Core Java Theory Notes

02 July 2025 16:05

Refer the Hardware concepts in notes

-->Java software belongs to oracle

Introduction to Core JAVA:

- Java was officially introduced in **1995** by **Sun Microsystems**.
  Quick Timeline:
  - **1991** – Project started by James Gosling and team (called "Oak" initially)
  - **1995** – Officially released as Java 1.0
  - **2009** – Acquired by Oracle Corporation from Sun Microsystems
- Java is Object-Oriented, simple, secured, Platform independent and robust.
- Distributed, multithreaded, High performance, Interpreted, Dynamic
- Java is Architecture Neutral, Portable

Notes:
--> Platform is Software - OS(Windows, Linux, etc)
--> JVM - Java Virtual Machine for different OS's
--> C is faster than C++ and Java because C - compiler - MLL whereas Java is compiled then bytecode and then JVM and MLL.

Why Java is Called a Dynamic Language

1. **Definition**:
   Java is called a *dynamic* language because it supports features like dynamic memory allocation, runtime polymorphism, and loading classes dynamically during program execution.

```
//Java called as DYNAMIC language because : It loads class only when we need it but not all together
-----
```

Why is Java called as Platform Independent?

- Because it can be compiled in any OS and can be executed in any other OS
- WORA -> Write once and run anywhere/everywhere

Why Is java portable? - Can be transportable to other OS
Why Java is Architecture Neutral? -Whatever the hardware is ,Java is going to run as it is neutral to all kind of architecture

---

JVM - Java Virtual Machine

- JVM is a part of JRE responsible for **running the Java bytecode**.
- When we compile .java files, they are converted into .class files (bytecode).
- JVM takes this bytecode and **converts it into machine-specific code** (OS + CPU dependent).
- It **loads classes**, **allocates memory**, **verifies bytecode**, **executes instructions**, and **manages garbage collection**.
- JVM is **platform-dependent**, but the compiled bytecode is platform-independent.
- It provides important services like:
  - **Class Loader** (loads .class files into memory),
  - **Bytecode Verifier** (ensures code doesn't violate security/access rules),
  - **Interpreter/Just-In-Time Compiler** (executes bytecode),
  - **Garbage Collector** (frees memory from unused objects).

```
Demo d = new Demo();
// JVM loads class Demo into memory (if not already loaded)
// Then allocates memory for object in heap and stores its reference in stack
```

JRE - Java runtime Environment:

- JRE is inside the RAM/share piece of memory where our java program runs/gets executed
- JRE - divided into 4 segments
  1. Code
  2. Static
  3. Heap
  4. Stack
- Note:
  - Static Segment-not used at all (**Method Area** - prefered)(or **Meta space**)(Old name: **Permanent Generation(PermGen))**

When we run a program JRE does all these below inside JVM (JRE = JVM + libraries)

Consider Instance variables and below its memory allocate

- First written code goes to code segment
- When we create a new object of a class a brand new memory is allocated for that obj in heap seg
- Then it goes to class and searches for instance variables
- And inside the heap segments brand new memory created for object, the instance vars will also be allocated with the memory and values of that variables will be never be left empty by JVM
- Hence JVM gives default values like 0 for byte and short, int and long, 0.0 float and double, false for boolean and for char its empty(empty is default value for char) and for string its null

```
Demo de = new Demo();
// Here de is reference variable for the object
```

- Here de is the reference variable of the object, and memory of ref of object is allocated in Stack seg
- Ref var - Its a variable which stores the address of the object
  Eg : In heap seg lets say the objects memory address is 1000, then the de ref var will have 1000

Consider local variables in the memory point of veiw

- Local variables memory is allocated inside the stack segment not heap
- JVM does not allocate the default values for local variables since we dont create a object for it .
- Must be initialized as they dont get default values otherwise will get error without initializing when we try to print.

INTERVIEW QUES ON JVM JRE JDK:

🔍 What is JVM?

JVM (Java Virtual Machine) is a virtual engine that runs Java bytecode and converts it to machine-specific instructions so your program can run on any platform.

🔍 What is JRE?

JRE (Java Runtime Environment) is the software package that contains everything needed to run Java programs — including JVM, core libraries, and supporting files.

