4, 5, 6 } };
        int i = 0;
        int sum = 0;
        while (i < arr.length) {
            int j = 0;
            while (j < arr[0].length) {
                sum += arr[i][j];
                j++;
            }
            i++;
        }
        float avg = (float) sum / (arr.length * arr[0].length);
        System.out.println("sum: " + sum + "\nAvg: " + avg);
    }
}
// sum: 21
// Avg: 3.5
```

11) Sum of 2 diagonal elements

```
public class Sumofdiagonal {
    public static void main(String[] args) {
        int[][] arr = { { 1, 2 }, { 7, 10 } };
        int i = 0, digSum = 0;
        while (i < arr.length) {
```

```
            int j = 0;
            while (j < arr[0].length) {
                if (i == j)
                    digSum += arr[i][j];
                j++;
            }
            i++;
        }
        System.out.println("Diagonal Sum: " + digSum); // Diagonal Sum: 11

    }
}
```

In Java, reference types are types that refer to objects, not actual values. They store memory addresses (references) of the objects.

```
public class Temp {
    public static void main(String[] args) {
        int[] arr = { 1, 2, 3 };
        System.out.println(arr); // [I@2c7b84de
    }
}
```

6. Classes & objects

6.1) (Process/Function/Procedure oriented) Vs. Object oriented

Procedure-Oriented Programming (POP):

A programming approach where the focus is on writing **functions/procedures** that operate on **data**. The program is divided into functions. Example is

A **recipe book** – each step (procedure) tells what to do with the ingredients (data). All steps can access the same ingredients.

Object-Oriented Programming (**OOP):**

A programming style that focuses on **objects**, which are instances of **classes**. Each object contains **data (fields)** and **behaviors (methods)**.Example is

A **car** is an object.

It has **properties** like color, speed, model (data).

It has **behaviors** like drive(), brake(), honk() (methods).

You can create many **car objects** from one **Car class**, each with its own data.

6.2) Instance variables and methods

Properties(variables) of the class which are declared inside the the class but outside any method are know to be the instance variables.

Non-static methods inside a class is known as an instance method.

```
public class Person {
    String name;  // instance variable
    int age;      // instance variable

    // instance method
    void greet() {
        System.out.println("Hello, my name is " + name);
    }
}
```

Instance variables and instances aren't same.Instances are objects of class whereas instance variables are declared inside a class outside a method.

A simple analogy is given below,

Class → Blueprint

Object → A house that was built from blueprint

Instance variables → Rooms inside that house.

```
public class Car {
    // instance variables
    int noOfWheels;
    String color;
    float maxSpeed;
    float currentFuelInLit;
    int noOfSeats;

    // instance methods
    public void drive() {
        System.out.println("Driving....");
```

```
        currentFuelInLit--;
    }
    public void addFuel(float fuel) {
        currentFuelInLit += fuel;
    }
    public float currentFuelLevel() {
        return currentFuelInLit;
    }
}
```

6.3) Declaring an object

New keyword is used to instantiate/create an object of a class.

Class declaration doesn't takes up the memory but when the object is created the memory will be allocated for the object in the heap.

```
public class Driver {
    public static void main(String[] args) {
        Car c1; // defining an reference variable for the car class
        // new Car() is the object
        c1 = new Car(); // storing the address of the object in c1,now c1 is known to be the
                        reference variable which stores the address of the object.
        // Car()--> is known as the constructor.
    }
}
```

New keyword unlocks the dynamic allocation feature which means that memory is created at runtime/while execution by the JVM.

Constructor is called to initialize the object (car() is the constructor used to create an object c1).

C1 stores the reference of the object.

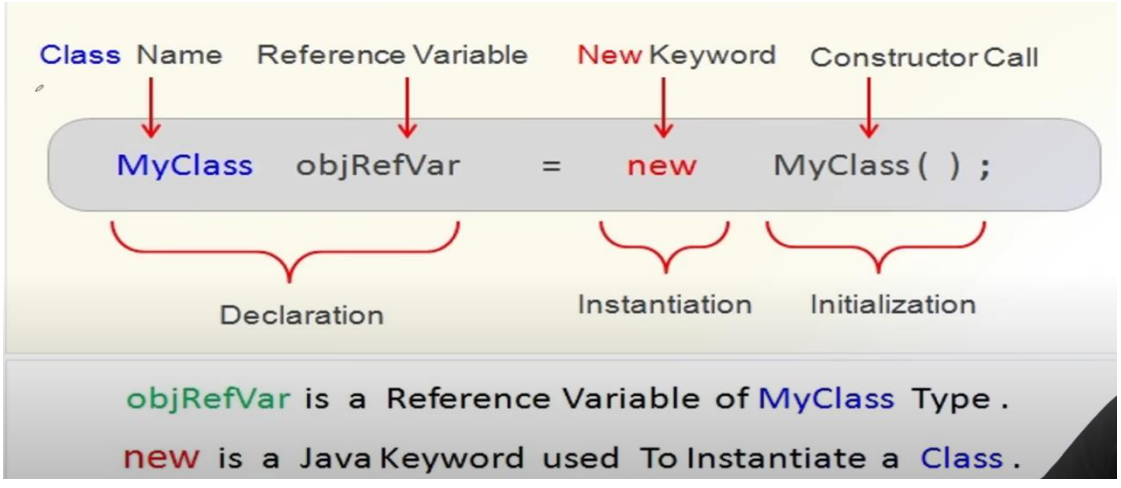
```
System.out.println(c1);
//Car@3fee733d
```

new Car() ⟶ it is the actual object.

C1 ⟶ it is the reference variable used to store the object address.

Better to call c1 as reference variable rather than calling it as an object as it is not an object actually.

Syntax as follows



Dot operator(.) is used to access the members of the class through reference variable.

```
ppublic class Driver {
    public static void main(String[] args) {
        Car c1;
```

```
        c1 = new Car();
        System.out.println(c1.currentFuelInLit);// 0.0
        c1.drive(); // Driving....
        System.out.println(c1.color); // null
    }
}
```

Instance variables on declaration stores default values where as the local variables(members of a method) need to be initialized first in order to use them.

6.4) Class vs. Object

Class is a blueprint and object is the real value.Take an example house blueprint and real house.

“House blueprint doesn’t occupy some part of land but real house occupies the real land”.This statement justifies the 2nd point of 6.3

6.5) this keyword

this keyword is used only within the class to refer to the current object instance of the class.

Refers to the current class reference variable simply this refers to the current object.

It is used to access instance variables and methods from within the class i.e, if the class itself want to access its own instance variables and methods.

Chaining is shown below

```
public class Car {
    // instance variables
    int noOfWheels;
    String color;
    float maxSpeed;
    float currentFuelInLit;
    int noOfSeats;

    // instance methods
    public Car start() {
        if (currentFuelInLit == 0)
            System.out.println("No fuel,ca't start");
        else if (currentFuelInLit < 5)
            System.out.println("In reserve mode, Refeul needed!");
        else
            System.out.println("Starting.......");
        return this;
    }
    public void drive() {
        System.out.println("Driving....");
        currentFuelInLit--;
    }
    public void addFuel(float fuel) {
        currentFuelInLit += fuel;
    }
    public float currentFuelLevel() {
        return currentFuelInLit;
    }
}
```

```
public class Driver {
    public static void main(String[] args) {
        Car swift = new Car();
        swift.addFuel(7);
        Car startedCar = swift.start();
```

```
        startedCar.drive();
    }
}
// Car startedCar = swift.start();
// startedCar.drive();
// the above 2 statements can be replaced with swift.start().drive() and everything works fine.
```

this() → used to invoke a constructor of the same class.

This can be passed as an argument in the method.

```
class Helper {
    void display(Person p) {
        System.out.println("Name: " + p.name);
        System.out.println("Age: " + p.age);
    }
}

class Person {
    String name;
    int age;
    static Helper helper = new Helper();
    Person(String name, int age) {
        this.name = name;     // this refers to instance variable
        this.age = age;       // same here
        helper.display(this);  // pass current object to display
    }
}
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 25);
    }
}
```

This also returns the current class instance.

It's ok for a class to not have a main method, as we can see the class car it can not be executed directly but they can be used inside another class(Driver) of main method to execute.

```
public class Car {
   float currentFuelInLit;
        public void addFuel(float currentFuelInLit) {
                currentFuelInLit += currentFuelInLit;
    }
}
```

In the above case, the parameter in the () has the high priority than the instance variable. So that in the addFuel() function definition, the argument passed to that function will be added to itself which is invalid.so that,

```
public class Car {
    float currentFuelInLit;
    public void addFuel(float currentFuelInLit) {
            this.currentFuelInLit += currentFuelInLit;
        }
    }
```

Now the this.currentFuelLimit refers to instance variable which is declared inside a class and outside any method but not refer to the parameter of the addFuel() method.

Static keyword

Local variables are declared inside a method and their scope is within that method only.

Static methods can not access non-static members.

Class need not be instantiated to call static methods if only within the class.

static variables and methods are accessible anywhere inside the same class.

Static members can also be accessed from outside the class, using:

```
ClassName.staticVariable
ClassName.staticMethod()
```

We can access a static method from non-static method in java.

Reference variables are created for a class to access instance variables and instance methods.

.ClassName is used only to access static methods and static variables.

All the below scenarios in the table are to access the members of one class from another class.

Task	Use new (Object)	Use ClassName. (Static)
Access non-static method	✅Yes	❌No
Access static method	❌No	✅Yes
Call constructor	✅Yes	❌No
Shared utility method	❌No	✅Yes

Shared utility method is a static method which is used for common operations like add,sort,etc.,

Below is an example for utility method.

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

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.add(5, 3);  // shared utility method
        System.out.println("Sum: " + result);  // Output: Sum: 8
    }
}
```

A top class can not be declared as static.

```
static class MyClass {  // ❌Error
    // ...
}
```

But a nested class can be declared as static.

```
class Outer {
    static class Inner {
        void display() {
            System.out.println("Inside static nested class");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();  // ✅No need to create Outer object
        obj.display();  // Output: Inside static nested class
    }
}
```

Use OuterClass.NestedClass to access a **static nested class** without creating an instance of the outer class.

Non-static nested class can't be accessed without creating an object for the outer class as well as for the inner class as shown below

```
class Outer {
    class Inner {  // non-static nested class
        void show() {
            System.out.println("Inside non-static inner class");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Without this line, we cannot create an instance of Inner class
        Outer outer = new Outer();                // Must create outer class object
        Outer.Inner inner = outer.new Inner();    // Now, create inner class object
        inner.show();                             // Output: Inside non-static inner class
    }
}
```

Static variables belongs to the class, not individual instances.these variables are shared among all instances of the class.Justification is shown below

```
class Counter {
    static int staticCount = 0; // shared among all
    int instanceCount = 0;      // separate for each object

    void increment() {
        staticCount++;
        instanceCount++;
    }
    void showCounts() {
        System.out.println("Static Count: " + staticCount);
        System.out.println("Instance Count: " + instanceCount);
    }
}
public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.increment();
        c1.showCounts();  // staticCount = 1, instanceCount = 1
        c2.increment();
        c2.showCounts();  // staticCount = 2, instanceCount = 1
    }
}
```

In java static variables are class-level not method-level.which means they can not be declared inside any method but can be declared as a global variable(inside a class,outside a method).

```
void myMethod() {
    static int x = 10; // ❌Compilation error!
}
```

```
class MyClass {
    static int x = 10; // ✅Valid static variable

    void myMethod() {
```

```
        int y = 5; // ✅Valid local variable
    }
}
```

Static methods can be called without creating an object of the class.can only access static variables and other static methods.They can be invoked using *className.staticMethodName()* and static variables through *className.staticVariableName.*

```
class MyClass {
    static int count = 0;

    static void showMessage() {
        System.out.println("Hello from static method!");
    }
}
public class Main {
    public static void main(String[] args) {
        // Accessing static variable
        System.out.println(MyClass.count);
        // Calling static method
        MyClass.showMessage();
    }
}
```

We can access the static variables and methods without *className.* Only from inside the same class where they are defined.

```
class MyClass {
    static int count = 5;

    static void show() {
        // Accessing without class name
        System.out.println(count);
        showMessage();
    }
    static void showMessage() {
        System.out.println("Static method called!");
    }
    public static void main(String[] args) {
        show(); // Also valid: MyClass.show();
    }
}
```

Static methods can not access non-static members directly of the class, but can be accessed through an object.

```
class MyClass {
    int x = 5; // non-static

    static void show() {
        System.out.println(x); // ❌Error: non-static variable x cannot be referenced from a
static context
    }
}
```

```
class MyClass {
    int x = 5;

    static void show() {
        MyClass obj = new MyClass(); // create an object
        System.out.println(obj.x);   // ✅Access non-static variable via object
    }
```

```
}
```

We can not access non-static members from static methods directly but we can access static members form non-static methods directly.

```
class Demo {
    static int a = 10;      // static
    int b = 20;             // non-static

    static void staticMethod() {
        // System.out.println(b); ❌Invalid
        Demo obj = new Demo();
        System.out.println(obj.b);  // ✅Valid via object
    }
    void nonStaticMethod() {
        System.out.println(a);  // ✅Valid: static can be accessed directly
    }
}
```

6.6) Constructor

Every class has a default constructor provided by the JVM if no constructor is explicitly defined.

However, if you add your own constructor, the JVM does not add the default constructor automatically.

constructors initialize new objects and sets state for the object's attributes.

Name of the constructor must have the same name as the class.

There will be no return type for a constructor.

constructors are automatically called at the time of the object creation. No need to call them explicitly.

A class can have multiple constructors — this is called constructor overloading, but the constructors must have different parameter lists.

```
public class Cars {
    String color;
    float price;

    Cars() { // default constructor
        color = "black";
        price = 500000;
    }
    public static void main(String[] args) {
        Cars swift = new Cars();
  System.out.println(swift.color); //black
    }
}
```

From above, “black” and 500000 are the initial values. anyways you can modify them later.

There will be 2 types of constructors

1. Default constructor - If no constructor is explicitly defined, java provides a default constructor that initializes all the member variables to default values.
2. parameterised constructor - constructors can have parameters to pass the values when creating an object.

```
public class Cars {
    String color;
    float price;

    Cars(String color) { // parameterised constructor
        this.color = color;
```

```
        price = 500000;
    }

    public static void main(String[] args) {
        Cars swift = new Cars("red");
        System.out.println(swift.color); // red
    }
}
```

constructor chaining/constructor overloading is shown below

```
public class Cars {
    String color;
    float price;

    Cars(String color) {
        this.color = color;
        price = 500000;
    }

    Cars() {
        color = "black";
        price = 500000;
    }

    public static void main(String[] args) {
        Cars swift = new Cars("red");
        System.out.println(swift.color); // red
        Cars alto = new Cars();
        System.out.println(alto.color); //black
    }
}
```

the above can also be done with this below simple code.

```
public class Cars {
    String color;
    float price;

    Cars(String color) {
        this.color = color;
        price = 500000;
    }

    Cars() {
        this("black"); // calling its own constructor
    }

    public static void main(String[] args) {
        Cars swift = new Cars("red");
        System.out.println(swift.color); // red
        Cars alto = new Cars();
        System.out.println(alto.color); // black
    }
}
```

below are some key points of constructor chaining

only within the same class - use this() to call another constructor from the same class

this() should be the 1st statement.

constructor chaining should not form a loop, and must be terminated at some point.

```
public class Demo {
    Demo() {
        this(10); // calls Demo(int)
        System.out.println("Default constructor");
    }

    Demo(int x) {
        this(); // calls Demo()
        System.out.println("Parameterized constructor");
```

```
    }
}
```

below will also be an example of the infinite looping

```
 Demo(int x) {
        this(10);
 }
```

Code Blocks {}

There are 2 types of code blocks

1) Instance initialization block - runs every time when an object is created, even before the execution of the constructor. For sharing common code among all the constructors.
2) static initialization block - static block is executed only once, and that's when the class is loaded into memory for the first time, even before the main() method or any object is created. used to initialize static variables.

```
public class Example {
    static {
        System.out.println("Static block");
    }

    {
        System.out.println("Instance block");
    }

    Example() {
        System.out.println("Constructor");
    }

    public static void main(String[] args) {
        new Example();
        /*Static block
        *Instance block
        *Constructor
        */
        new Example();
        /*Instance block
         *Constructor
         */
    }
}
```

So even if you create 100 objects, the static block still runs only once.

If the class is never referenced, the static block won't run. The example is shown below

```
public class Test {
    static {
        System.out.println("Static block in Test");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Main method only"); // Main method only
    }
}
```

Static block will not be executed in the above code.

```
public class Main {
    public static void main(String[] args) {
        Test.doSomething(); // Now we're referencing the class
    }
}

class Test {
    static {
        System.out.println("Static block in Test");
```

```
    }

    static void doSomething() {
        System.out.println("Doing something");
    }
}

// Static block in Test
// Doing something
```

static block will be executed in the above code.

scope of the variables that are initialized inside these blocks/local variables can not be accessed outside of the block. Example is if we declare int i=0 in a for loop, then the scope of that variable will be in that for loop only.

CHALLENGES

1) create a Book class for a library system

Title,author,isbn as instance variables

totalBooks as a static variable

borrowBook(),returnBook() as instance methods

getTotalBooks() as static method.

```
public class Book {

    static int totalBooks;
    String title, author, isbn;
    boolean isBorrowed;
    static {
        totalBooks = 0;
    }
    {
        totalBooks++;
    }
    Book(String title, String author, String isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
    }
    Book(String isbn) {
        this("Unknown", "Unknown", isbn);
    }
    public void borrowBook() {
        if (isBorrowed)
            System.out.println("Book already taken");
        else
            isBorrowed = true;
    }
    public void returnBook() {
        if (isBorrowed)
            isBorrowed = false;
        else
            System.out.println("Already in the pool");
    }
    public static void getTotalBooks() {
        System.out.println(totalBooks);
    }
    public static void main(String[] args) {
        Book b1 = new Book("c by ansi", "payal", "20SSH");
```

```
        Book b2 = new Book("2024SSH");
        getTotalBooks(); // 2
        b2.borrowBook();
        b1.borrowBook();
        b1.returnBook();
        getTotalBooks(); // 2
        b2.returnBook();
        getTotalBooks(); // 2
        b2.returnBook(); // Already in the pool
    }
}
```

6.7 Satck Vs. Heap Memory

Memory is of 2 types

1) **Stack** - ordered on top of each other, where method invocations and local variables live.

Those method invocations are known as the call stack.

Follows static memory allocation - the size and lifetime of variables are determined when the method is called.

all the variables declared inside the methods live in that allocated memory of the method in the stack.

2) **Heap** - No particular order, where all the objects live.

Follows dynamic memory allocation - memory is allocated at the runtime.

When an object is created,

Class c1=new Class();

new Class() is the object which will be stored in the heap, where as the c1 is the reference of the type Class will be stored in the stack.

6.8 Primitive vs. Reference Types

Data types in java = Primitive + Non-Primitive

Primitive = Numeric(integer,floating point numbers) + Non-numeric(characters,boolean)

Non-Primitive = Class, Array, String …

- We know that references are stored in the stack, the amount of space occupied in the memory will be 4 bytes as it stores the address.
- Accessing Primitive types is faster than the non-primitive types. That's why C is faster than java as it doesn't contains oops concepts.
- Primitives stores the actual values where as reference types stores the addresses of the objects.
- Primitive data types are stored in the stack, whereas non-primitive in the heap.
- Comparing can be done by using == for primitive data types whereas .equals() is used to compare reference types.

6.9 Variable Scopes

A variable's scope is limited to the block {} where it is declared.

Instance variables are those declared inside a class but outside any method. Their scope spans the entire class. Also known as the global variables.

Local variables, which are declared inside a method or loop, are limited to that specific method or loop only.

6.10 Garbage collection & Finalize

In C, dynamically allocated memory must be manually deallocated using free(), or it causes memory leaks.

In Java,

- memory is managed automatically
- unused objects are cleared by the Garbage Collector which is managed by the JVM.
- No manual control - we can't deallocate memory manually.
- GC can affect the application performance.
- GC occurs in heap memory, where all the objects reside.

1. Generational Garbage Collection in Java means the heap is divided into parts based on how long objects live.Young Generation: Where new objects go. Most of them die quickly and are cleaned often.
2. Old Generation: Stores long-living objects that survived multiple cleanups.
3. Permanent Generation/Metaspace (Java 8+): Stores class-related info, not regular objects.Memory cleanup frequency is very low on comparison with the other 2 generations.

finalize()

- finalize() is a method that belongs to the Object class.
- It is called by the Garbage Collector before an object is destroyed.
- You can override it in your class to write cleanup code.

```
@Override
protected void finalize() throws Throwable {
    System.out.println("Object is being garbage collected");
}
```

- finalize() is not reliable.
- It has been deprecated in Java 9 and removed in later versions.

KEY POINTS

- Oops concerns about data not the logic where as method oriented programming languages like c, concerns about logic rather than data.
- Instance methods/non-static methods in java can not be called without creating an object of the class.
- Class is a template used to create objects.

- Static methods of the class can be called without creating an object.
- new keyword is used to declare an array.
- Variables declared inside a method are called local variables not instance variables.
- We can not access instance variables through static methods directly, but can be through object creation.
- Each object has its own copy of instance variables, but static variables are shared across all instances of the class.
- In java any thing declared with the static keyword belongs to the whole class but not belongs to the every individual object.
- Variable scope is not defined at run-time but can be known at the compile-time on where the variable is declared in the code block.

7. Control statements, Math & String

7.1) Ternary Operator

Variable=condition?expression1:expression1;

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int a, b, max = 0;
        System.out.print("Enter first number: ");
        a = s.nextInt();
        System.out.print("Enter second number: ");
        b = s.nextInt();
        max = a > b ? a : b;
        System.out.println("Max of " + a + " and " + b + " is " + max);
    }
}
// Enter first number: 10
// Enter second number: 20
// Max of 10 and 20 is 20
temp = (a > b ? a : b) > c ? (a > b ? a : b) : c; // to find max of 3
```

7.2) Switch

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int num;
        System.out.print("Enter the week number: ");
        num = s.nextInt();
        switch (num) {
            case 1:
                System.out.println("Sunday");
                break;
            case 2:
                System.out.println("Monday");
                break;
            case 3:
                System.out.println("Tuesday");
                break;
            case 4:
                System.out.println("wednesady");
                break;
            case 5:
                System.out.println("thursday");
                break;
            case 6:
                System.out.println("friday");
                break;
            case 7:
                System.out.println("saturday");
                break;
            default:
                System.out.println("you enetred num>7");
                break;
```

```
        }
    }
}
// Enter the week number: 7
// saturday
```

Usage of enhanced switch(after java12) was shown below

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int num;
        System.out.print("Enter the week number: ");
        num = s.nextInt();
        String out = switch (num) {
            case 2 -> "Monday";
            case 3 -> "Tuesday";
            case 4 -> "Wednedday";
            case 5 -> "Thursday";
            case 6 -> "Friday";
            case 7, 1 -> "Holiday";
            default -> "invalid(num>7)";
        };
        System.out.println(out);
    }
}
// Enter the week number: 7
// Holiday
```

The above can also be implemented through traditional switch using fall-through.

Fall-through occurs when multiple case labels share the same block of code because there's no break statement between them.

```
int day = 1;

switch (day) {
    case 1:
    case 2:
    case 3:
        System.out.println("Weekday");
        break;
    case 4:
    case 5:
        System.out.println("Almost Weekend");
        break;
    case 6:
    case 7:
        System.out.println("Weekend");
        break;
    default:
        System.out.println("Invalid day");
}
// Weekday
```

CHALLENGES

1) Min of 2 numbers

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        int a = 5, b = 6, min;
```

```
        min = a < b ? a : b;
        System.out.println("min:" + min); // min:5
    }
}
```

2) Even or odd

```
public class Chapter7 {
    public static void main(String[] args) {
        int a = 5;
        String res = (a % 2 == 0) ? "Even" : "Odd";
        System.out.println(res);
        // Odd
    }
}
```

3) Absolute of an integer

```
public class Chapter7 {
    public static void main(String[] args) {
        int a = -605;
        int res = a >= 0 ? a : -a;
        System.out.println(res);
        // 605
    }
}
```

4) Score categorize

High if >80

Moderate if 50-80

Low if <50

```
public class Chapter7 {
    public static void main(String[] args) {
        int score = 49;
        String res = score > 80 ? "High" : (score < 50 ? "Low" : "MOderate");
        System.out.println(res);
        // Low
    }
}
```

5) Display month of the year

```
public class Chapter7 {
    public static void main(String[] args) {
        int month_num = 10;
        String month = switch (month_num) {
            case 1 -> "Jan";
            case 2 -> "Feb";
            case 3 -> "Mar";
            case 4 -> "Apr";
            case 5 -> "May";
            case 6 -> "Jun";
            case 7 -> "Jul";
            case 8 -> "Aug";
            case 9 -> "Sep";
```

```
            case 10 -> "Oct";
            case 11 -> "Nov";
            case 12 -> "Dec";
            default -> "Invalid input";
        };
        System.out.println(month);
    }
}
```

6) Simple calculator using switch

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int a, b;
        String whatToDo;
        System.out.print("Enter first num: ");
        a = s.nextInt();
        System.out.println("Enter second num: ");
        b = s.nextInt();
        System.out.print("What to do?(sum/sub/mul/div): ");
        whatToDo = s.next();
        int out = switch (whatToDo) {
            case "sum" -> a + b;
            case "sub" -> a - b;
            case "mul" -> a * b;
            case "div" -> a / b;
            default -> 0;
        };
        System.out.println(out);
    }
}
// Enter first num: 5
// Enter second num:
// 25
// What to do?(sum/sub/mul/div): sum
// 30
```

7.3) Loops

Do-While

- First execute the block then check the condition.
- Guarantees to execute atleast once.
- Unlike while, first iteration is unconditional.
- Update the loop to avoid infinite loop.

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int age;
        do {
            System.out.print("Enter your age: ");
            age = s.nextInt();
        } while (age < 0 || age > 100);
```

```
        System.out.println("you age is " + age);
    }
}
// Enter your age: -7
// Enter your age: 200
// Enter your age: 101
// Enter your age: 48
// you age is 48
```

Above is a program that asks the user repeatedly until he enters correct age.

For loop

- Standard and most used loop.
- Syntax:

  for(initialization;condition;updation){

  // body of for loop

  }

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        for (int i = 0; i <= 100; i++)
            System.out.println(i);
    }
}
```

Repeated age input using for loop

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int age;
        for (age = -1; age < 0 || age > 100;) {
            System.out.print("Enter age: ");
            age = s.nextInt();
        }
        System.out.println("your age is " + age);
    }
}
// Enter age: 1000
// Enter age: -25
// Enter age: 150
// Enter age: 20
// your age is 20
```

Enhanced-for loop/For each loop

```
public class Chapter7 {
    public static void main(String[] args) {
        int[] arr = new int[] { 1, 2, 3, 4, 5 };
        for (int i : arr)
            System.out.println(i);
    }
```

```
}
// 1
// 2
// 3
// 4
// 5
```

.length → used to find the length of an array.

.length() → used to find the length of an string.

7.4) Using Break & Continue

Break → stops a loop early / breaks out of the loop.

Continue → skips one iteration

```
public class Chapter7 {
    public static void main(String[] args) {
        int i;
        for (i = 0; true; i++) {
            if (i % 2 == 0)
                continue;
            else if (i == 51)
                break;
            System.out.print(i + " ");
        }
        System.out.println("\nBreaked out of the loop with i value " + i);
    }
}
// 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
// Breaked out of the loop with i value 51
```

7.5) Recursion

Function calling itself is known as recursion.

Ideal for problems divisible into smaller and smaller problems.

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number which you want to find the factorial: ");
        int num = s.nextInt();
        System.out.println("Factorial of " + num + " is " + factorial(num));
    }
    public static long factorial(int num) {
        if (num == 0 || num == 1)
            return 1;
        else
            return num * factorial(num - 1);
    }
}
// Enter the number which you want to find the factorial: 5
// Factorial of 5 is 120
```

CHALLENGES

1) Use a do-while loop to find password checker until a valid password is entered.

```java
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String pass = "Java@123", passCheck;
        do {
            System.out.print("Enter the password: ");
            passCheck = s.next();
        } while (!pass.equals(passCheck));
        System.out.println("You have entered the correct password!");
    }
}
// Enter the password: java@123
// Enter the password: Java@123
// You have entered the correct password!
```

We can't compare 2 strings using == operator as it compares the address stores in the reference variable not the actual value.

2) Number guessing game

```java
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int num = 7, check;
        do {
            System.out.print("Guess the num(1-10): ");
            check = s.nextInt();
        } while (num != check);
        System.out.println("You have guessed it right!");
    }
}
// Guess the num(1-10): 5
// Guess the num(1-10): 4
// Guess the num(1-10): 6
// Guess the num(1-10): 1
// Guess the num(1-10): 9
// Guess the num(1-10): 7
// You have guessed it right!
```

3) Multiplication table for a number using for-loop

```java
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Multiplication table for what number: ");
        int num = s.nextInt();
        for (int i = 1; i <= 10; i++) {
            System.out.println(num + " * " + i + " = " + (num * i));
        }
    }
}
```

```
// Multiplication table for what number: 10
// 10 * 1 = 10
// 10 * 2 = 20
// 10 * 3 = 30
// 10 * 4 = 40
// 10 * 5 = 50
// 10 * 6 = 60
// 10 * 7 = 70
// 10 * 8 = 80
// 10 * 9 = 90
// 10 * 10 = 100
```

4) Prime or not using for loop

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a num: ");
        int num = s.nextInt(), cnt = 0;
        for (int i = 1; i <= num; i++) {
            if (num % i == 0)
                cnt++;
        }
        if (cnt == 2)
            System.out.println(num + " is Prime");
        else
            System.out.println(num + " is not prime");
    }
}
// Enter a num: 5
// 5 is Prime
```

5) Max of integer array using for-each loop

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        int max = 0;
        int[] arr = { 1, 2, 3, 4, 254, 5, 6, 7, 8 };
        for (int i : arr) {
            if (i > max)
                max = i;
        }
        System.out.println("Max of the array is " + max);
    }
}
// Max of the array is 254
```

6) Occurrence of a specific element in an array using for-each

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int cnt = 0;
        int[] arr = { 1, 2, 3, 4, 4, 254, 5, 6, 7, 8, 1, 5, 2, 8, 11 };
        System.out.print("Enter the number to find the no.of occurences in the array: ");
        int find = s.nextInt();
```

```
        for (int i : arr) {
            if (i == find)
                cnt++;
        }
        System.out.println(find + " occured " + cnt + " times!");
    }
}
// Enter the number to find the no.of occurences in the array: 5
// 5 occured 2 times!
```

7) Read input from the user in a loop until they enter “exit”

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String str;
        for (int i = 0; true; i++) {
            System.out.print("Enter the magic word: ");
            str = s.next();
            if (str.equals("exit"))
                break;
        }
        System.out.println("Hurrayyyyy, Doors opened for youhhhhhh!!!!");
    }
}
// Enter the magic word: abrakadabra
// Enter the magic word: bushhhhh
// Enter the magic word: exit
// Hurrayyyyy, Doors opened for youhhhhhh!!!!
```

8) Sum all the +ve numbers entered by the user and skip -ve numbers

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int temp, sum = 0;
        for (int i = 1; i <= 10; i++) {
            System.out.print("Enter number " + i + ": ");
            temp = s.nextInt();
            if (temp < 0)
                continue;
            else
                sum += temp;
        }
        System.out.println("+ves sum = " + sum);
    }
}
// Enter number 1: 5
// Enter number 2: 7
// Enter number 3: 1
// Enter number 4: 2
// Enter number 5: 3
// Enter number 6: 4
// Enter number 7: -5
// Enter number 8: -7
// Enter number 9: -2
// Enter number 10: 0
```

```
// +ves sum = 22
```

9) Print only even numbers.

```
public class Chapter7 {
    public static void main(String[] args) {
        for (int i = 0; i <= 50; i++) {
            if (i % 2 == 1)
                continue;
            else
                System.out.print(i + " ");
        }
    }
}
// 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
```

10) Fibonacci series upto specified number using recursion

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number upto which you want to print fibonacci: ");
        int cnt = s.nextInt();
        for (int i = 0; i < cnt; i++) {
            System.out.print(fibonacci(i) + " ");
        }
    }
    public static int fibonacci(int num) {
        if (num <= 1)
            return num;
        return fibonacci(num - 1) + fibonacci(num - 2);
    }
}
// Enter the number upto which you want to print fibonacci: 7
// 0 1 1 2 3 5 8
```

Integer.toString() → used to convert an integer to string.

11) String is palindrome or not, using recursion.

```
import java.util.Scanner;

public class Chapter7 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the String: ");
        String str = s.next();
        System.out.println("The string you have entered " + str + " was " +
                (palindromeChecker(str) == true ? "Palindrome"
                        : "not a palindrome"));
    }
    public static boolean palindromeChecker(String str) {
        if (str.length() == 1)
            return true;
        if (str.charAt(0) != str.charAt((str.length() - 1)))
            return false;
        String newStr = str.substring(1, str.length() - 1);
        return palindromeChecker(newStr);
    }
}
// Enter the String: laaaaaaaaal
```

```
// The string you have entered laaaaaaaaal was Palindrome
```

7.6) Random numbers & Math class

Math methods are static and can be accessed directly.

Key methods are

- Math.abs()
- Math.ceil() ⟶ Always rounds *up* to the nearest integer, regardless of the decimal part.
- Math.floor()
- Math.round() ⟶ Rounds to the nearest integer.If the decimal is 0.5 or more, it rounds up.If the decimal is less than 0.5, it rounds down.
- Math.max(a,b)
- Math.min(a,b)
- Math.pow(base,exponent)
- Math.sqrt()
- Math.random() ⟶ generates a random number between 0 and 1
- Math.exp(),Math.log()

```java
public class Chapter7 {
    public static void main(String[] args) {
        System.out.println(Math.abs(-20)); // 20
        System.out.println(Math.ceil(0.4)); // 1.0
        System.out.println(Math.round(0.4)); // 0
        System.out.println(Math.floor(5.5)); // 5.0
        System.out.println(Math.random()); // 0.5910423373540226
        System.out.println(Math.PI); // 3.141592653589793
        System.out.println((int) (Math.random() * 100)); // 80
    }
}
```

7.7) toString method

- Provides the string representation of an object.
- It is inherited form the object class.
- By default it returns object_name@hash
- Can be overriden for the meaningful output.

```java
public class Home {
    int noOfdoors;
    int noOfwindows;
    String hName;

    public Home(int doors, int windows, String hName) {
        this.noOfdoors = noOfdoors;
        this.hName = hName;
        this.noOfwindows = noOfwindows;
    }
    public static void main(String[] args) {
        Home villa = new Home(8, 10, "vaikuntapuram");
        System.out.println(villa.toString()); // Home@2c7b84de
        System.out.println(villa); // Home@2c7b84de
```

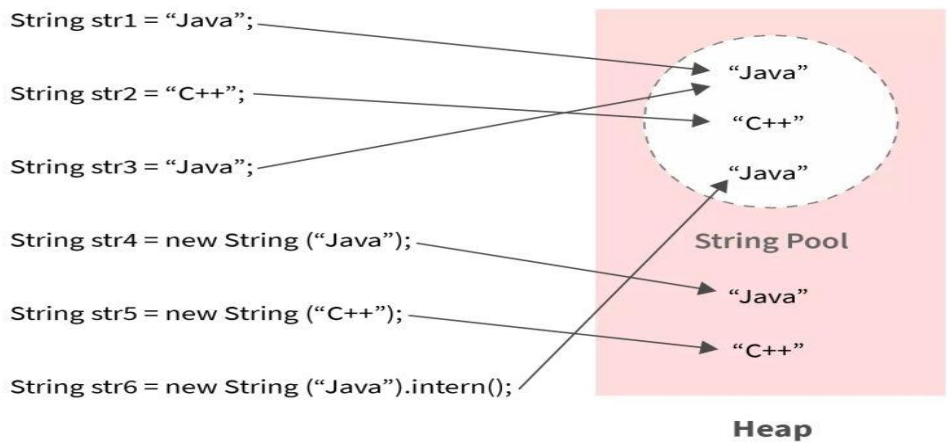
```
    }
}
```

If you override the existing toString() method in java, the code is as follows

```
public class Home {
    int noOfdoors;
    int noOfwindows;
    String hName;

    public Home(int doors, int windows, String hName) {
        this.noOfdoors = noOfdoors;
        this.hName = hName;
        this.noOfwindows = noOfwindows;
    }
    @Override
    public String toString() {
        return ("you are in " + this.hName);
    }
    public static void main(String[] args) {
        Home villa = new Home(8, 10, "vaikuntapuram");
        System.out.println(villa.toString()); // you are in vaikuntapuram
        System.out.println(villa); // you are in vaikuntapuram
    }
}
```

7.8) String class

- String objects are immutable.
- Java maintains a pool of strings for efficiency.
- References like str1,str2 stores in the stack whereas the actual string content is stored in the heap.
- String object created with new keyword was stored inside the heap but outside the string pool A String created using a string literal (e.g., "Java") is stored inside the String Pool.
- Use .equals() for content comparison and == for object reference comparison I.e, whether the two references points to the same memory location.
- Memory for objects is allocated on the heap at runtime, whereas stack memory is allocated at the time of function call execution.



```java
public class StringTest {
    public static void main(String[] args) {
        String str1 = "Java";                 // stored in string pool
        String str2 = "Java";                 // reuses str1 from string pool
        String str3 = new String("Java");     // new object in heap
        String str4 = str3.intern();          // refers to object in pool

        // Comparing references
        System.out.println("str1 == str2: " + (str1 == str2));   // true (same reference)
        System.out.println("str1 == str3: " + (str1 == str3));   // false (different objects)
        System.out.println("str1 == str4: " + (str1 == str4));   // true (interned reference)
        // Comparing values
        System.out.println("str1.equals(str3): " + str1.equals(str3)); // true (same content)
        System.out.println("str3.equals(str4): " + str3.equals(str4)); // true (same content)
    }
}
```

- Can be concatinated using + operator.
- Some of the methods of strings are length(), subString(), equals(), compareTo(), indexOf()

  compareTo() compares the 2 strings lexicographically.

  Return type of compareTo() function was int.

  int res=Str1.compareTo(Str2);

  0 if both are same, +ve if str1>str2, -ve if str1<str2.
- In array .length is a field where as in string .length() is a method.
- String uses more memory if you are modifying it frequently.

Formatted printing just like in c is possible with java also.

```java
public class Chapter7 {
    public static void main(String[] args) {
        String name = "Hemanth";
        int marks = 75;
        System.out.printf("Hey %s!, you have got %d marks", name, marks); // Hey Hemanth!, you
have got 75 marks
    }
}
```

Above practice takes up the less memory compared to traditional string concatenating println() method.

7.9) StringBuffer vs. StringBuilder

StringBuilder is commonly used because most applications operate in a single-threaded environment, where performance is more important than thread safety.

```java
public class Chapter7 {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Original line. ");
        sb.append("appended line 1. ");
        sb.append(52).append(" is your age");
        // No need to explicitly call sb.toString() in System.out.println()
        System.out.println(sb); // Java automatically calls sb.toString()
    }
}
// Original line. appended line 1. 52 is your age
```

Feature	String	StringBuilder	StringBuffer
Mutable	❌No	✅Yes	✅Yes
Thread-Safe	✅Yes	❌No	✅Yes
Performance	❌Slow	✅Fastest	Slower than Builder
Best For	Read-only	Single-threaded	Multi-threaded

7.10) Final Keyword

- Name of the final variables should be of capital letters and should follow snake case naming convention.
- When final is applied to a variable, it becomes a constant.
- It must be initialized at the time of declaration or within the constructor (for instance variables).
- Once assigned, its value cannot be changed — making it immutable.

```
public class Chapter7 {
    public static void main(String[] args) {
        final int PI_VALUE= 3.14;
        PI_VALUE= 5;
        System.out.println(PI_VALUE);
       // error: cannot assign a value to final variable PI_VALUE
    }
}
```

CHALLENGES

1) Define a student class with fields name,age and print details using toString.

```
public class Student {
    String name;
    int age;

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        return sb.append(name).append("!, you are of age ").append(age).toString();
    }
    public static void main(String[] args) {
        Student s1 = new Student("Hemanth", 20); // Hemanth!, you are of age 20
        System.out.println(s1);
    }
}
```

2) Concat 2 strings and convert the result into uppercase.

```
import java.util.Scanner;

public class Student {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        StringBuilder sb = new StringBuilder();
        System.out.print("Enter your first name: ");
        String first_name = s.next();
        System.out.print("Enter your last name: ");
```

```
        String last_name = s.next();
        sb.append(first_name).append(" ").append(last_name).toString();
        System.out.printf("Your name is: %S", sb); // %S caps(s) is used to convert into
capitals
    }
}
// Enter your first name: Hemanth
// Enter your last name: Muvvala
// Your name is: HEMANTH MUVVALA
```

OR

```
public class Student {
    public static void main(String[] args) {
        String fname = "Hemanth", lname = " Muvvala";
        System.out.println(fname.concat(lname).toUpperCase()); // HEMANTH MUVVALA
        System.out.println(fname); // Hemanth
        System.out.println(lname); // Muvvala
        // above 2 print statements shows that strings are immutable.
    }
}
```

3) Find area and circumference of a circle for a given radius using Math.PI

```
import java.util.Scanner;

public class Student {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the radius of the circle: ");
        double r = s.nextFloat(), circumference, area;
        circumference = 2 * Math.PI * r;
        area = Math.PI * Math.pow(r, 2);
        System.out.printf("Area = %.3f \ncircumference = %.3f", area, circumference);
    }
}
// Enter the radius of the circle: 2
// Area = 12.566
// circumference = 12.566
```

4) Simulate a rolling dice and display the outcome(1-6)

```
import java.util.Scanner;

public class Student {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        char decision;
        do {
            System.out.print("Do you want to roll the dice(y/n): ");
            decision = s.next().charAt(0);
            if (decision == 'n')
                continue;
            else
                System.out.println((((int) (100 * Math.random())) % 6) + 1);
        } while (decision == 'y');
    }
}
// Do you want to roll the dice(y/n): y
// 4
// Do you want to roll the dice(y/n): y
```

```
// 4
// Do you want to roll the dice(y/n): y
// 5
// Do you want to roll the dice(y/n): n
```

OR

```
int dice = (int)(Math.random() * 6) + 1;
```

OR

```
int dice = (int)(Math.ceil(Math.random() * 6));
```

5) Number guessing game where program selects a number and user had to guess.

```
import java.util.Scanner;

public class Student {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int num;
        int sys_num = ((int) (Math.random() * 10)) + 1;
        do {
            System.out.print("Guess the num(1 - 10): ");
            num = s.nextInt();
        } while (num != sys_num);
        System.out.println("you have guessed it right!");
    }
}
// Guess the num(1 - 10): 1
// Guess the num(1 - 10): 2
// you have guessed it right!
```

6) Take an array of words and concatenate them using stringBuilder.

```
public class Student {
    public static void main(String[] args) {
        String arr[] = { "apple", "banana", "potato", "carrot" };
        StringBuilder sb = new StringBuilder();
        for (String str : arr)
            sb.append(str).append(" ");
        System.out.println(sb); // apple banana potato carrot
    }
}
```

7) Create an object with final fields and use constructor to initialize them.

```
public class Student {
    final int age;
    final String name;

    Student(int age, String name) {
        this.age = age;
        this.name = name;
        show(this.age, this.name);
    }
    public static void show(int age, String name) {
        System.out.printf("Hey %S! you are %d years old.\n", name, age);
    }
    public static void main(String[] args) {
        new Student(20, "Hemanth");
```

```
        Student s = new Student(18, "Ramesh");
        //s.age = 21; //Student.java:18: error: cannot assign a value to final variable age
    }
}
// Hey HEMANTH! you are 20 years old.
// Hey RAMESH! you are 18 years old.
```

KEY POINTS

- switch statement is only used to check the equality with constants.
- Math.random() generates values from [0,1)
- stringBuilder is faster than stringBuffer.

8. Encapsulation & Inheritance

8.1) Intro to OOP Principle

Oops principles are not specific to java.

There are 4 oop principles,

1) Encapsulation - only exposing the selected information from an object.
2) Abstraction - Hides the complex details to reduce complexity.
3) Inheritence - Entity can inherit attributes from another entity.
4) Polymorphism - Entities can have more than one form.

8.2) Encapsulation

Class is the combination of variables and methods.

Encapsulation hides internal data and allows access only through methods.

Encapsulation can be implemented in java through these access modifiers public, private, protected.

Getter and setter methods hides the properties and creates methods to expose information.

Maintains integrity by avoiding external interference.

Enhances modularity with the concepts of imports and packages.

8.3) Import and packages

Packages are declared at the beginning of the source file using the package keyword followed by package name.

No 2 classes can have the same name in the same package.

If we have car class in package1 and it is completely valid to have car class in package2 also.

This is how we can use the car class of package2 from package1.

```java
package in.Package2;

import java.util.Scanner;

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

```java
// need to use the fully-qualified-name which is in.Package2.Car inorder to use it if we are not using the import statement.
```

```java
package in.Package1;

public class Car {
    public static void main(String[] args) {
        in.Package2.Car car_class_in_Package2 = new in.Package2.Car();
    }
}
```

File structure was

in/ > (Package1/ > Car.java)& (Package2/ > Car.java)

Packages avoids name collisions.

Packages can be used with the help of import statement.

Using import statement we can import classes, methods, packages and even interfaces.

Types of imports

1) Single_type import - imports a single class or interface from a package.

```
import java.util.Scanner;
```

2) On_demand import - imports all the classes and interfaces form a package.

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

There are 2 types of packages

1) Built-in packages

2) User-defined packages

Some of the built-in packages of java are java.util, java.lang, java.io etc.,

Built-in packages are those packages which we need not to import them explicitly.

8.4) Access modifiers

In Java, access modifiers are used to define the visibility and accessibility of classes, methods, and variables.

4 types of access modifiers

1) Public - Anything declared as public can be accessible form everywhere.
2) Protected - Anything declared as Protected can be accessed with in the same class and within the same package and also from the subclasses even if they are form different packages
3) Default (No modifier) a/o package-private -

   Any thing that wasn't declared (default) can be accessed within the defining class and also across all the classes of that package, members outside the package can not access. That's why it is known as the package private.

   From the above example,

   Files from in.package2 can access one another, and files form in.Package1 can access one another if they haven't declared anything.

   But a class from in.Package1 can not access a class from in.Package2 if it haven't declared public.
4) Private - restricts access in the defined class only. - Any thing that declared as private can be accessible within that class only.

Modifier	Access Level
private	Accessible only within the same class
default *(no modifier)*	Accessible within the same package only
protected	Accessible within the same package and by subclasses
public	Accessible from any other class

A class can be either public/ default but can not be private/ protected.

Access Modifier	Within Class	Within Package	Outside Package by Subclass Only	Outside Package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

As java is a case-sensitive language, it allows “Public” as the class name but not public.

Anything declared as private in Java can be accessed or modified *indirectly* using getter and setter methods, which are usually declared as public and it is a fundamental concept in encapsulation.

```
package in.Package1;
//car.java
public class Car {
    public String name;
    public String model;
    private int costOfPurchase;
    public Car(String name, String model, int costOfPurchase) {
        this.name = name;
        this.model = model;
        this.costOfPurchase = costOfPurchase;
    }
    public Car() {
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        return sb.append("name= ").append(name).append("\nmodel: ").append(model).append("\n
        CostOf Purchase: ").append(costOfPurchase).toString();
    }
}
```

```
package in.Package1;
//AccessTest.java
public class AccessTest {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.name = "suzuki";
        c1.model = "swift";
        // c1.costOfPurchase = 5000; ✕can not access private variables directly.
        System.out.println(c1);
        Car c2 = new Car("Toyota", "Fortuner", 50000);
        System.out.println(c2);
    }
}
```

```
Output:

name= suzuki
model: swift
 CostOf Purchase: 0
name= Toyota
model: Fortuner
 CostOf Purchase: 50000
```

You can read the private field costOfPurchase using the overridden toString() method from another class.You cannot modify it directly from outside the Car class due to its private access.