🆚 Difference Between JVM and JRE

Feature	JVM (Java Virtual Machine)	JRE (Java Runtime Environment)
Role	Executes bytecode (.class)	Provides environment to run Java apps (includes JVM)

Part of JDK?	✅ Yes	✅ Yes
Platform Dep?	✅ Yes (custom JVM for each OS)	✅ Yes
Contains	Class loader, bytecode verifier, interpreter	JVM + libraries + other runtime components
Developer Use	Mainly used internally by JRE	Needed to **run** any Java application
File Type	.class loader and executor	Entire Java runtime bundle

JVM - Java Virtual Machine

- JVM is a part of JRE and is responsible for **running Java bytecode**.
- It **converts .class files (bytecode)** into machine-specific code using its interpreter or JIT compiler.
- It **allocates memory**, **loads classes**, **executes instructions**, and **cleans unused memory (GC)**.
- JVM is **platform-dependent**, but the bytecode it executes is **platform-independent**.

✅ What is JVM?

JVM (Java Virtual Machine) is a part of Java Runtime that runs compiled Java bytecode (.class files) and converts them to native code for execution on your machine.

✅ What is JRE?

JRE (Java Runtime Environment) is a package that contains everything required to **run a Java program** — including the JVM, core Java class libraries, and supporting files.

✅ Difference Between JVM and JRE

Feature	**JVM (Java Virtual Machine)**	**JRE (Java Runtime Environment)**
Purpose	Executes bytecode (runs Java programs)	Provides complete environment to run Java apps
Contains	Class loader, interpreter, JIT, garbage collector	JVM + Java class libraries + other runtime tools
Platform Dependent	✅ Yes (JVM differs for each OS)	✅ Yes (contains OS-specific JVM)
Part of	JRE & JDK	Part of JDK
Needed For	Running Java code	Running Java applications
Developer Access	Not used directly, runs behind the scenes	Installed by end-users to run Java apps
Example File	Runs .class files	Includes JVM, jars, bin, lib, etc.

📚 Interview Questions with Answers

1. What is the role of JVM in Java?

JVM executes Java bytecode and converts it to machine code specific to the host system. It handles memory allocation, class loading, and garbage collection.

2. Is JVM platform-independent?

❌ No. JVM is platform-dependent because it is implemented differently for each OS.
✅ But the bytecode it runs (.class) is platform-independent.

3. What is JRE and how is it different from JVM?

JRE is a full runtime environment including JVM + libraries to run Java apps.
JVM is just the engine that runs the bytecode. JRE gives JVM what it needs.

4. Can Java run without JRE?

❌ No. Java needs JRE to run .class files. JRE includes JVM which actually executes the code.

5. What are the memory areas used by JVM?

JVM divides memory into:

- **Code Segment**: Bytecode instructions
- **Static Segment**: For static variables
- **Heap**: For objects and instance variables

- **Stack**: For method calls and local variables

6. Where are local/instance/static variables stored in JVM memory?

Variable Type	Stored In
Local Variables	Stack
Instance Variables	Heap
Static Variables	Method/Static Area

7. What is the difference between JDK, JRE, and JVM?

Term	Description
JVM	Executes Java bytecode
JRE	Runtime environment (JVM + Libraries)
JDK	Full development kit (JRE + compiler + tools)

8. How does garbage collection work in JVM?

JVM automatically detects unused objects and reclaims their memory using Garbage Collector (GC) to improve performance and avoid memory leaks.

9. What happens when we run a Java program internally?

1. Source code is compiled by **javac** → .java → .class (bytecode)
2. JVM loads .class file using **Class Loader**
3. Bytecode is verified by **Bytecode Verifier**
4. JVM **interprets or JIT compiles** bytecode
5. Memory is managed and **GC** handles cleanup

10. Is Java 100% platform-independent?

No. Java is platform-independent at **source and bytecode level**, but **JVM is platform-dependent**.
So Java is **"Write Once, Run Anywhere"** thanks to the JVM on each platform

---

Object Orientation:

- World is a collection of objects.
- Every object belongs to a class (Class is imaginary, Object is Real)
- Every Object is having 2 parts, has(state & properties) coded using datatypes and does(behaviour) coded using methods

Class >>> JVM >>> Object

EG:

```
class Dog{
    String name;
    String color;
    Float price;
    void barks(){}
    Void eats(){}
    Void bites(){}
}

Dog browny = new Dog();
```

```
browny.barks();
browny.eats();
browny.bites();
```

Main Method :

- OS gives the control of execution/permission for programming language through main method
- Main method is a crucial component of every java program serving as entry point for execution.
- It is essential because the OS must transfer control of execution to programming lang through this method
- Only with this OS gives the permission of access.