Other than toString(), An inner class can access private members of the outer class in java.

```
public class Car {
    private int cost = 1000;

    class Engine {
        void showCost() {
            System.out.println("Cost: " + cost);
        }
    }
}
```

8.5) Getter and Setter methods

- Getters are methods used to retrieve the value of a private variable.
- Setters are methods used to modify the value of a private variable.
- They encapsulate the access logic and provide controlled access to private fields of a class.

```
// Car.java
public class Car {
    private String model;
    private int price;

    // Getter for model
    public String getModel() {
        return model;
    }
    // Setter for model
    public void setModel(String model) {
        this.model = model;
    }
    // Getter for price
    public int getPrice() {
        return price;
    }
    // Setter for price
    public void setPrice(int price) {
        if (price > 0) {
            this.price = price;
        }
    }
}
```

```
// Main.java
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();

        // Using setters to assign values
        myCar.setModel("Honda City");
        myCar.setPrice(12000);
        // Using getters to access values
        System.out.println("Model: " + myCar.getModel()); // Model: Honda City
        System.out.println("Price: $" + myCar.getPrice()); // Price: $12000

    }
}
```

If we use parameterized constructor then there will be no use of setter methods.Getter and setter methods are typically declared as public, but they can also have default access if restricted to the same package.

CHALLENGES

1) Create 2 packages com.example.geometry and com.example.utils, in geometry package create classes like circle and rectangle and in utils create a class calculator to compute the area of these shapes.

```
package com.example.geometry;

public class Circle {
    public int radius;
    public Circle(int radius) {
        this.radius = radius;
    }
}
```

```
package com.example.geometry;

public class Triangle {
    public int base;
    public int height;
    public Triangle(int base, int height) {
        this.base = base;
        this.height = height;
    }
}
```

```
package com.example.utils;

import com.example.geometry.Circle;
import com.example.geometry.Triangle;
class Calculator {
    public static void main(String[] args) {
        Circle c1 = new Circle(5);
        Triangle t = new Triangle(5, 4);
        System.out.println("Area: " + Math.PI * Math.pow(c1.radius, 2));
        System.out.println("Area of Triangle: " + (0.5 * t.base * t.height));
    }
}
```

2) Define a BankAccount class with private attributes like accountNumber, accountHolderName, balance. Provide public methods to deposit and withdraw money ensuring that these methods don't allow illegal operations like withdrawing more money than the current balance.

```
public class BankAccount {
    private long accountNumber, balance;
    private String accountHolderName;

    BankAccount(long accountNumber, String accountHolderName) {
        this.accountHolderName = accountHolderName;
        this.accountNumber = accountNumber;
    }
    public void balance() {
        System.out.println("Balance: " + this.balance);
    }
    public void deposit(long money) {
```

```
        if (money <= 0) {
            System.out.println("Invalid deposit");
        } else {
            this.balance += money;
            balance();
        }
    }
    public void withdraw(long money) {
        if (this.balance < money)
            System.out.println("Insufficient funds!");
        else {
            this.balance -= money;
            balance();
        }
    }
}
```

3) Define a class Employee with private attributes name,age,salary and public methods to get and set these attributes, and a package private method to display employee details. Create another class in the same package to test access to the displayEmployeeDetails method.

```
public class Employee {
    private String name;
    private int age;
    private double salary;

    Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public double getSalary() {
        return salary;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    void displayEmployeeDetails() {
        System.out.printf("Name: %S\nAge: %d\nSalary: %d", getName(), getAge(), getSalary());
    }
}
public class TestGetSet {
    public static void main(String[] args) {
        Employee e = new Employee("Hemanth", 20, 50000);
        e.displayEmployeeDetails();
        e.setAge(20);
```

```
        e.displayEmployeeDetails();

    }
}
```

8.6) What is inheritance?

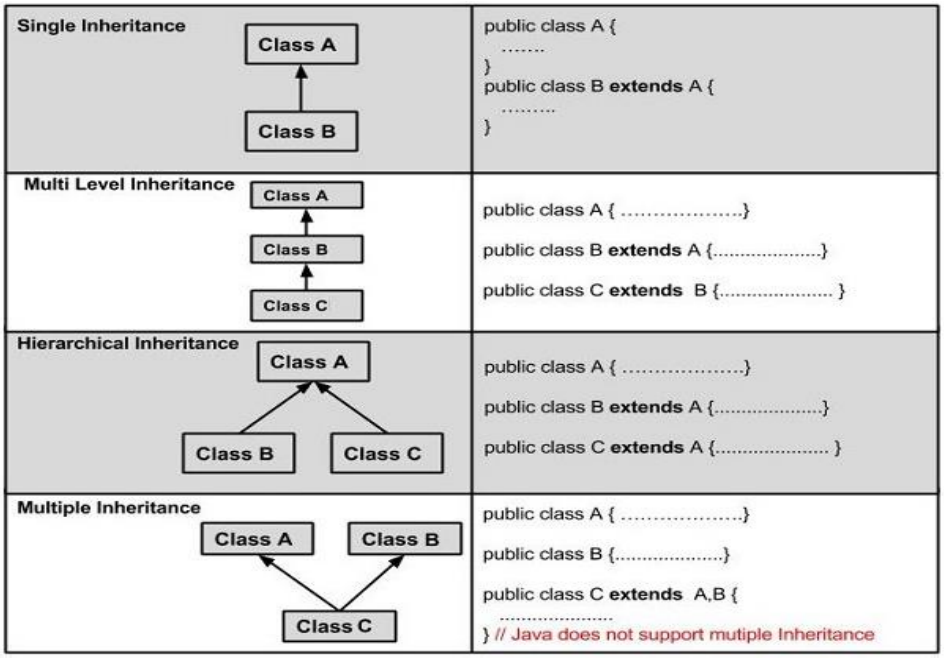
- Allows a new class(sub class/child class) to inherit features from an existing class(super class/parent class).
- Allows code reusability as sub classes can access the methods and variables of the super class.
- Protected access modifier was mostly used in inheritance to allow subclass access to super class members.

8.7) Types of inheritance

We can inherit the properties of one class at a time that's why multiple inheritance is not supported in java.The concept of interfaces was used to implement multiple inheritance.

4 types of inheritance are

1) Simple / single
2) Multiple (Not supported in java)
3) Multi-level
4) Hierarchical



8.8) Object class

In Java, the Object class is the superclass of all classes. Every class in Java implicitly extends Object, unless it explicitly extends another class. Since Java does not support multiple inheritance with classes, a class can extend only one other class at a time.

In any type of inheritance, such as single inheritance, if class B extends class A, then class A implicitly extends the Object class. This means that all classes in Java—either directly or through a chain of inheritance—ultimately inherit from the Object class.

Some of the methods of the object class are

- toString()
- equals()
- hashCode()
- getClass()

8.9) Equals and HashCode

hashCode() is not designed to determine full equality, but to narrow down comparisons. It acts as a fast pre-check: if two objects have different hash codes, they are definitely not equal. If they have the same hash code, then we use equals() to do a full comparison.

If a class has many fields (say, 100), and you want to compare two objects using equals(), then comparing each of those 100 fields can be expensive in terms of time and resources — especially in large data structures like HashSet or HashMap.

That's where hashCode() provides a performance optimization, two objects can have the same hash code if their content is the same even if they belong to different classes, especially if they override the hashCode() method.

Hashcode is of type int.

If the hashcodes of both the objects are same then equals() may return true/ false.

If 2 objects returns true on equals() method then their hashcodes must be equal.

```
public class Employe {
    private String name;
    private int age;
    private String id;
```

```
    public Employe(String name, int age, String id) {
        this.name = name;
        this.age = age;
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public String getId() {
        return id;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setId(String id) {
        this.id = id;
    }
    @Override
    public String toString() {
        return "EqualsCheck [name=" + name + ", age=" + age + ", id=" + id + "]";
    }
}
```

```
public class EqualsCheck {
    public static void main(String[] args) {
        Employe e1 = new Employe("Hemanth", 20, "004738");
        Employe e2 = new Employe("Hemanth", 20, "004738");
        System.out.println(e1 == e2); // false -> as == compares the references
        System.out.println(e1.equals(e2)); // false -> as equals method here also compares the
                        refernces, so we need to overeride equals method
    }

}
// toString method in String class doesn't compares the references as it overrides equals() by
   default.
```

After overriding the equals method,

instanceof is a keyword that was used to check if the object is the instance of particular class / subclass.

```
public class Employe {
    private String name;
    private int age;
    private String id;

    public Employe(String name, int age, String id) {
        this.name = name;
        this.age = age;
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
```

```
    }
    public String getId() {
        return id;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setId(String id) {
        this.id = id;
    }
    @Override
    public String toString() {
        return "EqualsCheck [name=" + name + ", age=" + age + ", id=" + id + "]";
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Employe) {
            Employe e = (Employe) obj;
            return e.name.equals(name) && e.age == age && e.id.equals(id);
        } else {
            return false;
        }
    }
}
```

```
public class EqualsCheck {
    public static void main(String[] args) {
        Employe e1 = new Employe("Hemanth", 20, "004738");
        Employe e2 = new Employe("Hemanth", 20, "004738");
        System.out.println(e1 == e2); // false -> as == compares the references
        System.out.println(e1.equals(e2)); // true -> compares the content
    }

}
```

Below is the Summary of equals-hashcode contract,

- If equals is true, hashCode must be equal. (REQUIRED)
- If hashCodes differ, equals must be false. (EXPECTED)
- If hashCodes are the same, equals can be either true or false. (DEPENDS)

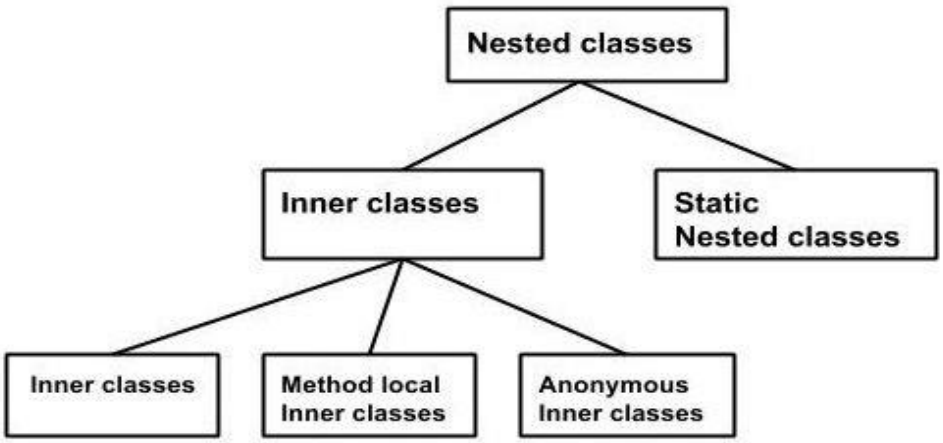
Below is the system generated methods of hashcodes and equal methods that are done through Right click > source action > generate hashCode() and equals()

```
//Employe.java

@Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        result = prime * result + age;
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        return result;
    }

    @Override
```

```java
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Employe other = (Employe) obj;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    if (age != other.age)
        return false;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    return true;
}
```

8.10) Nested and Inner classes

Nested classes are defined within another class .



- **Static Nested Classes**

  Act like static members of the outer class.

  Can only access static members of the outer class directly.

  Do not require an instance of the outer class to be created.

- **Inner (Non-static) Classes**

  Belong to an instance of the outer class.

  Can access all members of the outer class, including private members.

  Require an instance of the outer class to be instantiated.

- **Local Inner Classes**

  Defined inside a method or block.

  Scope is limited to that method/block.

  Cannot be accessed outside their defined scope.

- **Anonymous Inner Classes**

  Classes without a name, created for one-time use (usually to implement interfaces or abstract classes).

  Defined and instantiated in a single expression.

- **Top-Level Classes**

  Cannot be declared private or protected.

  Can only be public or package-private (default).

  Inner classes, however, can be private, protected, default, or public.

a non-static inner class cannot be accessed without creating an object of the outer class.

```
public class Outer {
    class Inner {
        void show() {
            System.out.println("Non-static inner class");
        }
    }

    public static void main(String[] args) {
        Outer outerobj = new Outer();
        Outer.Inner inner = outerobj.new Inner();
         //or Outer.Inner inner=new Outer().new Inner();
        inner.show();
    }
}
```

Static Nested Class Does not require an instance of the outer class.Only needs to be accessed using the outer class name.

```
public class Outer {
    static class Inner {
        void show() {
            System.out.println("Static nested class");
        }
    }

    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner(); // ✅
        inner.show();
    }
}
```

Local Inner Class (Defined in a method/block) are accessible only within the block where it is defined.

```
public class Outer {
    void method() {
        class Inner {
            void show() {
                System.out.println("Local inner class");
```

```
            }
        }

        Inner inner = new Inner(); // ✅Only accessible here
        inner.show();
    }
    public static void main(String[] args) {
        new Outer().method();
    }
}
```

Upcasting allows run-time polymorphism → ParentClass obj=new ChildClass();

CHALLENGES

1. Start with a base class LibraryItem that has itemID, title, author and methods checkOut(), returnItem(). create sub class like book, magazine, DVD, each inheriting from LibraryItem. Add unique attributes to each subclass like ISBN for book, issueNumber for magazine and duration for DVD.

```
package in.Package2;

public class LibraryItem {
    private int itemID;
    private String title;
    private String author;
    public LibraryItem(int itemID, String title, String author) {
        this.itemID = itemID;
        this.title = title;
        this.author = author;
    }
    public int getItemID() {
        return itemID;
    }
    public String getTitle() {
        return title;
    }
    public String getAuthor() {
        return author;
    }
```

```
    public void setItemID(int itemID) {
        this.itemID = itemID;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    @Override
    public String toString() {
        return "LibraryItem [itemID=" + itemID + ", title=" + title + ", author=" + author +
"]";
    }
    public void checkOut() {
        System.out.println(this.toString() + " Checked out!");
```

```
    }
    public void returnItem() {
        System.out.println(this.toString() + " returned successfully");
    }
}
```

```
package in.Package2;

public class Book extends LibraryItem {
    public int ISBN;
    public Book(int itemID, String title, String author, int iSBN) {
        super(itemID, title, author);
        ISBN = iSBN;
    }
    public int getISBN() {
        return ISBN;
    }
    @Override
    public String toString() {
        return super.toString() + "Book [ISBN=" + ISBN + "]";
    }
    public void setISBN(int iSBN) {
        ISBN = iSBN;
    }
    public static void main(String[] args) {
        Book b1 = new Book(4738, "Noting", "who knows", 143);
        System.out.println(b1);
    }
}
```

```
package in.Package2;

public class Magazine extends LibraryItem {
    private int issueNumber;
    public Magazine(int itemID, String title, String author, int issueNumber) {
        super(itemID, title, author);
        this.issueNumber = issueNumber;
    }
    public int getIssueNumber() {
        return issueNumber;
    }
    @Override
    public String toString() {
        return super.toString() + "Magazine [issueNumber=" + issueNumber + "]";
    }
    public void setIssueNumber(int issueNumber) {
        this.issueNumber = issueNumber;
    }
    public static void main(String[] args) {
        Magazine m1 = new Magazine(123, "vov", "vvit", 25);
        m1.checkOut();
    }
}
```

```
package in.Package2;

public class Dvd extends LibraryItem {
    private int duration;
    public Dvd(int itemID, String title, String author, int duration) {
        super(itemID, title, author);
        this.duration = duration;
```

```
    }
    public static void main(String[] args) {
        Dvd d1 = new Dvd(154, "my project", "me", 50);
        d1.returnItem();
    }
    public int getDuration() {
        return duration;
    }
    @Override
    public String toString() {
        return super.toString() + "Dvd [duration=" + duration + "]";
    }
    public void setDuration(int duration) {
        this.duration = duration;
    }
}
```

2. Create a person class with name and age as attributes and override equals to compare person objects based on their attributes and override hashcode() consistent with the definition of equals.

```
package in.Package2;

public class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        result = prime * result + age;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        if (age != other.age)
            return false;
        return true;
    }
}
```

3. Create a class ArrayOperations with a static nested class Statistics. Statistics should have methods like mean(), median() which operates on an array.

```java
package in.Package2;

import java.util.Arrays;

public class ArrayOperations {

    static class Statistics {

        public void mean(int[] arr) {
            double sum = 0;
            for (int i : arr)
                sum += i;
            System.out.println("Mean: " + (sum / arr.length));
        }

        public void median(int[] arr) {
            Arrays.sort(arr);
            double median;
            int mid = arr.length / 2;
            if (arr.length % 2 == 0) {
                median = (arr[mid - 1] + arr[mid]) / 2.0;
            } else {
                median = arr[mid];
            }
            System.out.println("Median: " + median);
        }
    }
    public static void main(String[] args) {
        ArrayOperations.Statistics obj = new ArrayOperations.Statistics();
        obj.mean(new int[] {3,1,4,2});
        obj.median(ne int[] {3,1,4,2});
    }
}
```

KEY POINTS

- Parent classes are known as general classes, whereas child classes are considered specific classes.
- The access modifier of the object class can be either public or default, but it cannot be private or protected as it is not an inner class.
- A static nested class cannot access instance variables of the outer class.
- Override both the toString() and hashCode() methods to maintain consistency.

9. Abstraction and Polymorphism

9.1) Abstraction

Abstraction hides complex implementation details and focuses only essential features.

We can't see the fan's circuit but it will work if we turn on the switch.

Focuses on functionality, more on what an object does but not how it does through clear interfaces.

Reduces the complexity by showing only relevant information in a class design.

9.2) Abstract Keyword

The abstract keyword in Java can be applied to both classes and methods. An abstract class cannot be instantiated, meaning we cannot create an object of it directly. However, it can be inherited by other classes.

For example, consider an abstract class named Vehicle. We can create multiple subclasses that extend the Vehicle class, such as Car, Bike, Cycle, etc. If someone asks you to create a Cycle, you can create an object of the Cycle class. But if someone asks you to create a Vehicle, the question becomes unclear—what kind of vehicle? Since Vehicle is abstract and represents a general concept, you cannot create an object of it. Instead, you instantiate one of its specific subclasses.

```
package Lecture9;

abstract class Vehicle {
    private int noOfTyres;
    public Vehicle(int noOfTyres) {
        this.noOfTyres = noOfTyres;
    }
    public void commute() {
        System.out.println("starting.......");
    }
    public int getNoOfTyres() {
        return noOfTyres;
    }
    public void setNoOfTyres(int noOfTyres) {
        this.noOfTyres = noOfTyres;
    }
}
```

```
package Lecture9;

public class Car extends Vehicle {
    private int noOfDoors;
    public Car(int noOfTyres, int noOfDoors) {
        super(noOfTyres);
        this.noOfDoors = noOfDoors;
    }
    public int getNoOfDoors() {
        return noOfDoors;
    }
    public void setNoOfDoors(int noOfDoors) {
```

```
        this.noOfDoors = noOfDoors;
    }
}
```

```
package Lecture9;

public class Test {
    public static void main(String[] args) {
        // Vehicle v = new Vehicle(4); can't be instantiated
        Car c = new Car(4, 5);
        // v.commute();
        c.commute();
    }
}
```

- Abstract method defines methods without implémentation.
- Abstract methods must be defined within abstract classes only.
- In Java, place the abstract keyword after the access modifier when declaring an abstract class or method.
- If your abstract class contains an abstract method, then any subclass must either be declared abstract itself or override the abstract method from the parent class.
- When a method must have different implementations in each subclass, we declare it as an abstract method in an abstract class. For instance, if the abstract class Vehicle contains an abstract method startSound(), then any concrete subclass like Car must override the startSound() method.

  This is because each vehicle may produce a different starting sound.

  If the subclass does not override the abstract method, it must itself be declared abstract.
- An abstract class in Java can have multiple abstract methods.
- In order to create a child class of an abstract class, the child class must either override all abstract methods from the parent class or declare itself as abstract.

```
package Lecture9;

public abstract class Vehicle {
    private int noOfTyres;
    public abstract void startSound();
    public Vehicle(int noOfTyres) {
        this.noOfTyres = noOfTyres;
    }
    public void commute() {
        System.out.println("starting.......");
    }
    public int getNoOfTyres() {
        return noOfTyres;
    }
    public void setNoOfTyres(int noOfTyres) {
        this.noOfTyres = noOfTyres;
    }
}
```

```java
package Lecture9;

public class Car extends Vehicle {
    private int noOfDoors;
    public Car(int noOfTyres, int noOfDoors) {
        super(noOfTyres);
        this.noOfDoors = noOfDoors;
    }
    @Override
    public void startSound() {
        System.out.println("vroooooohhhhhhhhh");
    }
    public int getNoOfDoors() {
        return noOfDoors;
    }
    public void setNoOfDoors(int noOfDoors) {
        this.noOfDoors = noOfDoors;
    }
}
```

9.3) Interfaces

- Interfaces in Java are not classes, but they are similar in structure. They are defined using the interface keyword. To implement an interface in a class, we use the implements keyword, which works similarly to extends in class inheritance, but is specifically used for interfaces.
- Abstract methods can have a return type and parameters, but they do not have a method body—this is because they are meant to be implemented by subclasses. The return type helps define what kind of value the method is expected to return once it is implemented.
- When an interface is compiled, it generates a .class file, just like a regular class.
- Naming conventions for interfaces are the same as those for classes (typically starting with an uppercase letter and using CamelCase).
- By default, interfaces in Java can only contain abstract methods (before Java 8). All methods in an interface are implicitly public and abstract, so there's no need to declare them explicitly with those modifiers.
- On the other hand, abstract methods in an abstract class can have any access modifier: public, protected, default, or even private (though private abstract doesn't make practical sense since it can't be overridden).
- You can think of an abstract class as a "responsibility" that subclasses must fulfill. If a subclass does not override all abstract methods of its abstract parent, it must also be declared abstract.
- If the first subclass does override all abstract methods, then its own subclasses are not required to override anything and don’t need to be abstract.

```java
package Lecture9;

public interface Transport {
    public abstract void getSetGo();
```

```
}
```

```
package Lecture9;

public abstract class Vehicle implements Transport {
    private int noOfTyres;
    public abstract void startSound();
    public Vehicle(int noOfTyres) {
        this.noOfTyres = noOfTyres;
    }
    public void commute() {
        System.out.println("starting.......");
    }
    public int getNoOfTyres() {
        return noOfTyres;
    }
    public void setNoOfTyres(int noOfTyres) {
        this.noOfTyres = noOfTyres;
    }
}
```

```
package Lecture9;

public class Car extends Vehicle {
    private int noOfDoors;
    public Car(int noOfTyres, int noOfDoors) {
        super(noOfTyres);
        this.noOfDoors = noOfDoors;
    }
    @Override
    public void startSound() {
        System.out.println("vroooooohhhhhhhhh");
    }
    @Override
    public void getSetGo() {
        System.out.println("3,2,1.....");
    }
    public int getNoOfDoors() {
        return noOfDoors;
    }
    public void setNoOfDoors(int noOfDoors) {
        this.noOfDoors = noOfDoors;
    }
}
```

Where to use extends and where to use implements?

Relationship	Keyword	Example
Class → Class	extends	class B extends A
Interface → Interface	extends	interface B extends A
Class → Interface	implements	class B implements A

A class in Java can implement multiple interfaces, and it can also extend a class while implementing multiple interfaces.

```
class Dog extends Animal implements Pet, Guard {
    // class body
}
```

Interfaces can have default methods and also static methods.

```
interface Pet {
    void play(); // abstract method

    default void sleep() {
        System.out.println("Sleeping..."); // default method
    }
    static void petInfo() {
        System.out.println("Pets are friendly animals."); // static method
    }
}
```

Default methods can be overridden(not compulsory if there is no conflict) in the implementing class.

```
// No conflict

interface A {
    default void show() {
        System.out.println("Default A");
    }
}

class MyClass implements A {
    // No override needed
}
public class Main {
    public static void main(String[] args) {
        new MyClass().show(); // Output: "Default A"
    }
}
```

```
// conflict - must be overridden

 interface A {
    default void show() {
        System.out.println("A");
    }
}

interface B {
    default void show() {
        System.out.println("B");
    }
}
class MyClass implements A, B {
    // ❌Compilation error unless we override show()
    // ✅Resolving the conflict:
    public void show() {
        System.out.println("MyClass resolves conflict");
    }
}
```

Static methods can not be overriden and are called using interface name.

InerfaceNmae.methodName();

```
Pet.petInfo();
```

Java does not support multiple inheritance with classes (i.e., a class cannot extend more than one class) to avoid ambiguity problems like the Diamond Problem.

However, Java *does* support multiple inheritance using interfaces.

A class can implement multiple interfaces, effectively inheriting behaviors from all of them.

```
interface Flyable {
    void fly();
}

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

Abstract classes can have concrete methods, static methods.

```
abstract class Animal {
    abstract void makeSound(); // abstract method

    void breathe() { // concrete method
        System.out.println("Breathing...");
    }
    static void info() { // static method
        System.out.println("Animals are living beings.");
    }
}
class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark!");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal.info(); // ✅Calling static method from abstract class
        new Dog().breathe(); // ✅Calling concrete method
    }
}
```

CHALLENGES

1) Create an abstract class shape with an abstract method calculateArea(). Implement 2 sub classes: circle and square, each subclass should have relevant attributes like radius for circle, side for square and their own implementation of the calculateArea() method.

```
package Lecture9;

public abstract class Shape {
    public abstract void calculateArea();
}
```

```
package Lecture9;
```

```java
public class Circle extends Shape {
    private final int r;
    public Circle(int r) {
        this.r = r;
    }
    @Override
    public void calculateArea() {
        System.out.println("Area of the circle: " + Math.PI * r * r);
    }
    public int getR() {
        return r;
    }
}
```

```java
package Lecture9;

public class Square extends Shape {
    private final int side;
    public Square(int side) {
        this.side = side;
    }
    public int getSide() {
        return side;
    }
    @Override
    public void calculateArea() {
        System.out.println("Area of square: " + side * side);
    }
```

```java
package Lecture9;

public class TestShape {
    public static void main(String[] args) {
        // Shape s=new Shape(); can't instantiate abstract classes
        Circle c = new Circle(10);
        c.calculateArea();
        Square s = new Square(10);
        s.calculateArea();
    }
}
```

- %5.2f means 5 characters wide with 2 digits after decimal point.
- Interfaces can be public and default but can't be private/ protected.

2) Create an interface Flyable with an abstract method fly(). create an abstract class Bird that implements Flyable. Implement a subclass eagle that extends bird. Provide an implementation for the fly() method.

```java
package Lecture9;

public interface Flyable {
    void fly();
}
```

```java
package Lecture9;
```

```java
public abstract class Bird implements Flyable {
    private final String breed;
    public Bird(String breed) {
        this.breed = breed;
    }
    public String getBreed() {
        return breed;
    }
}
```

```java
package Lecture9;

public class Eagle extends Bird {
    public Eagle() {
        super("Eagle");
    }
    @Override
    public void fly() {
        System.out.println("Eagle is flying");
    }
}
```

```java
package Lecture9;

public class FlyTest {
    public static void main(String[] args) {
        Eagle e = new Eagle();
        e.fly();
    }
}
```

9.4) What is polymorphism?

Polymorphism means "many forms". In Java, polymorphism allows the same method name to behave differently based on the object that calls it.

Compiler converts .java file to .class file(byte code).

Polymorphism types are

1) Compile-time polymorphism

- Which method need to be invoked is decided at the compile time.
- Can be achieved through method overloading or operator overloading.

2) Run-time polymorphism

- Which method need to be invoked is decided at the run time.
- Can be achieved through method overriding (when a subclass provides a specific implementation of a method already defined in its superclass).
- Also known as the dynamic method dispatch.

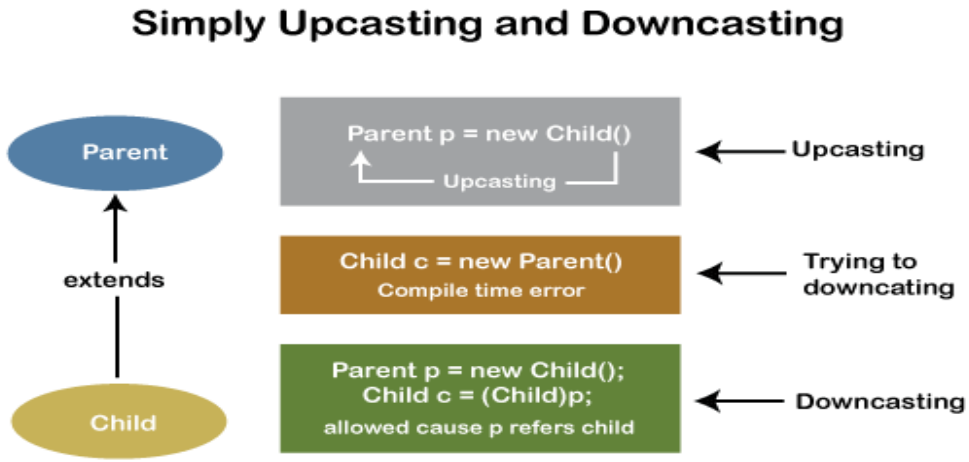
9.5) References and objects

Upcasting:

- Assigning a subclass object to a superclass reference.
- Automatic and safe

Downcasting:

- Casting a superclass reference back to a subclass type. To access sub-class specififc methods.
- Manual and risky.



When you override a method in a subclass and then upcast the object to its superclass, Java still calls the overridden method in the subclass, not the one in the superclass.

```
public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // Upcasting
        a.sound();             // Output is "Dog barks" not "Animal sounds"
    }
}
```

9.6) Method /constructor overloading

- Also known as compile time polymorphism.
- Occurs when multiple methods in the same class have the same name but with different parameters(either in the number of parameters or their types).
- Overloaded methods must differ in the number, type, or sequence of their parameters.
- In Java, the return type does not affect method overloading.

```
package Lecture9;

public class Overloading {
    public int add(int a, int b) {
        return a + b;
    }
    public String add(String a, String b) {
        return a + b;
    }
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String[] args) {
        Overloading obj = new Overloading();
        System.out.println(obj.add("Hemanth ", "Muvvala")); // Hemanth Muvvala
        System.out.println(obj.add(5, 10)); // 15
        System.out.println(obj.add(1, 2, 3)); // 6
    }
```

```
}
```

```
package Lecture9;

public class ConstrucorOverloading {
    ConstrucorOverloading() {
        System.out.println("It's a default constructor");
    }
    ConstrucorOverloading(String name) {
        System.out.printf("Hey %S! it's a parametarised constructor");
    }
    public static void main(String[] args) {
        ConstrucorOverloading obj1 = new ConstrucorOverloading();
         // It's a default constructor
        ConstrucorOverloading obj2 = new ConstrucorOverloading("Hemanth");
        // Hey Hemanth! It's a parameterized constructor
    }
}
```

9.7) Super Keyword

Super can be used to refer immediate parent class instance variables.

Can be used to invoke immediate parent class method.

super() can be used to invoke immediate parent class constructor.

Can not be called with the object reference because it is not reference to any particular object.

```
Dog dog = new Dog();
dog.super.sound();  // Compile-time error: 'super' cannot be used with an object reference.
```

In Java, when a method with the same name exists across multiple levels of an inheritance hierarchy—such as classes A, B, and C, where C extends B and B extends A—each class can override that method. If class C calls super.start(), it will invoke the start() method defined in class B, because super refers only to the immediate parent class. Even though class A also defines a start() method, it cannot be directly accessed from class C using something like super.super.start(), because Java does not support multi-level super calls like super.super. To access class A's start() method from class C, an indirect approach must be used: you can define a method in class B that explicitly calls super.start() (which refers to A's method), and then call that method from class C.

```
package Lecture9;

public class Animal {
    public void sound() {
        System.out.println("Animal is sounding");
    }
}
package Lecture9;

public class Dog extends Animal {
    public void sound() {
        System.out.println("Dog is barking");
    }
    public void sountOfAnimal() {
        super.sound();
    }
}
```

```java
package Lecture9;

public class SuperTest {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // Dog is barking
        d.sountOfAnimal(); // Animal is sounding
    }
}
```

9.8) Method/ Constructor overriding

- Method overriding occurs when subclass provides a specific implementation for a methods which was already defined in it's super class.
- Also known as run-time polymorphism.
- Overriden method can be called using super keyword.
- Signature must be same(name, return type and parameters) as the method of the parent class.
- @override annotation is optional, used for readability.
- When you override a method in a subclass, the access level cannot be more restrictive than the method in the superclass.
- If the parent class defines the start() method with default access, then the start() method in the child classes can be default or public, but not more restrictive (i.e., it cannot be private or protected).
- If the parent class defines the start() method as protected, then the start() method in the child classes can be protected, default, or public, but not private.

Superclass Method Modifier	Allowed Modifiers in Subclass	Not Allowed
public	public	protected, default, private
protected	protected, public	private
default (package-private)	default, protected, public (within the same package)	private
private	Cannot be overridden (not visible)	N/A

```java
package Lecture9;

public class Machine {
    public void start() {
        System.out.println("Machine is starting");
    }
}
```

```java
package Lecture9;

public class Tractor extends Machine {
    public void start() {
        System.out.println("Tractor is starting");
    }
```

```
}
```

```
package Lecture9;

public class Plane extends Machine {
    // public void start() {
    // System.out.println("Plane is on runway");
    // }
}
// Method not overridden, so inherits Machine's start()
```

```
package Lecture9;

public class TestMachine {
    public static void main(String[] args) {
        Machine m = new Machine();
        Tractor t = new Tractor();
        Plane p = new Plane();
        m.start(); // Machine is starting
        t.start(); // Tractor is starting
        p.start(); // Machine is starting

    }
}
```

9.9) Final keyword revisited

Value of a final variable can not be changed once initialized.

Final method can not be overridden by subclass.

Final class can not be inherited.

We can't declare an abstract method as final throws compile-time error.

9.10) Pass by Value vs Pass by reference

Variable pass by value - it's a default method, copies the argument values to function parameters. change in functions doesn't affect the original variable.

```
package Lecture9;

public class PassByValue {
    public static void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
        System.out.printf("%d , %d", a, b);
    }
    public static void main(String[] args) {
        int X = 10;
        int Y = 5;
        swap(X, Y); // 5 10
        System.out.printf("%d , %d", X, Y); // 10 5
    }
}
```

Java passes reference's value for objects, modifications to the objects in the method can affect the originals.

Primitive types always passes by value. Infunction changes doesn't affect the originals.

```java
package Lecture9;

public class PassByReference {
    public static class Pointer {
        public int x;
        public int y;
        public Pointer(int x, int y) {
            this.x = x;
            this.y = y;
        }
        @Override
        public String toString() {
            return "Pointer [x=" + x + ", y=" + y + "]";
        }
    }
    public static void move(Pointer p) {
        p.x++;
        p.y++;
        System.out.println("Inside move(): " + p);
    }
    public static void main(String[] args) {
        Pointer p = new Pointer(10, 11);
        System.out.println("Before move(): " + p);
        move(p);
        System.out.println("After move(): " + p);
    }
}
// Pointer [x=10, y=11]
// Pointer [x=11, y=12]
// Pointer [x=11, y=12]
```

In the above code, if inner class is not static then,

```java
PassByReference.Pointer p = new PassByReference().new Pointer(10, 11);
```

CHALLENGES

1) In a class calculator, create multiple add methods that overloads each other and can sum 2 integers, 3 integers, or 2 doubles. Demonstrate how each can be called with different number of parameters.

```java
package Lecture9;

public class Calculator {
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println(c.add(10, 20));// 30
        System.out.println(c.add(4.5, 3.5)); // 8.0
        System.out.println(c.add(10, 20, 30)); // 60
    }
}
```

If the method add(int a, int b) is commented out, then the call c.add(10, 20) will use the method add(double a, double b) because Java supports implicit type casting from int to double.

However, if the method add(double a, double b) is commented out, the call c.add(4.5, 3.5) will result in a compile-time error since Java does not allow implicit narrowing from double to int.

2) Define a base class Vehicle with method service() and a subclass Car that overrides the serivice().In car's service() provide a specific implementation that calls super.service() as well, to show how overriding works.

```
public class Vehicle {
    public void service() {
        System.out.println("Vehicle is servicing....");
    }
}
```

```
public class Car1 extends Vehicle {

    @Override
    public void service() {
        System.out.println("car is servicing.....");
    }
    public void callParentService() {
        super.service();
    }
    public static void main(String[] args) {
        Car1 c = new Car1();
        c.service(); // car is servicing.....
        c.callParentService(); // Vehicle is servicing....
    }
}
```

KEY POINTS

1) Abstraction hides the internal implementation and shows only the functionality to the users.

2) An abstract class in Java does not need to have any abstract methods. However, if a method is declared as abstract, then the class must also be declared abstract.

3) In Java, an interface cannot have instance fields or variables.

All variables declared in an interface are implicitly:

```
interface MyInterface {
    int VALUE = 100; // implicitly public static final
}
```

Is same as

```
interface MyInterface {
    public static final int VALUE = 100;
}
```

And can not be private.

4) Overloaded methods need not to have same return type.

5) A public subclass can override a protected superclass as we can increase the access down the line but can't restrict the access.

6) Java supports pass by value for reference types.

7) When an object is passed to a method then java copies reference to the object.

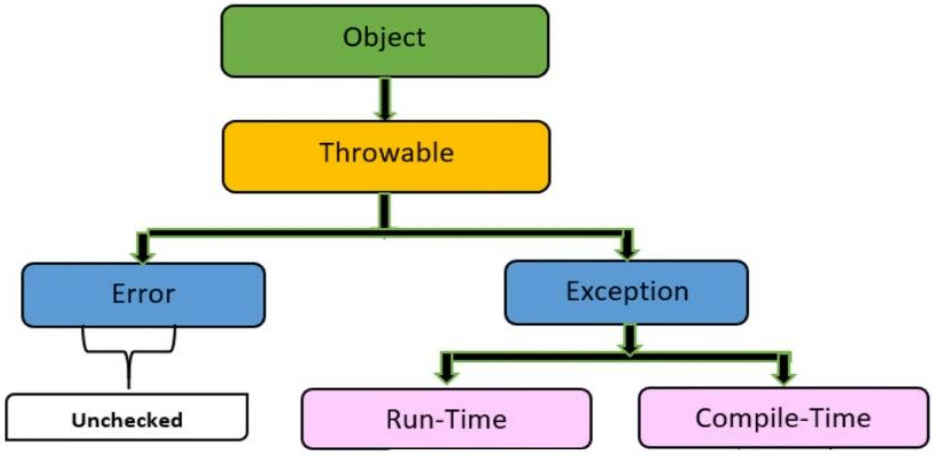
10. Exception and File Handling

10.1) What is an Exception

Exception is a disruptive event that interrupts the normal flow of execution of a program.

It's an instance of a problem that arises when program is running.

Exceptions are objects that encapsualtes the info about he error and that info includes type and state of the progarm when the error occurred.



- Errors are not under the control of the programmer.
- Errors generally represent serious problems that are outside the scope of application handling (e.g., OutOfMemoryError) and are considered unrecoverable.
- Exceptions are of two types:
  1. Runtime exceptions (unchecked)
  2. Compile-time exceptions (checked)
- Exceptions occur due to logical errors made by the programmer.
- ArrayIndexOutOfBoundsException and ArithmeticException are examples of runtime exceptions.
- Compile-time exceptions (also known as checked exceptions) are exceptions that the compiler knows may occur, and therefore must be handled in advance using try-catch blocks or by declaring them with throws.
- As shown in the diagram , the Object class is the parent of all classes that do not explicitly extend another class.
- Throwable is a class (not abstract) that extends Object and acts as the superclass for all errors and exceptions.
- Error and Exception are two direct subclasses of Throwable.

10.2) Try-Catch

Exceptions are classes.

Try block is the block of code that causes the exception.

Catch block is the block of code that catches and handles the exception thrown by the try block.

If your code has multiple catch blocks for different exceptions, only the first exception that is thrown in the try block and matches a catch block will be caught. Once an exception is caught by a catch block, no further catch blocks will be checked for that exception.

Below is how you handle a single exception.

```
package Lecture10;

import java.util.Scanner;
public class Calculator {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int a, b;
        System.out.print("Enter numerator: "); // Enter numerator: 5
        a = s.nextInt();
        System.out.print("Enter denominator:"); // Enter denominator: 0
        b = s.nextInt();
        try {
            System.out.println("Division: " + (a / b));
        } catch (ArithmeticException hurrrayyy) {
            System.out.println(hurrrayyy.getMessage()); // by zero
            System.out.println("can't divide with zero"); // can't divide with zero
        }
    }
}
```

when writing multiple catch blocks in Java, the general rule is that the more specific exceptions should be caught first, followed by the more general ones.

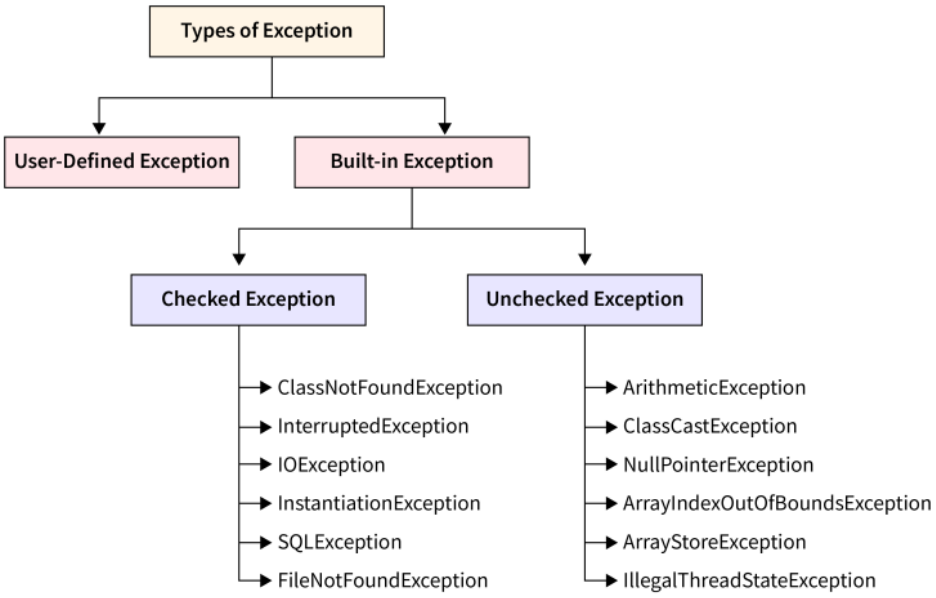
- Most specific exceptions should come first (like ArithmeticException).
- More general exceptions like Exception should come after specific ones.
- The most general Throwable class should be the last one, as it can catch any exception or error that wasn't caught earlier.

We can write multiple try-catch blocks.

```
package Lecture10;

import java.util.Scanner;
public class Calculator {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int a, b;
        System.out.print("Enter numerator: "); // Enter numerator: 5
        a = s.nextInt();
        System.out.print("Enter denominator: "); // Enter denominator: 0
        b = s.nextInt();
        try {
            // This will cause ArrayIndexOutOfBoundsException
            int[] arr = new int[5];
            arr[5] = 10; // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {
            // Catch both exceptions in a single block
            System.out.println("Exception caught: " + e.getMessage());
            if (e instanceof ArithmeticException) {
                System.out.println("Can't divide by zero.");
            } else if (e instanceof ArrayIndexOutOfBoundsException) {
                System.out.println("Limit of array exceeded.");
```

```
            }
        } catch (Exception ex) {
            System.out.println("General exception");
        } catch (Throwable t) {
            System.out.println("Throwable Exception.");
        }
        try {
            // Now perform the division operation, which could throw ArithmeticException
            System.out.println("Division: " + (a / b)); // This can throw ArithmeticException
        } catch (ArithmeticException hurrrayyy) {
            System.out.println(hurrrayyy.getMessage());
            System.out.println("Can't divide with zero");
        } catch (Exception ex) {
            System.out.println("General exception");
        } catch (Throwable t) {
            System.out.println("Throwable Exception.");
        }
    }
}
```

10.3) Types of Exceptions



Checked exceptions are those exceptions that must be either caught or declared in the method using throws keyword.

Unchecked exceptions are need not to be handled explicitly.

All the exceptions we handles above are the unchecked exceptions.

Checked exceptions must be handled and it's your wish to handle unchecked exceptions.

User-defined exceptions in Java are custom exception classes that you create yourself to handle specific, application-related error.

10.4) Throw and throws keyword

Throws keyword

Used in method declarations to indicate that a method might throw one or more checked exceptions.

When a method declares that it throws a checked exception using throws, any method that calls it must either:

- ✓ Handle the exception using try-catch, or
- ✓ Declare it again using throws

This is similar to abstract classes: if a class inherits an abstract method, it must override it, unless it's also abstract.

```
public void methodName() throws IOException {
    // code that might throw IOException
}
```

Only used for checked exceptions (e.g., IOException, SQLException), not for unchecked ones like ArithmeticException.

```
class Calculator {

    // Method that declares an exception using throws
    public void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        System.out.println("Division: " + (a / b));
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        try {
            calc.divide(10, 0);  // Must handle the exception because divide() throws
ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Throw keyword

Used to explicitly throw an exception(either new or an existing exception object) form a method or a block of code.

You can throw any Throwable (checked or unchecked).

Ex: throw new ArithmeticException(“Division by zero”);

exit code = 0: program ran successfully.

exit code ≠ 0: abnormal termination (usually due to an unhandled exception).

```
package Lecture10;

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

```
        b();
    }
    public static void b() {
        c();
    }
    public static void c() {
        d();
    }
    public static void d() {
        Scanner s = new Scanner(System.in);
        int a, b;
        System.out.print("Enter numerator: ");
        a = s.nextInt();
        System.out.print("Enter denominator:");
        b = s.nextInt();
        try {
            int[] arr = new int[5];
            arr[5] = 10;
            System.out.println("Division: " + (a / b));
        } catch (ArithmeticException hurrrayyy) {
            System.out.println(hurrrayyy.getMessage());
            System.out.println("can't divide with zero");
        } catch (Throwable t) {
            System.out.println("Limit of array exceeded.");
            throw t;
        }
    }
}
```

Below is the calling stack of the above progarm

```
>>
Enter numerator: 5
Enter denominator:0
Limit of array exceeded.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for
length 5
        at Lecture10.Calculator.d(Calculator.java:33)
        at Lecture10.Calculator.c(Calculator.java:20)
        at Lecture10.Calculator.b(Calculator.java:16)
        at Lecture10.Calculator.a(Calculator.java:12)
        at Lecture10.Calculator.main(Calculator.java:7)
```

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exceptions cannot be propagated using throw.	Checked exceptions can be propagated with throws.
Syntax: throw is followed by an instance of an exception class.	Syntax: throws is followed by exception class names.
Example: throw new NumberFormatException("Invalid input");	Example: throws IOException, SQLException
throw is used inside a method body.	throws is used in the method declaration (signature).
You can throw only one exception at a time using throw.	You can declare multiple exceptions using throws.

Example: throw new IOException("Connection failed!!");	Example: public void method() throws IOException, SQLException

```
public class ThrowDemo {
    public static void readFile() throws IOException {
        throw new IOException("File not found");
    }

    public static void main(String[] args) throws IOException {
        readFile(); // exception passed to JVM
    }
}
```

Here the exception is not handled by main() methods, it also escaped by defining throws in its signature.

Here is how you create and handle a user-defined exception.

```
// This is a checked exception because it extends Exception
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

```
public class VoterEligibility {

    public static void checkEligibility(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above to vote.");
        }
        System.out.println("You are eligible to vote!");
    }
}
```

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your age: ");
        int age = sc.nextInt();
        try {
            VoterEligibility.checkEligibility(age);
        } catch (InvalidAgeException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

10.5) Finally block

Nested try-catch block is also possible in Java.

The finally block executes compulsorily, regardless of whether an exception occurs or not.

It is ideal for closing resources like files or database connections to prevent resource leaks.

```
try {
    try {
```

```
        // Inner try block logic
    } catch (Exception e1) {
        // Inner catch block
    } catch (AnotherException e2) {
        // Another inner catch block
    } finally {
        // Inner finally block (always executes)
    }
} catch (OuterException e3) {
    // Outer catch block
} catch (Exception e4) {
    // Another outer catch block
} finally {
    // Outer finally block (always executes)
}
```

10.6) Custom exceptions

These are the user-defined exception classes that extend either Exception(for checked exceptions) or RuntimeException(for unchecked exceptions).

```
package Lecture10;

public class CustomException extends RuntimeException {
    private boolean ticketPurchased;
    public CustomException(boolean ticketPurchased) {
        this.ticketPurchased = ticketPurchased;
    }
    public boolean isTicketPurchased() {
        return ticketPurchased;
    }
    @Override
    public String getMessage() {
        return "you can't enter the theater";
    }
}
```

```
package Lecture10;

import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Did you purchased the ticket: ");
        boolean bool = s.nextBoolean();
        try {
            if (!bool) {
                throw new CustomException(bool);
            } else {
                System.out.println("You can gooo!");
            }
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println("General exception: " + e.getMessage());
        } finally {
            System.out.println("Thaks for using this service");
        }
    }
}
```

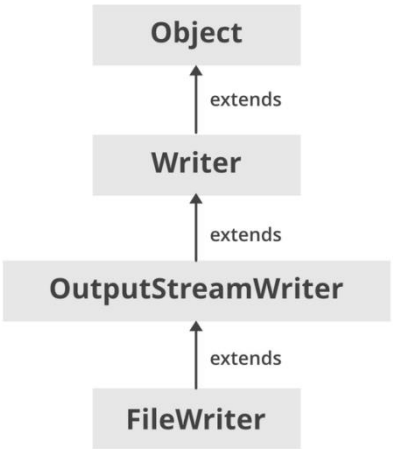
If we enter a number instead of true/false then it will be caught by the generic catch (Exception e) block in our code.

CHALLENGE

Write a program that asks the user to enter 2 integers and then divides first by second. The program should handle any arithmetic exceptions that may occur (like division by 0) and display an appropriate message.

```
package Lecture10;

import java.util.Scanner;
public class DivisonBy0 {
    public static void main(String[] args) {
        try {
            Scanner s = new Scanner(System.in);
            int a, b;
            System.out.print("Enter 1st num: "); // true
            a = s.nextInt();
            System.out.print("Enter 2nd num: ");
            b = s.nextInt();
            System.out.println("Division: " + (a / b));
        } catch (ArithmeticException e) {
            System.out.println("Invalid input \nArithmetic exception occured");
        } catch (Exception e) {
            System.out.println("General exception: " + e.getMessage()); // General exception:
For input string: "true"
        } finally {
            System.out.println("Thank you for using this program!"); // Thank you for using
this program!
        }
    }
}
```

10.7) FileWriter class



FileWriter is used to write a stream of characters into files.

Prefer writing text over binary data.

Constructor can be created in 2 ways

- ✓ FileWriter(String filename) → creates a FileWriter obj given the name of the file to write to.

- ✓ FileWriter(File f) ⟶ creates a FileWriter obj given a file object.

flush() ⟶ forcefully clear any buffered data and send it to the intended destination I.e., file.

close() ⟶ Releases the resources.

close() automatically calls flush(), but calling flush() manually is useful when you want to write intermediate data without closing the stream.

```java
import java.io.FileWriter;

public class FileWriting {
    public static void main(String[] args) {
        String fname = "sample.txt";
        try (FileWriter fw = new FileWriter(fname)) {
            fw.write("Hello world!");
            fw.flush();
            System.out.println("File written succesfully");
        } catch (Exception e) {
            System.out.println("General Exception: " + e.getMessage());
        }
    }
}
```

If we use the above try with resource method then we don't need to explicitly close the file writer.

OR

```java
import java.io.FileWriter;

public class FileWriting {
    public static void main(String[] args) {
        String fname = "sample.txt";
        try {
            FileWriter fw = new FileWriter(fname);
            fw.write("Hello world!");
            fw.flush();
            fw.write("My name is Hemanth");
            fw.flush();
            fw.close();
            System.out.println("File written succesfully");
        } catch (Exception e) {
            System.out.println("General Exception: " + e.getMessage());
        }
    }
}
```

10.8) FileReader

Used to read the stream of characters from files.

It's a character based stream meaning it reads characters.

Procedure of the Constructor creation is as same as of FileWriter.

Common methods are,

read() ⟶ reads single character and returns it as an integer, returns -1 if the end of the stream is reached.

read(char[] cbuff) ⟶ reads the charcters into an array also known as cbuf(character buffer) and returns the no.of characters read.

once you've read to the end with FileReader, the only reliable way to re-read is to reopen the file.

```java
import java.io.FileReader;
import java.io.IOException;

public class FileReading {
    public static void main(String[] args) {
        String fname = "/C://Users//91868//OneDrive//Desktop//JAVA//CODE//sample.txt/";
        int val = 0;
        try (FileReader fr = new FileReader(fname)) {
            while ((val = fr.read()) != -1) {
                System.out.print((char) val); // Hello world!My name is Hemanth
            }
        } catch (IOException e) {
            System.out.println("IOException occred: " + e.getMessage());
        }
    }
}
```

CHALLENGE

File not found exception handling.

Write a program to read a filename from the user and display its content. The progarm should handle the situation where the file doesnot exists.

```java
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class Challenge {
    public static void main(String[] args) throws IOException {
        Scanner s = new Scanner(System.in);
        String fname;
        System.out.print("Enter the file name:");
        fname = s.next();
        int val = 0;
        try (FileReader fr = new FileReader(fname)) {
            while ((val = fr.read()) != -1) {
                System.out.print((char) val);
            }
        } catch (FileNotFoundException f) {
            System.out.println("file can not be found: " + f.getMessage());
        }
    }
}
```

OR

```java
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class Challenge {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String fname;
        System.out.print("Enter the file name:");
        fname = s.next();
        int val = 0;
        try (FileReader fr = new FileReader(fname)) {
```

```
            while ((val = fr.read()) != -1) {
                System.out.print((char) val);
            }
        } catch (FileNotFoundException f) {
            System.out.println("file can not be found: " + f.getMessage());
        } catch (IOException e) {
            System.out.println("IOException occured: " + e.getMessage());
        }
    }
}
```

Below throws an error because FileNotFoundException itself is a child class of IOException.

```
catch (FileNotFoundException | IOException f) {
            if (f instanceof FileNotFoundException) {
                System.out.println("File not found : " + f.getMessage());
            } else {
                System.out.println("IOException: " + f.getMessage());
            }
        }
```

KEY POINTS

- A method that throws a checked exception must declare the exception using the throws keyword in its signature.
- We can handle more than 1 exception in a single catch block.
- Finally block will always be executed even if try-catch have return statements.
- Catch block will not be executed if an error isn't occurred in the try block.
- Unchecked exceptions are direct subclass of Exception not Throwable but Exception is direct subclass of Throwable.
- A catch block can not exist independently without try block.
- throw keyword is used to propagate an exception up the call stack.

11. Collections & Generics

11.1) Variable arguments

- Java's varargs feature allows methods to accept any number of arguments.
- It is declared using an ellipsis (...).
- Internally, varargs are treated like arrays.
- A method using varargs can still be called even if no arguments are passed.
- If you want to require at least two arguments, you can declare two fixed parameters followed by varargs:

```
void sum(int a, int b, int... nums){


}
```

- Here, a and b are mandatory, and nums can accept any additional arguments.
- **Varargs must always be the last parameter in the method signature.**
- **Its also valid to write String... args as the main method signature.**

```
package Lecture11;

public class VarArgs {
    public static void main(String... args) {
        System.out.println("Hello world! " + args[0]);
        VarArgs obj = new VarArgs();
        obj.sum(1, 2, 3, 4, 5, 6);
        obj.sum(5, 7);
    }
    public void sum(int... nums) {
        int sum = 0;
        for (int var : nums) {
            sum += var;
        }
        System.out.println("Sum: " + sum);
    }
}
// Hello world!
// Sum: 21
// Sum: 12
```

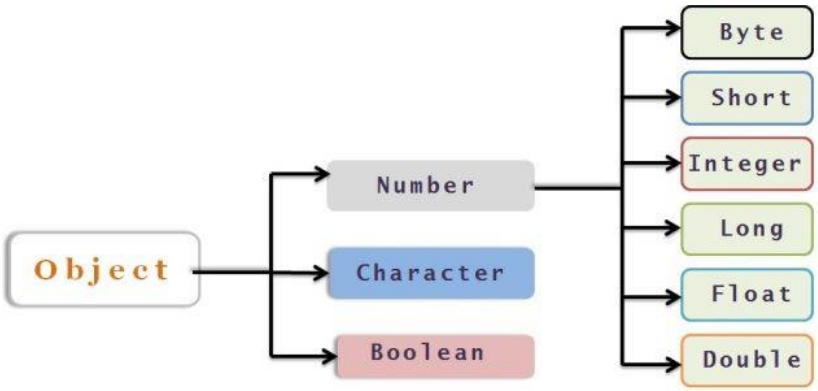
```
PS C:\Users\91868\OneDrive\Desktop\JAVA\CODE> javac Lecture11\VarArgs.java
PS C:\Users\91868\OneDrive\Desktop\JAVA\CODE> java Lecture11.VarArgs Hemanth
Hello world! Hemanth
Sum: 21
Sum: 12
PS C:\Users\91868\OneDrive\Desktop\JAVA\CODE>
```

11.2) Wrapper classes

- Using wrapper classes we can use primitive datatypes as objects.
- Automatic conversion between the primitive types and their corresponding wrapper classes.
- Once created, value of a wrapper object can not be changed.
- Provides useful methods like valueOf(), compareTo(),…
- Mainly used to store primitives in collection objects like ArrayList, HashMap,….
- We can assign null to primitive values if needed.

```
        int a = null; // invalid
        Integer b = null; // valid
```

- AutoBoxing is the automatic conversion of primitive types to their corresponding wrapper class objects.
- Unboxing is the automatic conversion of wrapper class objects back to their respective primitive types.

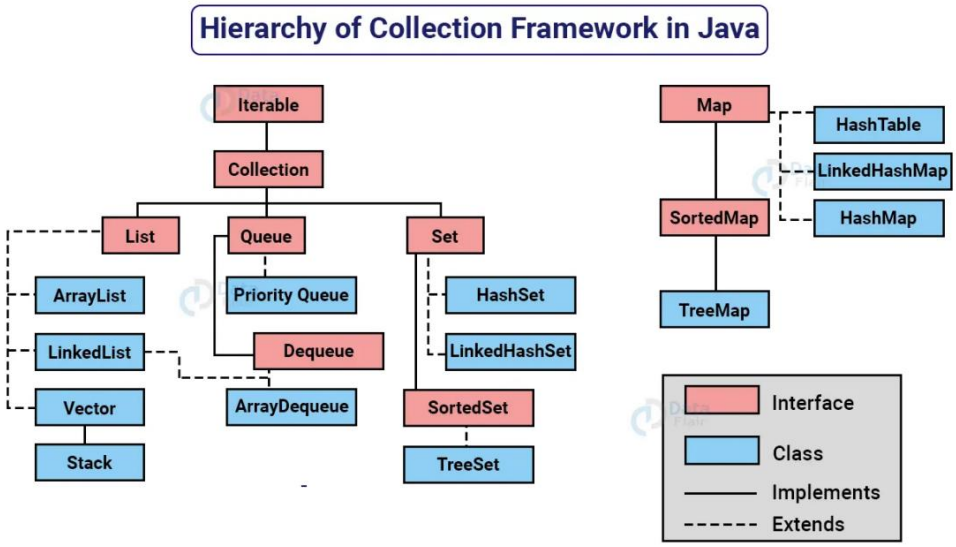



```
package Lecture11;

public class WrapperClass {
    public static void main(String[] args) {
        Integer num = Integer.valueOf("55"); // Converts a String to an Integer object
        Integer num2 = 54; // Autoboxing: primitive int to Integer
        Integer num3 = Integer.valueOf(12); // Explicit boxing
        int primitiveNum = num2; // unboxing: Integer to int
        int sum = primitiveNum + num3;
        // Output
        System.out.println("num  = " + num);
        System.out.println("num2 = " + num2);
        System.out.println("num3 = " + num3);
        System.out.println("primitiveNum (unboxed from num2) = " + primitiveNum);
        System.out.println("Sum of primitiveNum and num3 = " + sum);
    }
}
// num = 55
// num2 = 54
// num3 = 12
// primitiveNum (unboxed from num2) = 54
// Sum of primitiveNum and num3 = 66
```

11.3) Collections library

- Collections are groups of similar elements.
- They support dynamic memory allocation, unlike arrays which are fixed in size.
- The Collection and Map interfaces are part of the Java Collections Framework.
- The Collection interface is the root interface of the collection hierarchy and provides basic operations like: add(), remove(), clear(), and size().
- Any class that overrides these methods can be considered a collection type.

- We can create custom collections by implementing the Collection interface (or a more specific interface such as List, Set, or Queue) in our class and overriding methods such as add(), remove(), clear(), and size().
- A LinkedList in Java is an example of multiple inheritance via interfaces, as it implements both the List and Queue interfaces.



- List interface is an ordered collection that can contain duplicate elements.
- Set interface is a collection that cannot contain duplicate elements.
- Queue interface is a collection used to hold elements in FIFO.
- Map interface is not truly a collection, but part of the collections framework and stores the key-value pairs where keys are unique but not the values.

11.4) List Interface

- List is an ordered collection that allows duplicates.
- Elements can be accessed by their integer index and also maintains the insertion order of elements.
- Grows automatically as elements are added, offers fast random access and quick iteration.
- Preferred over arrays as size is dynamic.

```java
import java.util.ArrayList;

public class ListDemo {
    public static void main(String[] args) {
        // Create a new ArrayList to store Integer values
        ArrayList<Integer> list = new ArrayList<>();
        // Adding elements to the list
        list.add(10); // list: [10]
        list.add(30); // list: [10, 30]
        list.add(40); // list: [10, 30, 40]
        list.add(1, 20); // Inserts 20 at index 1 -> list: [10, 20, 30, 40]
```

```
        printList(list); // Output: 10, 20, 30, 40

        // Removing elements
        list.remove(0); // Removes element at index 0 (10) -> list: [20, 30, 40]
        list.remove(Integer.valueOf(40)); // Removes the value 40 -> list: [20, 30]

        printList(list); // Output: 20, 30

        // Accessing and modifying elements
        System.out.println(list.get(0)); // Gets the element at index 0 -> Output: 20
        list.set(0, 30); // Replaces element at index 0 with 30 -> list: [30, 30]
        // Note: set() cannot add a new element; it only modifies an existing one

        printList(list); // Output: 30, 30

        // Checking for presence and position of an element
        System.out.println(list.contains(25)); // false (25 not in list)
        System.out.println(list.indexOf(25)); // -1 (25 not found)

        list.clear(); // Removes all elements -> list: []

        printList(list); // Output: empty
    }
    public static void printList(ArrayList<Integer> list) {
        // for (int i : list) {
        // System.out.println(i);
        // }
        // OR
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        System.out.println("----------------------");
    }
}
```

```
// 10
// 20
// 30
// 40
// ----------------------
// 20
// 30
// ----------------------
// 20
// 30
// 30
// ----------------------
// false
// -1
// ----------------------
```

- <> are used in Java generics to help developers catch type-related errors at compile time. These are known as diamond operators or diamond brackets.
- After compilation, the compiler performs a process called type erasure, which removes the generic type information. This means the type parameters (inside <>) do not exist at runtime.

Generics are used at compile time only to enforce type safety. Example:

```
List li = new ArrayList();
```

The above line is valid in Java, but it creates a raw type. This means the list can store any type of object (e.g., Integer, String, etc.), but it disables compile-time type checking and may lead to runtime ClassCastException. It's recommended to use generics like this:

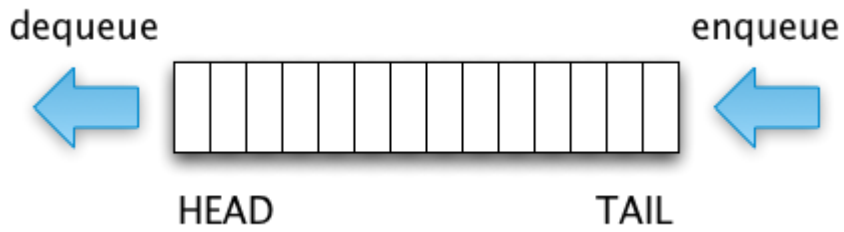
```
List<String> li = new ArrayList<>();
```

This ensures that only String objects can be added to the list, providing better type safety and avoiding casting issues.

Method	Description
add(E e)	→ Adds an element to the list at the end.
add(int index, E element)	→ inserts an element at the specified index, shifting elements to the right.
remove(int index)	→ Removes the element at the specified index.
remove(Object o)	→ Removes the first occurrence of the specified element.
get(int index)	→ Retrieves the element at the specified index.
set(int index, E element)	→ Replaces the element at the specified index with the given element.
contains(Object o)	→ Checks if the list contains the specified element.
indexOf(Object o)	→ Returns the index of the first occurrence of the specified element.
clear()	→ Removes all elements from the list.
size()	→ Returns the number of elements currently in the list.

11.5) Queue Interface

- Folllows FIFO.
- There are 2 ends - one for insertion and the other for removal.



Method	Description
add(E e) offer(E e)	→ inserts specific element, throws exception if it can not be added. → Same as above but returns false if the element can not be added.
remove() poll()	→ retreives and removes the head of the queue, throws exception if queue is empty. → same as above but returns null if queue is empty.
element() peek()	→ Retrieves but doesnot remove the head of the queue, throws exception if queue is empty. → Same as above but returns null if queue is empty.

Since we need to print collections multiple times, we define a custom utility class named Hemanth with a static method print(). This method accepts a Collection object as an argument, which allows it to handle any type of collection (like List, Queue, Set, etc.) because Collection is a common parent interface.

```java
import java.util.Collection;

public class Hemanth {
    public static void print(Collection obj) { // since Collection is a parent class, it can
accept any of its child
        System.out.print("Collection is: "); // class as parameter.
        for (Object a : obj) {
            System.out.print(a + " ");
        }
        System.out.println();
    }
}
```

```java
import java.util.LinkedList;
import java.util.Queue;

public class TestingQueue {
    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<>();
        q.add(10);
        q.offer(20);
        Hemanth.print(q);
        System.out.println(q.peek());
        System.out.println(q.element());
        Hemanth.print(q);
        System.out.println(q.poll());
        Hemanth.print(q);
        System.out.println(q.remove());
        Hemanth.print(q);
        System.out.println(q.poll());
    }
}
```

```
// Collection is: 10 20
// 10
// 10
// Collection is: 10 20
// 10
// Collection is: 20
// 20
// Collection is:
// null
```

11.6) Set Interface

- Contains unique elements.
- Doesnot guarentees any specific order of elements.
- No positional access I.e doesnot support indexing.
- Common implementations are HashSet, LinkedHashSet, and TreeSet(maintains order and uniqueness).
- The contains() method in a Set is usually faster than in a List.

Method	Description
add(E e)	→ Returns true if the element was added, false if it already exists
remove(Object o)	→ Returns true if the element was present and removed, else false.
contains(Object o)	→ Returns true if the element exists in the set.

size()	→ returns no.of elements in the set.
isEmpty()	→ Returns true if the set has no elements, else false.

```java
import java.util.HashSet;
import java.util.Set;

public class TestingSet {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        System.out.println(set.add("Hemanth"));
        System.out.println(set.add("Line2"));
        System.out.println(set.add("Apple"));
        System.out.println(set.add("Apple"));
        Hemanth.print(set);
        System.out.println(set.size());
    }
}
```

```java
// true
// true
// true
// false
// Collection is: Apple Hemanth Line2
// 3
```

11.7) Collections class

Collections is an utility class where Collection is an interface.

Offers different predefined static methods to perform operations like sort, max, min ,reverse, binary searching, shuffling ….

We can also make an unmodifiable list.

```java
List<Integer> finalList = Collections.unmodifiableList(list);
        finalList.add(10);
        Hemanth.print(finalList);
        // java.lang.UnsupportedOperationException
```

Use of some of the methods are

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

public class CollectionsTesting {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(5);
        list.add(-7);
        Hemanth.print(list);
        Collections.sort(list);
        Hemanth.print(list);
        System.out.println(Collections.max(list));
        System.out.println(Collections.min(list));
        Collections.reverse(list);
        Hemanth.print(list);
        System.out.println(Collections.binarySearch(list, 5));
        List<Integer> permanentList = Collections.unmodifiableList(list);// read-only list
    }
```

```
}
```

```
// Collection is: 10 5 -7
// Collection is: -7 5 10
// 10
// -7
// Collection is: 10 5 -7
// 1
```

CHALLENGES

1) Write a method concatinate strings that takes variable arguments of string type and concatinates them into a single string.

```
public class Challenge1 {
    public static void concatinateString(String... lines) {
        StringBuilder sb = new StringBuilder();
        for (String line : lines)
            sb.append(line).append(" ");
        System.out.println("Final String: " + sb.toString());
    }

    public static void main(String[] args) {
        concatinateString("Hello", "My", "Name", "is", "Hemanth");
    }
}
```

```

// Final String: Hello My Name is Hemanth
```

2) Write a program that sorts a list of String objects in descending order using a custom comparator.

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Arrays;;

public class ComparatorDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("C", "B", "A", "Z");
        Hemanth.print(list);
        sortDescending(list);
    }
    public static void sortDescending(List<String> list) {
        Collections.sort(list, new Comparator<String>() {
            @Override
            public int compare(String str1, String str2) {
                if (str1.equals(str2))
                    return 0;
                else if (str1.charAt(0) < str2.charAt(0))
                    return 1;
                else
                    return -1;
            }
        });
        Hemanth.print(list);
    }
}
```

```
Collection is: C B A Z
Collection is: Z C B A
```

OR

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

public class ListOfStrings {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("C");
        list.add("A");
        list.add("B");
        list.add("D");
        Hemanth.print(list);
        Collections.sort(list);
        Hemanth.print(list);
        Collections.reverse(list);
        Hemanth.print(list);
    }
}
```

```
// Collection is: C A B D
// Collection is: A B C D
// Collection is: D C B A
```

3) Use the collections class to count the frequency of a particular element in an arraylist.

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

public class FrequencyCount {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(2);
        Hemanth.print(list);
        System.out.println("Frequency of 2: " + Collections.frequency(list, 2));
    }
}
```

```
// Collection is: 1 2 3 2
// Frequency of 2: 2
```

4) Write a method to swap 2 elements in an ArrayList, given their indices.

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

public class Swap {
    public static void swap(List<String> list, int indx1, int indx2) {
        String temp1 = list.get(indx1);
        String temp2 = list.get(indx2);
        list.set(indx1, temp2);
        list.set(indx2, temp1);
        System.out.print("After ");
        Hemanth.print(list);
    }
```

```
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("one");
        list.add("Two");
        list.add("Three");
        System.out.print("Before ");
        Hemanth.print(list);
        swap(list, 0, 2);
    }
}
```

```
// Before Collection is: one Two Three
// After Collection is: Three Two one
```

5) Create a program that reverses the elements of a list and prints the reversed list.

```
import java.util.List;
import java.util.Arrays;

public class Reverse {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
        for (int i = 0, j = list.size() - 1; i < j; i++, j--) {
            int temp1 = list.get(i);
            int temp2 = list.get(j);
            list.set(j, temp1);
            list.set(i, temp2);
        }
        System.out.println(list); // [6, 5, 4, 3, 2, 1]
    }
}
```

OR

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

public class ReversedList {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3);
        System.out.print("Before ");
        System.out.println(list);
        Collections.reverse(list);
        System.out.print("After ");
        System.out.println(list);
    }
}
```

```
// Before [1, 2, 3]
// After [3, 2, 1]
```

6) create a PriorityQueue of a custom class Student with attributes name and grade. Use a comparator to order by grade.

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class Student1 {
```

```
    public static class InnerStudent1 {
        private final String name;
        private final char grade;
        public InnerStudent1(String name, char grade) {
            this.name = name;
            this.grade = grade;
        }
        public String getName() {
            return name;
        }
        public char getGrade() {
            return grade;
        }
        @Override
        public String toString() {
            return name + " : " + getGrade();
        }
    }
    public static void main(String[] args) {
        PriorityQueue<InnerStudent1> q = new PriorityQueue<>(new Comparator<InnerStudent1>() {
            @Override
            public int compare(Student1.InnerStudent1 o1, Student1.InnerStudent1 o2) {
                return o1.getGrade() - o2.getGrade();
            }
        });
        q.offer(new InnerStudent1("Hemanth", 'E'));
        q.offer(new InnerStudent1("Amrish", 'C'));
        q.offer(new InnerStudent1("Nobitha", 'F'));
        q.offer(new InnerStudent1("Deskisuki", 'A'));
        System.out.println(q);
        System.out.println(q.poll());
        System.out.println(q.poll());
        System.out.println(q.poll());
        System.out.println(q.poll());
        System.out.println(q.poll());
    }
}
// [Deskisuki : A, Amrish : C, Nobitha : F, Hemanth : E]
// Deskisuki : A
// Amrish : C
// Hemanth : E
// Nobitha : F
// null
```

7) Write a program that takes a string and returns the number of unique characters using set.

```
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class Unique {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String name = s.next();
        Set<Character> set = new HashSet<>();
        for (int i = 0; i < name.length(); i++)
            set.add(name.charAt(i));
        System.out.println("Number of unique elements are: " + set.size());
    }
}
```

```
// Enter a string: Hemanth
// Number of unique elements are: 7
```

OR
we can use the method toCharArray.

```
char[] charArray = str.toCharArray();
```

11.8) Map Interface

Map interface is a part of Collections library/ Framework but not related to Collection interface.

Stores data as key-value pairs.

Each key can map to atmost one value.

Keys are unique, but multiple keys can map to same value.

Method	Description
put(K key, V value)	→ Associates the specified value with the specified key in the map.
get(Object key)	→ Returns the value to which the specified key is mapped, or null if no mapping exists.
remove(Object key)	→ Removes the mapping for a key if it is present.
containsKey(Object key)	→ Checks if the map contains a mapping for the specified key.
keySet()	→ Returns a Set view of the keys contained in the map.
values()	→ Returns a Collection view of the values contained in the map.

```
import java.util.HashMap;
import java.util.Map;

public class MapDemo {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("A", 65);
        map.put("B", 66);
        map.put("C", 67);
        map.put("D", 68);
        System.out.println(map.size());
        System.out.println(map.get("D"));
        System.out.println(map.get("E"));
        System.out.println(map.remove("D"));
        System.out.println(map);
        System.out.println(map.containsKey("D"));
        System.out.println(map.keySet());
        System.out.println(map.values());
        for (String key : map.keySet())
            System.out.printf("%s:%d ", key, map.get(key));
    }
}
```

```
// 4
// 68
// null
// 68
// {A=65, B=66, C=67}
```

```
// false
// [A, B, C]
// [65, 66, 67]
// A:65 B:66 C:67
```

11.9) Enums

- Enums (short for enumerations) are special types of classes that represent a fixed set of constants.
- They are used when a variable can only take one out of a small set of predefined values (e.g., directions, colors, traffic lights).
- Enum constants are usually written in uppercase letters by convention.
- Enums are defined using the enum keyword.

```
enum Color {
    RED, GREEN, BLUE;
}
```

Semicolon is optional if you are no longer declaring a constructor or an attribute or any methods.

You can access enum constants using the dot (.) operator:

```
Color myColor = Color.RED;
```

values() → Returns an array of all enum constants in the order they're declared.

valueOf(String name) → Returns the enum constant with the specified name (case-sensitive). Throws IllegalArgumentException if the name doesn't match.

```
public enum TreafficLights {
    RED("Stop"), GREEN("Go"), YELLOW("Get ready to go");

    private String action;
    private TreafficLights(String action) {
        this.action = action;
    }
    public String getAction() {
        return action;
    }
}
```

```
import javax.swing.Action;

public class EnumDemo {
    public static void main(String[] args) {
        TreafficLights currentLight = TreafficLights.GREEN;
        System.out.println("current light: " + currentLight);
        System.out.println("Action: " + currentLight.getAction());
        TreafficLights red = TreafficLights.valueOf("RED");
        System.out.println("Light from valueOf(): " + red);
        System.out.println("Action: " + red.getAction());
        System.out.println("All traffic lights and actions:");
        for (TreafficLights light : TreafficLights.values())
            System.out.println(light + " -> " + light.getAction());
    }
}
```

```
// current light: GREEN
// Action: Go
```

```
// Light from valueOf(): RED
// Action: Stop
// All traffic lights and actions:
// RED -> Stop
// GREEN -> Go
// YELLOW -> Get ready to go
```

11.10) Generics & Diamond Operators

- Generics enable you to write flexible, type-safe, and reusable code.
- They allow types (classes or interfaces) to be parameters when defining classes, interfaces, and methods.
- Generics provide compile-time type checking, reducing the risk of ClassCastException.

```
ArrayList list = new ArrayList();
list.add("hello");
list.add(123);  // No error at compile time

String str = (String) list.get(1);  // Runtime error: ClassCastException
```

```
ArrayList<String> list = new ArrayList<>();
list.add("hello");
list.add(123);  // ❌Compile-time error: incompatible types
```

- With generics, explicit casting is not required, as the type is already known to the compiler.

```
ArrayList list = new ArrayList();
list.add("Java");

String language = (String) list.get(0);  // You must cast manually
```

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");

String language = list.get(0);  // No casting required – clean and safe
```

- Generics are denoted using angle brackets (<>), where you specify the type parameter (e.g., List<String> means a list that holds String values).
- The "diamond operator" (<>) specifically refers to a feature introduced in Java 7 that allows you to omit the type on the right-hand side of a declaration when it can be inferred by the compiler.

Below is the SpecificClass implementation where we can only store integers (or the defined type). To store other types, you'd need separate classes.

```
public class SpecificClass {
    private final int var;

    public SpecificClass(int var) {
        this.var = var;
    }
    public int getVar() {
        return var;
    }
}
```

Below is the generic lass where we can create any type of object , no type cast needed:The type is preserved at compile time (but erased at runtime — known as type erasure).

```
public class GenericClass<T> {
    private final T var;

    public T getVar() {
        return var;
    }
    public GenericClass(T var) {
        this.var = var;
    }
}
```

```
GenericClass<String> stringObj = new GenericClass<>("Hello");
System.out.println(stringObj.getVar()); // Output: Hello

GenericClass<Integer> intObj = new GenericClass<>(42);
System.out.println(intObj.getVar()); // Output: 42
```

Type Erasure is a process by which Java compiler removes all generic type information during compilation. This means that generic types exist only at compile time, and the compiled bytecode contains non-generic code (i.e., raw types).

CHALLENGES

1) Create an enum called Day that represents the days of the week. Write a program that prints out all the days of the week from this enum.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

```
public class DayDemo {
    public static void main(String[] args) {
        for (Day name_of_the_day : Day.values())
            System.out.println(name_of_the_day);
    }
}
```

```
// MONDAY
// TUESDAY
// WEDNESDAY
// THURSDAY
// FRIDAY
// SATURDAY
// SUNDAY
```

2) Enhance the Day enum by adding an attribute that indicates whether it is a weekday or weekend. Add a method in the enum that returns whether it's a weekday or weekend, and write a program to print out each day along with its type.

```
public enum Day {
    MONDAY("Weekday"), TUESDAY("Weekday"), WEDNESDAY("Weekday"), THURSDAY("Weekday"),
        FRIDAY("Weekday"),SATURDAY("Weekend"), SUNDAY("Weekend");
```

```
    private final String type;
    private Day(String type) {
        this.type = type;
    }
    public String getType() {
        return type;
    }
}
```

```
public class DayDemo {
    public static void main(String[] args) {
        for (Day name_of_the_day : Day.values())
            System.out.println(name_of_the_day + "->" + name_of_the_day.getType());
    }
}
```

```
// MONDAY->Weekday
// TUESDAY->Weekday
// WEDNESDAY->Weekday
// THURSDAY->Weekday
// FRIDAY->Weekday
// SATURDAY->Weekend
// SUNDAY->Weekend
```

3) Create a map where the keys are country names(as String) and the values are their capitals(also String). Populate the map with at least 5 countries and their capitals. Write a program that prompts user to enter a country name and then displays the corresponding capital, if it exists in the map.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class CountryMap {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("India", "Delhi");
        map.put("Japan", "Tokyo");
        map.put("America", "Washington DC");
        map.put("China", "Beijing");
        map.put("Australia", "Canberra");
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the name of the country: ");
        String prompt = s.nextLine();
        System.out.println("Capital is: " + map.get(prompt));
        if (map.containsKey(prompt))
            System.out.println("Capital is: " + map.get(prompt));
        else
            System.out.println("can't found in our DB");
    }
}
```

```
// Enter the name of the country: australia
// Capital is: null
// can't found in our DB
```

KEYPOINTS

- Autoboxing automatically converts the primitive type to its respective wrapper class object. For example, int to Integer, double to Double, etc.
- Not every class in the collections library implements the Collection interface, like Map. Map is part of the Java Collections Framework, but it does not implement the Collection interface.
- List maintains insertion order and allows adding elements at specified positions. Implementations like ArrayList and LinkedList maintain order and allow indexed insertion.
- In Map, keys are unique, but values can be duplicated. A key maps to exactly one value, but multiple keys can map to the same value.
- Enums in Java can implement interfaces and can also have attributes, methods, and constructors. Enums are powerful and can behave much like regular classes.
- With the diamond operator (<>), we don't need to specify the type on the right-hand side when initializing an object. This was introduced in Java 7 for cleaner and type-safe code:

```
List<String> list = new ArrayList<>();
```

- Wrapper classes in Java like Integer and Double cannot be extended because they are final classes. You cannot inherit from these classes.

12. Multi threading & Executor Service

12.1) What is a Thread?

A Thread is a small part of a program that can execute concurrently with other parts of the program.

It allows a program to perform multiple tasks simultaneously, such as handling multiple user requests or executing different tasks at the same time.

Threads can be created by either:

- Extending the Thread class, or by
- Implementing the Runnable interface.

Use threads when you need to perform independent tasks concurrently, such as managing multiple requests or executing large, time-consuming jobs.

Thread Communication:

Threads can communicate with each other to coordinate their work using methods like wait(), notify(), and notifyAll().

- wait(): Causes the current thread to release the monitor and go to the waiting state.
- notify(): Wakes up a single thread that is waiting on the monitor.
- notifyAll(): Wakes up all threads that are waiting on the monitor.

Thread and CPU Interaction:

Threads have a direct relationship with the CPU. Each thread runs on a CPU core, and if you have more threads than CPU cores, some threads may enter a waiting state until a core becomes available.

Single-threaded Program:

In a single-threaded program (or single-core system), all tasks run sequentially, with only one task being processed at a time.

The main thread, created by the Java Virtual Machine (JVM), executes the program.

Dual-Core CPUs and Beyond:

Dual-core processors (released by Intel in 2005) allow for parallel execution of multiple threads, improving performance by utilizing both cores.

Multi-threading vs. Multi-programming:

- Multi-threading: Involves multiple threads running within the same program.
- Multi-programming: Involves multiple independent programs running on the system, often sharing system resources.

Thread Behavior on Multiple Cores:

If there are 4 threads in your program and your system has 4 cores, the threads can run simultaneously across the cores.

If the number of threads exceeds the number of available CPU cores, some threads will remain in a waiting state until a core becomes available.

```
public class Main {
    public static void main(String[] args) {
        long start_time = System.currentTimeMillis();
```

```
        // task-1
        for (int i = 1; i <= 100; i++) {
            System.out.printf("%d* ", i);
        }
        System.out.println("\nCompleted * task");
        // task-2
        for (int i = 1; i <= 100; i++) {
            System.out.printf("%d$ ", i);
        }
        System.out.println("\nCompleted $ task");
        // task-3
        for (int i = 1; i <= 100; i++) {
            System.out.printf("%d# ", i);
        }
        System.out.println("\nCompleted # task");
        long end_time = System.currentTimeMillis();
        System.out.println("Total Time taken(in ms) is: " + (end_time - start_time));
    }
}
```

```
// 1* 2* 3* 4* 5* 6* 7* 8* 9* 10* 11* 12* 13* 14* 15* 16* 17* 18* 19* 20* 21* 22* 23* 24* 25*
26* 27* 28* 29* 30* 31* 32* 33* 34* 35* 36* 37* 38* 39* 40* 41* 42* 43* 44* 45* 46* 47* 48*
49* 50* 51* 52* 53* 54* 55* 56* 57* 58* 59* 60* 61* 62* 63* 64* 65* 66* 67* 68* 69* 70* 71*
72* 73* 74* 75* 76* 77* 78* 79* 80* 81* 82* 83* 84* 85* 86* 87* 88* 89* 90* 91* 92* 93* 94*
95* 96* 97* 98* 99* 100*
// Completed * task
//
1$ 2$ 3$ 4$ 5$ 6$ 7$ 8$ 9$ 10$ 11$ 12$ 13$ 14$ 15$ 16$ 17$ 18$ 19$ 20$ 21$ 22$ 23$ 24$ 25$ 26$

27$ 28$ 29$ 30$ 31$ 32$ 33$ 34$ 35$ 36$ 37$ 38$ 39$ 40$ 41$ 42$ 43$ 44$ 45$ 46$ 47$ 48$ 49$ 50
$ 51$ 52$ 53$ 54$ 55$ 56$ 57$ 58$ 59$ 60$ 61$ 62$ 63$ 64$ 65$ 66$ 67$ 68$ 69$ 70$ 71$ 72$ 73$
74$ 75$ 76$ 77$ 78$ 79$ 80$ 81$ 82$ 83$ 84$ 85$ 86$ 87$ 88$ 89$ 90$ 91$ 92$ 93$ 94$ 95$ 96$ 97
$ 98$ 99$ 100$
// Completed $ task
// 1# 2# 3# 4# 5# 6# 7# 8# 9# 10# 11# 12# 13# 14# 15# 16# 17# 18# 19# 20# 21# 22# 23# 24# 25#
26# 27# 28# 29# 30# 31# 32# 33# 34# 35# 36# 37# 38# 39# 40# 41# 42# 43# 44# 45# 46# 47# 48#
49# 50# 51# 52# 53# 54# 55# 56# 57# 58# 59# 60# 61# 62# 63# 64# 65# 66# 67# 68# 69# 70# 71#
72# 73# 74# 75# 76# 77# 78# 79# 80# 81# 82# 83# 84# 85# 86# 87# 88# 89# 90# 91# 92# 93# 94#
95# 96# 97# 98# 99# 100#
// Completed # task
// Total Time taken(in ms) is: 57
```

Need of multi threading

1. Tasks are independent of each other.
2. Multi core CPU is sitting idle most of the time.
3. Big tasks can be divided into smaller parts.
4. To make responsive code.

12.2) Creating a Thread

1) By extending Thread Class

- We can't estimate which thread completes its execution first or which thread starts first. It's purely based on your computer's thread scheduler and the CPU's workload.
- No.of classes created is not equal to no.of threads.

- The number of instances is equal to the number of threads only if the class extends Thread or implements Runnable, and you explicitly call start() on each instance.
- For example, if you create three instances of such a class and call start() on each, then three separate threads will be created and run concurrently.
- But if you simply create 3 instances of a regular class that doesn't extend Thread or implement Runnable, then you won't have 3 threads - everything will run on the main thread.
- In the below example 3 threads are created and started and then executed independently.

```
public class Task extends Thread {
    private final char symbol;

    Task(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 100; i++) {
            System.out.printf("%d%c ", i, symbol);
        }
        System.out.println("\n" + Thread.currentThread().getName() + " : Completed " + symbol
+ " task");
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Task t1 = new Task('*');
        Task t2 = new Task('$');
        Task t3 = new Task('#');
        long start_time = System.currentTimeMillis();
        t1.start();
        t2.start();
        t3.start();
        long end_time = System.currentTimeMillis();
        System.out.println("Total Time taken(in ms) is: " + (end_time - start_time));
        System.out.printf("%s : completed", Thread.currentThread().getName());
    }
}
```

```
// 1# 2# 3# 4# 5# 6# 7# 8# 9# 10# 11# 12# 13# 14# 15# 16# 17# 18# 19# 20# 21#
// 22# 23# 24# 25# 26# 27# 28# 29# 30# 31# 32# 33# 34# 35# 36# 37# 38# 39# 40#
// 41# 42# 43# 44# 45# 46# 47# 1$ 2$ 3$ 4$ 5$ 6$ 7$ 8$ 9$ 10$ 11$ 12$ 13$ 14$
// 15$ 16$ 17$ 18$ 19$ 20$ 21$ 22$ 23$ 24$ 25$ 26$ 27$ 28$ 29$ 30$ 31$ 32$ 33$
// 34$ 35$ 36$ 37$ 38$ 39$ 40$ 41$ 42$ 43$ 44$ 45$ 46$ 47$ 48$ 49$ 50$ 51$ 52$
// 53$ 54$ 55$ 56$ 57$ 58$ 59$ 60$ 61$ 62$ 63$ 64$ 65$ 66$ 67$ 68$ 69$ 70$ 71$
// 72$ 73$ 74$ 75$ 76$ 77$ 78$ 79$ 80$ 81$ 82$ 83$ 84$ 85$ 86$ 87$ 88$ 89$ 90$
// 91$ 92$ 93$ 94$ 95$ 96$ 97$ 98$ 99$ 100$ 1* 2* 3* 4* 5* 6* 7* 8* 9* 10* 11*
// 12* 13* 14* 15* 16* 17* 18* 19* 20* 21* 22* 23* 24* 25* 26* 27* 28* 29* 30*
// 31* 32* 33* 34* 35* 36* 37* 38* 39* 40* 41* 42* 43* 44* 45* 46* 47* 48* 49*
// 50* 51* 52* 53* 54* 55* 56* 57* 58* 59* 60* 61* 62* 63* 64* 65* 66* 67* 68*
// 69* 70* 71* 72* 73* 74* 75* 76* 77* 78* 79* 80* 81* 82* 83* 84* Total Time
// taken(in ms) is: 0
// 48# 49# 50# 51# 52# 53# 54# 55# 56# 57# 58# 59# 60# 61# 62# 63# 64# 65# 66#
// 67# 68# 69# 70#
```

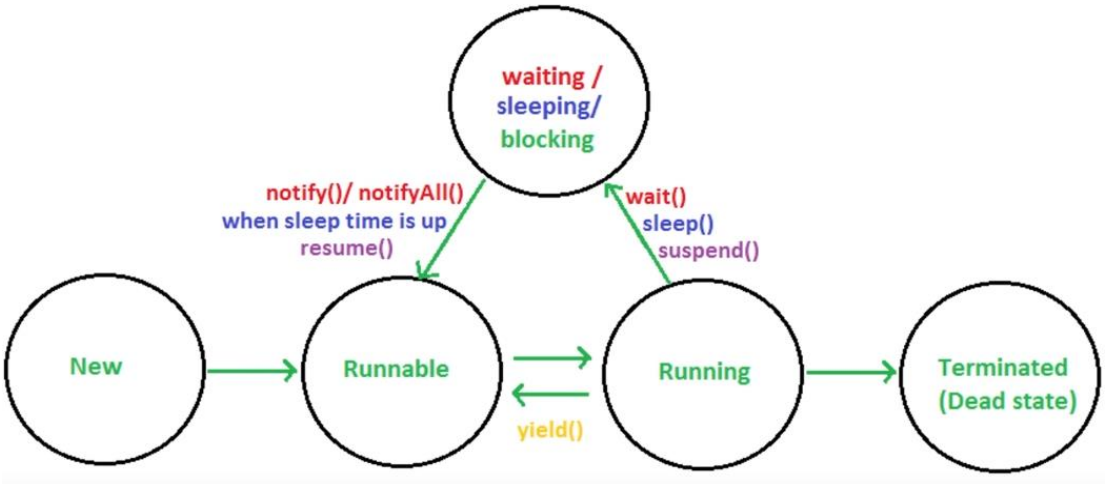
```
// Thread-1 : Completed $ task
// 85* 86* 87* 88* 89* 90* 91* 92* 93* 94* 95* 96* 97* 98* 99* 100* main :
// completed71#
// Thread-0 : Completed * task
// 72# 73# 74# 75# 76# 77# 78# 79# 80# 81# 82# 83# 84# 85# 86# 87# 88# 89# 90#
// 91# 92# 93# 94# 95# 96# 97# 98# 99# 100#
// Thread-2 : Completed # task
```

2) By implementing Runnable Interface

```
public class InterfaceDemo implements Runnable {
    private final char symbol;

    public InterfaceDemo(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.printf("%d%c ", i, symbol);
        }
        System.out.println(Thread.currentThread().getName() + " executed succesfully");
    }
}
```

```
public class MainInterface {
    public static void main(String[] args) {
        InterfaceDemo d1 = new InterfaceDemo('%');
        InterfaceDemo d2 = new InterfaceDemo('^');
        InterfaceDemo d3 = new InterfaceDemo('(');
        Thread t1 = new Thread(d1);
        Thread t2 = new Thread(d2);
        t1.start();
        t2.start();
        new Thread(d3).start();
        System.out.printf("%s executed succesfully.", Thread.currentThread().getName());
    }
}
```

12.3) States of a Thread



Thread State	Description
New	A thread is created but not yet started. It has been initialized but start() has not been called.
Runnable	The thread is ready for execution. The start() method has been called, and it is waiting to be scheduled by the JVM.
Running	The thread is actively executing its tasks. The JVM is actively scheduling the thread to run.
Blocked/Sleeping/Waiting	The thread is alive but not executing. It may be waiting for resources, locks, or other conditions to be met.
Terminated	The thread has finished execution or was stopped. It no longer runs and its lifecycle has ended.

12.4) Thread Priority

- Threads have priority levels form 1(lowest) to 10(highest), with a default value of 5.
- Thread priority just suggests the importance of a thread to the JVM's sheduler but doesn't guarantee the order of execution.
- Thread.setPriority(), Thread.getPriority() are the methods used to get and set the priority for a thread.

```
public class InterfaceDemo implements Runnable {
    private final char symbol;

    public InterfaceDemo(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.printf("%d%c ", i, symbol);
        }
        System.out.println(Thread.currentThread().getName() + " executed succesfully");
    }
}
```

```
public class MainInterface {
    public static void main(String[] args) {
        long start_time = System.currentTimeMillis();
        InterfaceDemo d1 = new InterfaceDemo('%');
        InterfaceDemo d2 = new InterfaceDemo('^');
        InterfaceDemo d3 = new InterfaceDemo('(');
        Thread t1 = new Thread(d1);
        Thread t2 = new Thread(d2);
        Thread t3 = new Thread(d3);
        t1.setPriority(Thread.MIN_PRIORITY);
        t1.start();
        t2.setPriority(Thread.MAX_PRIORITY);
        t2.start();
```

```
        t3.setPriority(Thread.NORM_PRIORITY);
        t3.start();
        long end_time = System.currentTimeMillis();
        System.out.printf("%s executed succesfully in %d ms.",
Thread.currentThread().getName(),
                (end_time - start_time));
    }
}
```

12.5) Join Method

1. Join method allows one thread to wait for the completion of another .
2. Join helps in synchronizing multiple threads, ensures that a thread completes its execution before the next steps in the calling thread proceed.
3. If t is a thread object whose thread is currently executing, t.join(); causes current thread to pause its execution until t's thread terminates.
4. Join allows overloads to allow the programmer to specify the waiting period.
   - join(long millis) -> waits for the thread to die for the specified no.of milliseconds.
   - join(long millis, int nanos) -> waits for the thread to die for the specified no.of (milliseconds + nanoseconds).
5. The join() method in Java throws a checked exception called InterruptedException.
6. Since it's a checked exception, it must be handled either using a try-catch block or by declaring throws InterruptedException in the method signature.

```
public class InterfaceDemo implements Runnable {
    private final char symbol;

    public InterfaceDemo(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.printf("%d%c ", i, symbol);
        }
        System.out.println(Thread.currentThread().getName() + " executed succesfully");
    }
}
```

```
public class JoinDemo {
    public static void main(String[] args) throws InterruptedException {
        InterfaceDemo d1 = new InterfaceDemo('A');
        InterfaceDemo d2 = new InterfaceDemo('B');
        InterfaceDemo d3 = new InterfaceDemo('C');
        Thread t1 = new Thread(d1);
        Thread t2 = new Thread(d2);
        Thread t3 = new Thread(d3);
        t1.start();
        t2.start();
        t2.join();
```

```
        t3.start();
    }
}
```

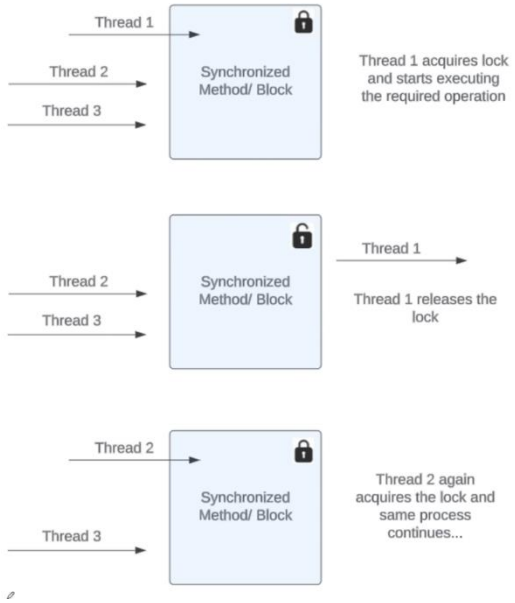
t1.start() → Starts thread t1. It runs concurrently with other threads.

t2.start() → Starts thread t2, which also runs concurrently.

t2.join() → The main thread pauses and waits for t2 to finish. This does not stop t1 - it continues executing independently.

t3.start() → Starts only after t2 has completed, because the main thread was waiting on t2.join().

12.6) Synchronize Keyword



- The synchronized keyword in Java ensures that only one thread can execute a block of code at a time, providing mutual exclusion and preventing race conditions.
- When a thread enters a synchronized block or method, it acquires a lock on the object (for instance methods) or on the class (for static methods).
- Changes made by threads inside a synchronized block are visible to all other threads.
    1. Instance method → locks the object
    2. Static method → locks the class

```
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

```
public class SynchronizedThread extends Thread {
    Counter count;
```

```
    SynchronizedThread(Counter count) {
        this.count = count;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 100000; i++) {
            count.increment();
        }
    }
}
```

```
public class SynchronizedDemo {
    public static void main(String[] args) {
        long start_time = System.currentTimeMillis();
        Counter c = new Counter();
        SynchronizedThread t1 = new SynchronizedThread(c);
        SynchronizedThread t2 = new SynchronizedThread(c);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted exception occured: " + e.getMessage());
        }

        long end_time = System.currentTimeMillis();
        System.out.printf("Total count: %d in %d ms", c.getCount(), (end_time - start_time));
    }
}
```

```
// Total count: 130823 in 8 ms
```

If we make the increment method synchronized then,

```
public class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

```
// Total count: 200000 in 20 ms
```

12.7) Thread Communication

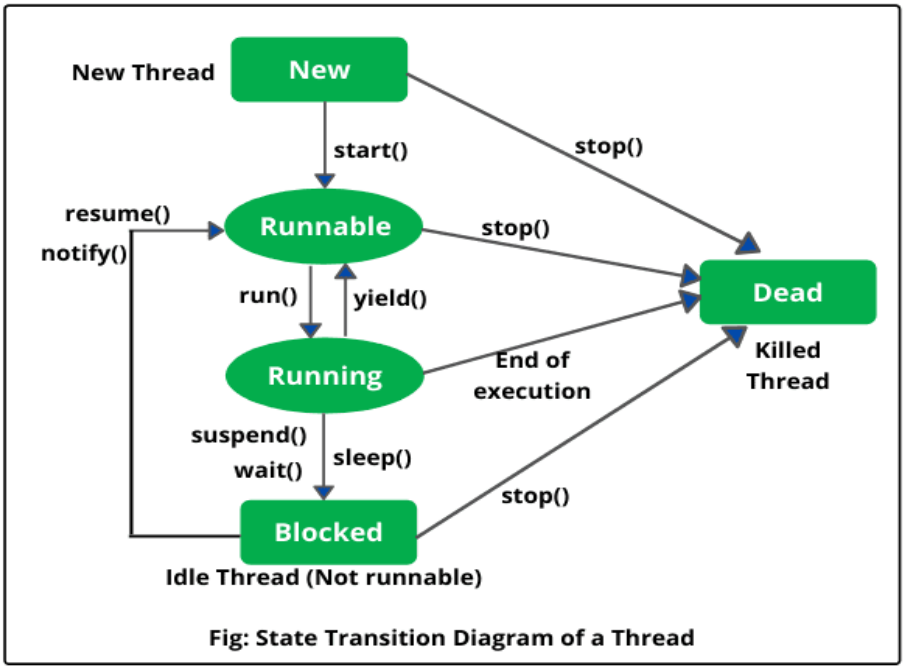


Fig: State Transition Diagram of a Thread

1. **sleep(long millis):**

   sleep should not be called on a specific thread object because it is a static method of the Thread class. This means that Thread.sleep(milliseconds) pauses the currently executing thread, regardless of which thread object it is called on. If we write Thread.sleep(timeInMilliseconds); inside the main method, the main thread will transition from the running state to the timed waiting state, and then back to the running state after the specified duration and sleep() need to handle the InterruptedException.

2. **yield():**

   Causes the currently executing threads to pause and allow other threads to execute. It's a way of suggesting that other threads of the same priority can run.

3. **wait():**

   Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object. It releases the lock held by this thread.

4. **notify():**

   Wakes up a single thread that is waiting on the objects monitor. If any threads are waiting, one is chosen to be awakened.

5. **notifyAll():**

   Wakes up all the threads that are waiting on the object's monitor.

CHALLENGES

1) Write a program that creates 2 threads. Each thread should print “Hello form Thread X”, where X is the no.of the thread(1 or 2), 10 times then terminate.

```
public class HelloThread extends Thread {
    private final int threadNo;

    public HelloThread(int threadNo) {
        this.threadNo = threadNo;
    }
    public int getThreadNo() {
        return threadNo;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.printf("(%d)Hello form Thread-%d\n", (i + 1), threadNo);
        }
    }
}
```

```
public class Challenge1 {
    public static void main(String[] args) {
        HelloThread t1 = new HelloThread(1);
        HelloThread t2 = new HelloThread(2);
        t1.start();
        t2.start();
    }
}
```

If we call t1.run() instead of t1.start(), the run() method will be executed just like a normal method call, and it will not start a new thread. As a result, the code inside run() will execute in the current thread, which is typically the main thread. So Thread.currentThread().getName() will return main.

```
// (1)Hello form Thread-2
// (2)Hello form Thread-2
// (3)Hello form Thread-2
// (4)Hello form Thread-2
// (5)Hello form Thread-2
// (6)Hello form Thread-2
// (1)Hello form Thread-1
// (2)Hello form Thread-1
// (3)Hello form Thread-1
// (4)Hello form Thread-1
// (5)Hello form Thread-1
// (6)Hello form Thread-1
// (7)Hello form Thread-1
// (8)Hello form Thread-1
// (9)Hello form Thread-1
// (10)Hello form Thread-1
// (7)Hello form Thread-2
// (8)Hello form Thread-2
// (9)Hello form Thread-2
// (10)Hello form Thread-2
```

2) Write a program that starts a thread and prints its state after each significant event (creation, starting and termination). use Thread.sleep() to simulate long-running tasks and Thread.getState() to print the thread's state.

```
public class PrintOdd extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 100; i += 2) {
            System.out.print(i + " ");
        }
    }
}
```

```
public class Challenge2 {
    public static void main(String[] args) throws InterruptedException {
        PrintOdd p1 = new PrintOdd();
        System.out.println(p1.getState());
        p1.start();
        System.out.println(p1.getState());
        p1.join();
        System.out.println(p1.getState());
    }
}
```

```
// NEW
// RUNNABLE
// 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53
// 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
// TERMINATED
```

Note:

The run() method (used in threads) is not allowed to say it throws a checked exception like InterruptedException. This is because the original run() method in the Runnable interface doesn't throw any checked exceptions.

So, if you use Thread.sleep() inside the run() method — which can throw InterruptedException — you must catch that exception using a try-catch block.

```
public void run() throws InterruptedException { // ✕Not allowed
    Thread.sleep(1000);
}
```

```
public void run() {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e); // Wrap it in an unchecked exception if needed
    }
}
```

So basically:

- run() can't use throws InterruptedException.
- You have to handle the exception inside the method using try-catch.

- If you want to stop or report the error, you can wrap it in a RuntimeException and throw that.

3) Create 3 threads. Ensure that 2nd thread starts only after the 1st thread ends and the 3rd starts only after the 2nd thread ends using the join method. Each thread should print its start and end along with its name.

```
public class Symbols extends Thread {
    private final char symbol;

    public Symbols(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        System.out.printf("%s is started", symbol);
        for (int i = 1; i <= 100; i++) {
            System.out.print(symbol + " ");
        }
        System.out.printf("Thread %c is ended", symbol);
    }
}
```

```
public class Challenge3 {
    public static void main(String[] args) throws InterruptedException {
        Symbols s1 = new Symbols('0');
        Symbols s2 = new Symbols('1');
        Symbols s3 = new Symbols('3');
        System.out.println(s1.getName() + " is getting invoked by start method");
        s1.start();
        s1.join();
        System.out.println(s2.getName() + " is getting invoked by start method");
        s2.start();
        s2.join();
        System.out.println(s3.getName() + " is getting invoked by start method");
        s3.start();

    }
}
```

```
// Thread-0 is getting invoked by start method
// 0 is started0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Thread 0 is
// endedThread-1 is getting invoked by start method
// 1 is started1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
// 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
// 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 Thread 1 is
// endedThread-2 is getting invoked by start method
// 3 is started3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
// 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
// 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 Thread 3 is ended
```

4) Simulate a traffic signal using threads. Create 3 threads representing three signals: RED, YELLOW, GREEN. Each signal should be on for a certain time, then switch to the next signal in order. Use sleep for timing and synchronize to make sure only one signal is active at a time.

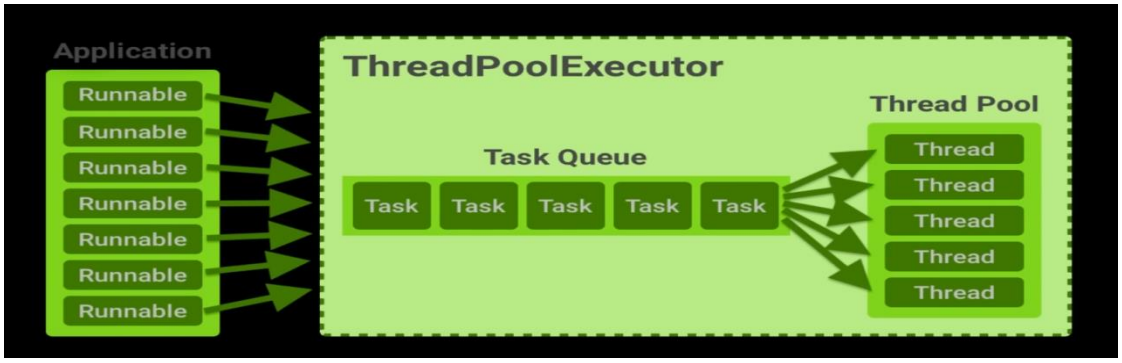
```
public enum TrafficEnum {
    RED("Stop"), YELLOW("turn on the engine"), GREEN("You can goooooo");

    final String action;
    TrafficEnum(String action) {
        this.action = action;
    }
    public String getAction() {
        return action;
    }
}
```

```
public class TrafficThread extends Thread {
    String light;

    TrafficThread(String light) {
        this.light = light;
    }
    @Override
    public void run() {
        System.out.println(TrafficEnum.valueOf(light).getAction());
    }
}
```

```
public class Challenge4 {
    public static void main(String[] args) throws InterruptedException {
        TrafficThread t1 = new TrafficThread("RED");
        TrafficThread t2 = new TrafficThread("GREEN");
        TrafficThread t3 = new TrafficThread("YELLOW");
        t1.start();
        Thread.sleep(10000);
        t3.start();
        Thread.sleep(5000);
        t2.start();

    }
}
```

```
// Stop
// turn on the engine
// You can goooooo
```

12.8) Intro to Executor Service



- ExecutorService is an interface in the Java Concurrency API that provides a powerful framework for managing and executing tasks asynchronously, without the need to manually handle thread creation and management.
- Instead of creating a new thread for each task, ExecutorService maintains a pool of reusable threads, which significantly improves performance — particularly when executing many short-lived tasks.
- This approach reduces the overhead associated with thread creation and destruction and allows for better control over resource usage.
- In simple terms, the Java Concurrency API enables multithreading, and ExecutorService is one of its core tools for handling tasks in a clean, efficient, and scalable way.
- In Java, threads managed by an ExecutorService do not die immediately after completing a task. Instead, they remain alive in a thread pool and wait to execute more tasks — until the executor is explicitly shut down.
- It's similar to a waiter in a restaurant:

  We don't fire the waiter after serving a single customer — they continue serving more customers as they arrive.
- Likewise, in ExecutorService, threads are reused to handle multiple tasks efficiently, rather than being destroyed and recreated for each one.
- This thread reuse significantly reduces performance overhead and provides better resource management compared to manually creating and starting a new thread for every task.

Demo of *newSingleThreadExecutor()* is shown below which executes only one thread at a time..

```
public class Symbols extends Thread {
    private final char symbol;

    public Symbols(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        System.out.printf("\n%s with the symbol:%s is started\n",
Thread.currentThread().getName(), symbol);
        for (int i = 1; i <= 100; i++) {
            System.out.print(symbol + " ");
        }
        System.out.printf("\n%s with the symbol:%s is ended\n",
Thread.currentThread().getName(), symbol);
    }
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Executor {
    public static void main(String[] args) {
        ExecutorService service = Executors.newSingleThreadExecutor();
```

```java
        Symbols s1 = new Symbols('$');
        Symbols s2 = new Symbols('%');
        service.submit(s1);
        service.submit(s2);
        service.shutdown();
    }
}
```

```
// pool-1-thread-1 with the symbol:$ is started
// $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
// $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
// $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
// pool-1-thread-1 with the symbol:$ is ended
// pool-1-thread-1 with the symbol:% is started
// % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
// % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
// % % % % % % % % % % % % % % % % % % % % % %
// pool-1-thread-1 with the symbol:% is ended
```

12.9) Multiple Threads with executor

Let us use multiple threads. In the below example we have used 2 threads to complete 3 tasks.

Thread 2 had done the 2 tasks that are 1st and 3rd , and Thread1 had only completed 2nd task.

```java
public class Symbols extends Thread {
    private final char symbol;

    public Symbols(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        System.out.printf("\n%s with the symbol:%s is started\n",
Thread.currentThread().getName(), symbol);
        for (int i = 1; i <= 100; i++) {
            System.out.print(symbol + " ");
        }
        System.out.printf("\n%s with the symbol:%s is ended\n",
Thread.currentThread().getName(), symbol);
    }
}
```

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Executor {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(2);
        Symbols s1 = new Symbols('$');
        Symbols s2 = new Symbols('%');
        Symbols s3 = new Symbols('=');
        service.submit(s1);
        service.submit(s2);
        service.submit(s3);
        service.shutdown();
    }
}
```

```
// pool-1-thread-2 with the symbol:% is started
// pool-1-thread-1 with the symbol:$ is started
// $ $ $ % % % % % % % % $ % % % % % % % % % % % % % % % % % % % $ $ $
// $ $ % $ % $ $ $ % % % $ % % % $ % % % $ $ $ $ $ % % % % % % % % $ % % % % $ %
// % % $ % $ $ % $ $ $ $ % % % % % $ $ $ % $ % $ $ % % % % % % % % % $ $ $ $ $
// $ $ $ % % % % % % $ % % $ $ $ $ % $ $ % $ % % % % % $ $ $ % % % $ % $ $ $ $
// % % % $ $
// pool-1-thread-2 with the symbol:% is ended
// $ $ $ $ $ $ $ $
// pool-1-thread-2 with the symbol:= is started
// = = $ $ $ $ = = $ = = = = = = = = $ $ = = $ $ $ $ = = = = $ $ $ $ $ $ $ $ $
// $ $ $ $ $ $ = = $ = = = = = = = = = = = = = = = = $ $ = = $ =
// pool-1-thread-1 with the symbol:$ is ended
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// pool-1-thread-2 with the symbol:= is ended
```

And lets see an example where we shutdown the executor within 10 seconds regardless of weather executor fully executed or not.

```
public class Symbols extends Thread {
    private final char symbol;

    public Symbols(char symbol) {
        this.symbol = symbol;
    }
    public char getSymbol() {
        return symbol;
    }
    @Override
    public void run() {
        System.out.printf("\n%s with the symbol:%s is started\n",
Thread.currentThread().getName(), symbol);
        for (int i = 1; i <= 100; i++) {
            System.out.print(symbol + " ");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.printf("\n%s with the symbol:%s is ended\n",
Thread.currentThread().getName(), symbol);
    }
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Executor {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(2);
        Symbols s1 = new Symbols('$');
        Symbols s2 = new Symbols('%');
        Symbols s3 = new Symbols('=');
        service.submit(s1);
        service.submit(s2);
        service.submit(s3);
        service.shutdown();
```

```
        if (service.awaitTermination(10, TimeUnit.SECONDS)) {
            System.out.println("Completed under 10 seconds");
        } else {
            System.out.println("Haven't completed under 10 seconds");
            service.shutdownNow();
        }
    }
}
```

```
// pool-1-thread-1 with the symbol:$ is started

// pool-1-thread-2 with the symbol:% is started
// $ % $ % $ % % $ $ % $ % % $ % $ $ % % $ $ % % $ $ % $ % $ % % $ $ % % $ % %
// $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ $
// % $ %
// $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ $
// % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ %
// $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % $ % Haven't completed under 10
// seconds
// $
```

12.10) Returning Futures

- In Java, we can use the Callable interface (instead of Runnable) when we want a thread to return a result or throw a checked exception.
- The result of a Callable task is wrapped in a Future object.
- The Future represents the result of an asynchronous computation — which may or may not be available yet.
- You submit a Callable to an ExecutorService using the submit() method, which returns a Future.
- You can then use future.get() to retrieve the result once the task is complete.

Key Difference:

- Runnable does not return a value.
- Callable<T> returns a value of type T, which can be accessed using a Future<T>.

```
import java.util.concurrent.Callable;

public class Name implements Callable<String> {
    private final String name;
    public Name(String name) {
        this.name = name;
    }
    @Override
    public String call() throws Exception {
        System.out.printf("Getting %s.......\n", name);
        Thread.sleep(4000);
        return name + " was returned";
    }
}
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureDemo {
```

```
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService service = Executors.newFixedThreadPool(2);
        Name task1 = new Name("hemanth");
        Name task2 = new Name("Java");
        Name task3 = new Name("KG Coding");
        Future<String> name1 = service.submit(task1);
        Future<String> name2 = service.submit(task2);
        Future<String> name3 = service.submit(task3);
        System.out.println("\n" + name1.get());
        System.out.println("\n" + name2.get());
        System.out.println("\n" + name3.get());
        service.shutdown();
    }
}
```

```
// Getting Java.......Getting hemanth.......hemanth was returned
// Java was returned
// Getting KG Coding.......KG Coding was returned
```

CHALLENGES

1) Write a program that creates a single-threaded executor service. Define and submit a simple Runnable task that prints numbers from 1-10. After submission, shutdown the executor.

```
public class Print1210 implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Challenge1InExecutor {
    public static void main(String[] args) {
        ExecutorService service = Executors.newSingleThreadExecutor();
        Print1210 task1 = new Print1210();
        service.submit(task1);
        service.shutdown();
    }
}
```

```
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
// 10
```

OR

```
import java.util.concurrent.ExecutorService;
```

```java
import java.util.concurrent.Executors;

public class Challenge1InExecutor {
    public static void main(String[] args) {
        try (ExecutorService service = Executors.newSingleThreadExecutor()) {
            Print1210 task = new Print1210();
            service.submit(task);
        }
        // we need not to explicitly shutdown the service as we are using try with
        // resources.
    }
}
```

2) Create a fixed threadpool with specified no.of threads using Executors.newFixedThreadPool(int). submit multiple tasks to this executor, where each task should print the current threads name and sleep for a random time between 1 and 5 seconds. Finally, shutdown the executor and handle proper termination using awaitTermination.

```java
public class ThreadOfChallenge2 extends Thread {
    @Override
    public void run() {
        System.out.println("Name is: " + Thread.currentThread().getName());
        try {
            Thread.sleep(((int) (Math.random() * 5) + 1) * 1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```java
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Challenge2Executor {
    public static void main(String[] args) throws InterruptedException {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter how many threads you need to complete this task: ");
        int count = s.nextInt();
        ExecutorService service = Executors.newFixedThreadPool(count);
        ThreadOfChallenge2 task1 = new ThreadOfChallenge2();
        ThreadOfChallenge2 task2 = new ThreadOfChallenge2();
        ThreadOfChallenge2 task3 = new ThreadOfChallenge2();
        ThreadOfChallenge2 task4 = new ThreadOfChallenge2();
        ThreadOfChallenge2 task5 = new ThreadOfChallenge2();
        service.submit(task1);
        service.submit(task2);
        service.submit(task3);
        service.submit(task4);
        service.submit(task5);
        service.shutdown();
        if (service.awaitTermination(10, TimeUnit.SECONDS)) {
            System.out.println("Completed in 10 seconds");
        } else {
            System.out.println("Haven't completed in 10 seconds");
            service.shutdownNow();
```

```
        }
    }
}
```

```
// Enter how many threads you need to complete this task: 2
// Name is: pool-1-thread-2
// Name is: pool-1-thread-1
// Name is: pool-1-thread-1
// Name is: pool-1-thread-2
// Name is: pool-1-thread-2
// Completed in 10 seconds
```

OR

```
public class ThreadOfChallenge2 extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is started");
        try {
            Thread.sleep(((int) (Math.random() * 5) + 1) * 1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(Thread.currentThread().getName() + " is ended");
    }
}
```

```
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Challenge2Executor {
    public static void main(String[] args) throws InterruptedException {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter how many threads you need to complete this task: ");
        int count = s.nextInt();
        ExecutorService service = Executors.newFixedThreadPool(count);
        for (int i = 1; i <= 5; i++) {
            ThreadOfChallenge2 task = new ThreadOfChallenge2();
            service.submit(task);
        }
        service.shutdown();
        if (service.awaitTermination(10, TimeUnit.SECONDS)) {
            System.out.println("Completed in 10 seconds");
        } else {
            System.out.println("Haven't completed in 10 seconds");
            service.shutdownNow();
        }
    }
}
```

```
// Enter how many threads you need to complete this task: 3
// pool-1-thread-2 is started
// pool-1-thread-3 is started
// pool-1-thread-1 is started
// pool-1-thread-1 is ended
// pool-1-thread-2 is ended
// pool-1-thread-2 is started
// pool-1-thread-1 is started
// pool-1-thread-2 is ended
```

```
// pool-1-thread-3 is ended
// pool-1-thread-1 is ended
// Completed in 10 seconds
```

3) Write a program that uses an executor service to execute multiple tasks. Each task should calculate and return the factorial of a number provided to it. Use future objects to receive the results of the calculations. After all tasks are submitted, retrieve the results from the future, print them, and ensure the executor service is shutdown correctly.

```
import java.util.concurrent.Callable;

public class Factorial implements Callable<Long> {
    private final int num;
    public Factorial(int num) {
        this.num = num;
    }
    @Override
    public Long call() throws Exception {
        long fact = 1;
        if (num == 0) {
            return fact;
        } else {
            for (int i = 1; i <= num; i++) {
                fact *= i;
            }
        }
        return fact;
    }
}
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Challenge3Executor {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService service = Executors.newFixedThreadPool(3);
        Factorial task1 = new Factorial(10);
        Factorial task2 = new Factorial(9);
        Factorial task3 = new Factorial(5);
        Future<Long> answer1 = service.submit(task1);
        Future<Long> answer2 = service.submit(task2);
        Future<Long> answer3 = service.submit(task3);
        System.out.println(answer1.get());
        System.out.println(answer2.get());
        System.out.println(answer3.get());
        service.shutdown();
    }
}
```

```
// 3628800
// 362880
// 120
```

OR

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

```java
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class Challenge3Executor {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        List<Future<Long>> list = new ArrayList<>();
        ExecutorService service = Executors.newFixedThreadPool(3);
        for (int i = 1; i <= 10; i++) {
            Factorial task = new Factorial(i);
            list.add(service.submit(task));
        }
        for (Future<Long> result : list) {
            System.out.println(result.get());
        }
        service.shutdown();
    }
}
```

```
// 1
// 2
// 6
// 24
// 120
// 720
// 5040
// 40320
// 362880
// 3628800
```

KEY POINTS

1. Main thread is the only user created thread that is created automatically when a program starts.
2. run() is automatically called when thread is started using the start().
3. A thread can not be in both runnable and running states.
4. join() methods causes the calling thread to stop execution until the thread it joins with, stops running.
5. Synchronized keyword prevents 2 threads from executing a method simultaneously.
6. notify() wakes up single thread, notifyAll() wakes up all the threads.
7. Executor service must be explicitly shutdown to terminate the threads it manages.
8. Thread.yield() is a static method in Java that hints to the thread scheduler that the current thread is willing to pause its execution temporarily, allowing other threads of the same or higher priority a chance to execute.

13. Functional Programming

13.1) What is Functional programming

1) It's a way of writing programs where we use functions as small building blocks.
2) Functions can be passed as arguments, returned from other functions, and assigned to variables.
3) Data is immutable.
4) Pure functions, always gives same results for same inputs known as the pure functions.
5) Supports functional interfaces, these are like templates for functions, making it easier to use them in different parts of the program.

13.2) Lambda Expressions

- Theses are quick and nameless functions for small tasks.
- Written as (parameters) -> {body}
- They work with interfaces that have only one method, making code concise.
- Great for managing collections like lists and sets, like filtering and sorting.

13.3) What is a stream

- Streams represent a sequence of elements.
- We can perform operations like map, filter and reduce on streams.
- Streams don't store data, they process it on on-the-fly from the sources like collections or arrays.
- Stream operations can be lazy, processes elements as they needed, which is efficient for large data.
- They support parallel processing.
- Streams are consumable and can not be reused once the stream has been used with an terminal operation.

13.4) Filtering & Reducing

Filtering

- Used to filter elements of a stream based on a given predicate(a condition).
- Those elements that satisfy the condition will be mentioned in the result.
- It's a lazy operation, meaning it doesn't execute until a terminal operation(like collect or forEach) is invoked on the stream.
- Filter returns a new stream with the elements that match the predicate.
- forEach: A terminal operation that performs an action for each remaining element. It does not return a stream or any value.

```
Stream<String> newFruits=fruits.stream().filter(fruit -> fruit.endsWith("e"));
```

The above will not be executed and doesn't store those fruits that ends with "e" as there is no terminal method there.

```
List<String> fruits = List.of("Orange", "Apple", "Banana");
System.out.print("listing normally: ");
for (String fruit : fruits)
```

```
    System.out.print(fruit + " ");
```

listing normally: Orange Apple Banana

```
List<String> fruits = List.of("Orange", "Apple", "Banana");
System.out.print("\nlisting using streams: ");
fruits.stream().forEach(new Consumer<String>() {
            @Override
            public void accept(String fruit) {
                System.out.print(fruit + " ");

            }
    });
```

listing using streams: Orange Apple Banana

```
List<String> fruits = List.of("Orange", "Apple", "Banana");
System.out.print("\nlisting using streams and lambda: ");
fruits.stream().forEach(fruit -> System.out.print(fruit + " "));
```

listing using streams and lambda: Orange Apple Banana

```
List<String> fruits = List.of("Orange", "Apple", "Banana");
System.out.print("\nThose fruits that name ends with 'e': ");
fruits.stream().filter(fruit -> fruit.endsWith("e"))
        .forEach(fruit -> System.out.print(fruit + " "));
```

Those fruits that name ends with 'e': Orange Apple

Reduce

- Used to reduce the elements of stream to a single value.
- It takes the binary operator as a parameter and applies it repeatedly, combining the elements of stream.
- Without an identity value reduce returns an optional.
- With an identity value, it returns a default value if the stream is empty.

```
import java.util.Optional;
import java.util.stream.Stream;

public class ReduceExample {
    public static void main(String[] args) {
        // Without identity
        Optional<Integer> sum = Stream.of(1, 2, 3, 4).reduce((a, b) -> a + b);
        System.out.println("Sum without identity: " + sum);
        // With identity
        int sumWithIdentity = Stream.of(1, 2, 3, 4).reduce(0, (a, b) -> a + b);
        System.out.println("Sum with identity: " + sumWithIdentity);
    }
}
```

```
// Sum without identity: Optional[10]
// Sum with identity: 10
```

```
import java.util.List;
import java.util.function.BinaryOperator;

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

```
        List<Integer> nums = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        int sum = 0;
        for (int num : nums)
            sum += num;
        System.out.println("Sum using normal foreach: " + sum);
        int newSum = nums.stream().reduce(0, new BinaryOperator<Integer>() {
            @Override
            public Integer apply(Integer t, Integer u) {
                return t + u;
            }
        });
        System.out.println("Sum using reduce and streams: " + newSum);
        int newNewSum = nums.stream().reduce(0, (a, b) -> a + b);
        System.out.println("sum using lambda, reduce and streams: " + newNewSum);
        int max = nums.stream().reduce(Integer.MIN_VALUE, (a, b) -> a > b ? a : b);
        System.out.println("Max of the list is " + max);
    }
}
```

```
// Sum using normal foreach: 45
// Sum using reduce and streams: 45
// sum using lambda, reduce and streams: 45
// Max of the list is 9
```

CHALLENGES

1) Write a lambda that takes 2 integers and returns their multiplication. Then, apply this lambda to a pair of numbers.

```
import java.util.function.BinaryOperator;

public class Challenge1 {
    public static void main(String[] args) {
        BinaryOperator<Integer> obj = (a, b) -> a * b;
        System.out.println("Mul: " + obj.apply(10, 4));
    }
}
```

```
// Mul: 40
```

OR

```
public interface FunctionalInterface {
    int hemanth(int a, int b);
}
```

```
public class Challenge1 {
    public static void main(String[] args) {
        FunctionalInterface obj = (a, b) -> a * b;
        System.out.println("Mul: " + obj.hemanth(5, 6));
    }
}

// Mul: 30
```

2) Convert an array of strings into a stream. Then, use the stream to print each string to the console.

```
import java.util.stream.Stream;
```

```
public class Challenge2 {
    public static void main(String[] args) {
        String[] strs = { "one", "two", "three", "four" };
        Stream<String> strsStream = Stream.of(strs);
        strsStream.forEach(str -> System.out.print(str + " "));
    }
}
```

```
// one two three four
```

3) Given a list of strings, use stream operations to filter-out strings that have length of 10 or more and then concatenate the remaining strings.

```
import java.util.List;

public class Challenge3 {
    public static void main(String[] args) {
        List<String> list = List.of("1234", "123456", "12345678910", "1234567891011");
        String str = list.stream().filter(name -> !(name.length() >= 10)).reduce("", (a, b) ->
a + " " + b);
        System.out.println(str);
    }
}
```

```
// 1234 123456
```

OR

```
import java.util.List;

public class Challenge3 {
    public static void main(String[] args) {
        List<String> list = List.of("1234", "123456", "12345678910", "1234567891011");
        System.out.print("Those len>10: ");
        list.stream().filter(name -> name.length() >= 10).forEach(name ->
System.out.print(name + " "));
        System.out.print("\nconactinating the remaining: ");
        list.stream().filter(name -> !(name.length() >= 10)).forEach(name ->
System.out.print(name));
    }
}
```

```
// Those len>10: 12345678910 1234567891011
// conactinating the remaining: 1234123456
```

4) Given a list of integers, use stream operations to filter odd numbers and print them.

```
import java.util.List;
import java.util.stream.Stream;

public class Challenge4 {
    public static void main(String[] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        System.out.print("Odds are: ");
        Stream<Integer> odds = nums.stream().filter(num -> num % 2 == 1);
        odds.forEach(num -> System.out.print(num + " "));
    }
}
```

```
// Odds are: 1 3 5 7 9
```

13.5) Functional Interfaces

- It has only 1 abstract method, it can have multiple default or static methods.
- They are intended to use with lamda expressions.
- @FunctionalInterface annotation is not mandatory but helps the compiler to identify.
- Predicate, Consumer, BinaryOperator, Runnable, Callable, Comparator are the user-defined intefaces can be functional if they have one abstract method.

13.6) Method references

- Described using ::
- Ex → System.out::println refers to the println method of the System.out object.
- These are used with functional interfaces.
- Makes code readable and concise.
- Can only be used for methods that fit the parameters and return type.
- Syntax:

Static method → refrences ClassName::staticMethodName

Instance method → Instance::instanceMethodName

Instance Method particular class → ClassName::MethodName

Constructor references → ClassName:: new

```java
import java.util.List;

public class MethodReferences {
    public static void main(String[] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        System.out.println("Normal displyaing: ");
        nums.stream().filter(num -> num % 2 == 1).forEach(num -> System.out.println(num));
        System.out.println("Display using Method references: ");
        nums.stream().filter(num -> num % 2 == 1).forEach(System.out::println);
        int add = nums.stream().reduce(0, Integer::sum);
        System.out.println("Addition: " + add);
    }
}
```

```
// Normal displyaing:
// 1
// 3
// 5
// 7
// 9
// Display using Method references:
// 1
// 3
// 5
// 7
// 9
// Addition: 45
```

13.7) Functional vs. Structural Programming

Aspect	Imperative (Structural)	Declarative (Functional)
1. Computation	You tell the computer how to do things, step by step.	You tell the computer what to do, and it figures out the steps.
2. Readability	Easy to follow steps, but can get long and complex as code grows.	Shorter and cleaner, easier to read once you understand it.
3. Customization	You have full control over the code and structure.	Harder to customize due to fixed structure or hidden logic.
4. Optimization	Harder to optimize because of too much control and complexity.	Easier to optimize and manage with simpler structure.
5. Structure	Code structure is detailed and specific (what to do, and how).	Code structure is simple and focused (just what to do).

SQL is an example of declarative paradigm.

C is an procedure oriented programming language and java is an object-oriented programming language.

13.8) Optional class

- We can create optional objects Optional.empty(), Optional.of(), Optional.ofNullable()
- To check the value is present or not we can use isPresent() and ifPresent()
- Default values orElse() and orElseGet()
- For Value Transformation we can use map()
- For throwing exceptions we can use orElseThrow()

```
import java.util.Optional;

public class OptionalClassDemo {
    public static void main(String[] args) {
        Optional<String> emptyOptional = Optional.empty();
        Optional<String> ofOptional = Optional.of("Hemanth");
        Optional<String> nullOptional = Optional.ofNullable(null);
        if (ofOptional.isPresent())
            System.out.println(ofOptional.get()); // Hemanth
        String res = emptyOptional.orElse("Java");
        System.out.println(res); // Java
        ofOptional.ifPresent(System.out::println); // Hemanth
    }
}
```

```
import java.util.List;
import java.util.Optional;

public class OptionalDemo {
    public static void main(String[] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        Optional<Integer> add = nums.stream().reduce(Integer::sum);
        if (add.isPresent())
            System.out.println(add.get()); // 45
```

```
        else
            System.out.println("Not found!");
    }
}
```

CHALLENGES

1) create your own functional Interface with a single abstract method that accepts an integer and returns a boolean. Implement using a lambda that checks if the number is prime.

```
public interface PrimeOrNot {
    boolean isPrime(int num);
}
```

```
public class Challengee1 {
    public static void main(String[] args) {
        PrimeOrNot obj = num -> {
            int cnt = 0;
            for (int i = 1; i <= num; i++) {
                if (num % i == 0)
                    cnt++;
            }
            if (cnt == 2)
                return true;
            return false;
        };
        System.out.println(obj.isPrime(3));
    }
}
```

2) Write 2 versions of a program that calculates the factorial of a number: one using structural programming and other using functional programming.

```
import java.util.Scanner;
import java.util.stream.IntStream;

public class Challengeee2 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = s.nextInt();
        int res = IntStream.rangeClosed(1, num).reduce(1, (a, b) -> a * b);
        System.out.println(res);
        // OR
        IntStream.range(1, num + 1).reduce((a, b) -> a * b).ifPresent(System.out::println);
    }
}
```

```
// Enter a number: 5
// 120
// 120
```

OR

```
public class Challengee2 {
    public static void main(String[] args) {
        Challengee2 obj = new Challengee2();
        System.out.println(obj.factorial(5));
    }
```

```
    int factorial(int num) {
        if (num == 0 || num == 1)
            return num;
        else
            return num * factorial(num - 1);
    }
}
```

AND

```
public interface FactInterface {
    int fact(int num);
}
```

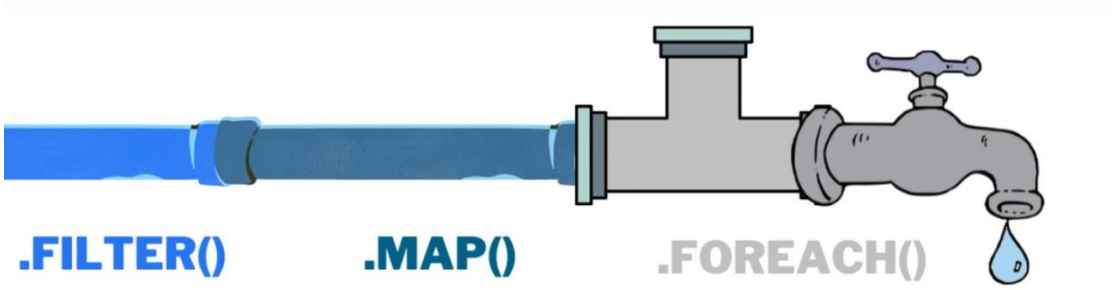
```
public class Challengee2 {
    public static void main(String[] args) {
        FactInterface obj = num -> {
            int fact = 1;
            if (num == 0 || num == 1)
                return num;
            else {
                for (int i = 1; i <= num; i++)
                    fact *= i;
            }
            return fact;
        };
        System.out.println(obj.fact(5));
    }
}
```

3) Write a function that accepts a string and returns an Optional<String>. If string is empty or null, return an empty optional, otherwise, return an optional containing the uppercase version of a string.

```
import java.util.Optional;

public class Challengee3 {
    public static void main(String[] args) {
        Challengee3 obj = new Challengee3();
        Optional<String> res = obj.method("hi");
        res.ifPresent(System.out::println);
        System.out.println(obj.method(null));
        System.out.println(obj.method(""));
    }
    Optional<String> method(String name) {
        if (name == null || name.isEmpty())
            return Optional.empty();
        return Optional.of(name.toUpperCase());
    }
}
```

```
// HI
// Optional.empty
// Optional.empty
```

13.9) Intermediate vs. Terminal Operations



Intermediate Operations

- Intermediate operations transform a stream into another stream.
- They do not execute immediately — they're lazy and only run when a terminal operation (like collect(), forEach(), etc.) is called.

Feature	Description
Lazy Execution	Only processed when a terminal operation is invoked.
Chainable	You can connect multiple intermediate operations (e.g., .filter().map()).
Transform Stream	They take a stream and return a new one (not a final result).
Stateless Operations	Do not rely on previously seen elements (e.g., map, filter).
Stateful Operations	May need to remember or compare elements (e.g., sorted, distinct).

```
List<String> names = List.of("John", "Jane", "Jack");

names.stream()
     .filter(name -> name.startsWith("J"))   // intermediate (stateless)
     .map(String::toUpperCase)               // intermediate (stateless)
     .sorted()                               // intermediate (stateful)
     .forEach(System.out::println);          // terminal
```

Terminal operations

- Initiates the stream processing and closes the stream and after this stream cannot be used.
- Produces a result(like a sum or list).
- It is not chainable.
- collect, forEach, reduce, sum, max, min, count are some examples of terminal.

13.10) Max, Min, Collect to list

max() → finds the largest element in the stream according to a given comparator or natural ordering.

min() → finds the smallest element in the stream according to a given comparator or natural ordering.

collect(Collectors.toList()) → gathers all the elements of the stream into a new list.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Program13_10 {
    public static void main(String[] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

```
        System.out.println(nums.stream().max(Integer::compareTo));
        nums.stream().min(Integer::compareTo).ifPresent(System.out::println);
        List<String> wordings = Arrays.asList("Hi", "My", "Name", "Is", "Hemanth");
        System.out.println(wordings.stream().collect(Collectors.toList()));
    }
}
```

```
// Optional[9]
// 1
// [Hi, My, Name, Is, Hemanth]
```

13.11) Sort, Distinct, Map

sorted() → orders the elements of a stream based on their natural order or a provided comparator.

distinct() → filters out duplicate elements.

map() → applies a function to every element of a stream.

```
import java.util.List;
import java.util.stream.Collectors;

public class Program13_11 {
    public static void main(String[] args) {
        List<String> fruits = List.of("apple", "banana", "grapes", "pineapple", "apple",
"orange", "grapes");
        List<String> sortedFruits = fruits.stream().sorted().collect(Collectors.toList());
        System.out.println(sortedFruits);
        List<String> distinctFruits =
sortedFruits.stream().distinct().collect(Collectors.toList());
        System.out.println(distinctFruits);
        distinctFruits.stream().map(String::toUpperCase).forEach(System.out::println);
    }
}
```

```
// [apple, apple, banana, grapes, grapes, orange, pineapple]
// [apple, banana, grapes, orange, pineapple]
// APPLE
// BANANA
// GRAPES
// ORANGE
// PINEAPPLE
```

CHALLENGES

1. Given an array of integers, create a stream, use the distinct operation to remove duplicates, and collect the result into a new list.

```
import java.util.List;
import java.util.stream.Collectors;

public class Challengi1 {
    public static void main(String[] args) {
        List<Integer> nums = List.of(1, 2, 3, 4, 1, 2, 3, 5, 6, 7, 8, 4, 5, 6, 9, 1);
        List<Integer> distinctNums = nums.stream().distinct().collect(Collectors.toList());
        System.out.println(distinctNums);
    }
}
```

```
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. Create a list of employees with name and salary fields. Write a comparator that sorts the employees by salary. Then use this comparator to sort your list using the sort stream operation.

```
public class Employee {
    private String empName;
    private int salary;

    public Employee(String empName, int salary) {
        this.empName = empName;
        this.salary = salary;
    }
    public String getEmpName() {
        return empName;
    }
    public int getSalary() {
        return salary;
    }
    @Override
    public String toString() {
        return getEmpName() + "- $" + getSalary();
    }
}
```

```
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class Challengi2 {
    public static void main(String[] args) {
        List<Employee> list = List.of(new Employee("Hemanth", 500),
                new Employee("Alexander", 1000),
                new Employee("Kurup", 700));
        Comparator<Employee> sortedBySalary = Comparator.comparingInt(Employee::getSalary);
        List<Employee> sortedBySalaryList = list.stream()
                .sorted(sortedBySalary)
                .collect(Collectors.toList());
        System.out.println(sortedBySalaryList);
        // OR
        sortedBySalaryList.forEach(System.out::println);
    }
}
```

```
// // [Hemanth- $500, Kurup- $700, Alexander- $1000]
// Hemanth- $500
// Kurup- $700
// Alexander- $1000
```

OR

```
import java.util.Comparator;
import java.util.List;

public class Challengi2 {
    public static void main(String[] args) {
        List<Employee> list = List.of(new Employee("Hemanth", 500),
                new Employee("Alexander", 1000),
                new Employee("Kurup", 700));
        list.stream().sorted(new Comparator<Employee>() {
            @Override
            public int compare(Employee o1, Employee o2) {
                return Integer.compare(o1.getSalary(), o2.getSalary());
```

```
            }
        }).forEach(System.out::println);
    }
}
```

```
// Hemanth- $500
// Kurup- $700
// Alexander- $1000
```

OR

```
    list.stream().sorted((e1, e2) -> e1.getSalary() - e2.getSalary())
                .forEach(System.out::println);
```

OR

```
list.stream().sorted((e1, e2) -> Integer.compare(e1.getSalary(), e2.getSalary()))
                .forEach(System.out::println);
```

OR

```
list.stream().sorted(Comparator.comparingInt(Employee::getSalary))
                .forEach(System.out::println);
```

3. Create a list of strings representing numbers (“1”,”2”,…). convert each string to an integer, then again calculating squares of each number using the map operation and sum up the resulting integers.

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class Challengi3 {
    public static void main(String[] args) {
        List<String> numWords = IntStream.range(1, 11)
                .mapToObj(Integer::toString)
                .collect(Collectors.toList());
        List<Integer> nums = numWords.stream()
                .map(Integer::parseInt)
                .collect(Collectors.toList());
        System.out.println(nums.stream()
                .map(num -> num * num)
                .reduce(0, Integer::sum)); // 385
        nums.stream()
                .map(num -> num * num)
                .reduce(Integer::sum).ifPresent(System.out::println); // 385
    }
}
```

OR

```
List<String> numWords = IntStream.range(1, 11)
                .mapToObj(Integer::toString)
                .collect(Collectors.toList());
        numWords.stream()
                .map(Integer::parseInt)
                .map(num -> num * num)
                .reduce(Integer::sum)
                .ifPresent(System.out::println);
```

KEYPOINTS

- Functions can be assigned to variables, passed as arguments, or returned from other functions.
- Lambda expressions in Java can be used only with functional interfaces — these are interfaces that have exactly one abstract method.If an interface has more than one abstract method, it cannot be implemented using a lambda expression.
- A java stream represents a sequence of elements and supports various methods which can be pipelined to produce the desired result.
- Filter method is an intermediate operation not the terminal operation and the return type is the type of the stream.
- Functional interface in java is an interface with exactly one abstract method.
- Optional class in java is used to avoid NullPointerException.
- method references in Java are not limited to static methods. They can refer to several types of methods — including static, instance, and even constructors.
- Intermediate operations doesnot execute immediately as they are lazy, they'll be executed only if they are followed by a terminal operation.
- The max and min operations on stream returns an Optional describing the maximum or minimum element.
- The sorted operation in stream is not terminal operation but a intermediate operation and it sorts elements of the stream in their natural order.