C -

```
#include <stdio.h>
Void main(){ printf("Hello World"); } - Main method in C
```

-Same in Java:

```
public class Demo{

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

}
```

- public- Access specifier which makes - OS see main method but cant access with only public
- Static - OS can access main method without creating object i.e can access directly using class
- Void: Return type, main method return nothing
- OS can access only with both public and static keyword
- String[] args : command line argument, its an array of strings that is used to pass arguments to execute before the main method

--> In notepad:

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

-Command prompt with address of that folder and
-Command : javac name.java
-creates demo.class file in the folder which is in bytecode
-To tell java to run the demo.class
-command: java demo - This gives output in the cmd prompt

-String... Or String[] are both same

Loops:

1. For loop
2. While loop
3. Do while loop
4. For-each loop

1.for loop:

- For loop is used when number of iterations is known

```
for (initialization; condition; update) {
   // loop body
}
```

- Initialization: Runs once before the loop starts (e.g., int i = 0).
- Condition: Checked before each loop iteration. If false, loop ends.
- Update: Runs after every iteration (e.g., i++).

2.while loop

- While loop is used when number of iterations is unknown and condition is checked before the loop

```
initialization;
while (condition) {
   // loop body
   update;
}
```

- Initialization: Done before the loop (e.g., int i = 0).
- Condition: Checked before each loop iteration. If false, loop ends.
- Update: Done inside the loop (e.g., i++).

3.do while loop

- Do-while loop is used when loop must execute at least once

```
initialization;
do {
   // loop body
   update;
} while (condition);
```

- Initialization: Done before the loop starts.
- Loop Body: Executes once before checking the condition.
- Condition: Checked after each loop iteration.
- Update: Done inside the loop.

4.for each loop:

- For-each loop is used to iterate over arrays or collections

```
for (dataType element : collection) {
   // loop body
}
```

- Initialization/Condition/Update: Not explicitly written.
- Iteration: Automatically goes through each element.
- Use case: Best for reading elements, not modifying index.

DATA TYPES: Its a converter that converts real world data into binary i.e 0's & 1's
-> data is converted into binary using base-2 format if its +ve (MSB will be 0)
And base-2 format + 2's complement if data is -ve (MSB will be 1)

There are 8 data types:
==> 1 **byte** = 8 bits
==> **short** = **2 bytes** = 16 bits
==> **Int = 4 bytes** = 32 bits
==>**long = 8 bytes** = 64 bits
==> **float = 4 bytes** = 32 bits
==> double = **8 bytes** = 64 bits
==> **char** = **2 bytes** = 16 bits
==> **boolean = 1 byte** = 8 bits

Note: To find the range of the data type the formula as given below:
N bits = (-2 power n-1) **to** (+2 power n-1)+1

1. **Byte:** 8 bits

byte a;
-> Here a is a **identifier** of that byte
-> **Range : -128 to +127**
-> byte can hold only when assigned with range of -128 to 127 only

---

2. **Short:** 2 bytes or 16 bits

short a;
-> Range :- -32768 to +32767
-> Can only hold when a is assigned with values that is within the above range

---

3. **Int:** 4 bytes or 32 bits

int a;
-->Range: -2147483648 to +2147483647
-->The range is around 200 crore bytes

---

4. **Long:** 8 bytes or 64 bits

long a;
-->Range: -2 power 63 to +2 power 63 -1
-->This range is around lakh of crores

---

5. **Float : 4** bytes or 32 bits

float a = (float)45.5; or 45.5f; -> 2nd one prefered
--> `If suffix f is not added, by default java will consider it as a double value`
-->Range: -3.4 x 10po38 to +3.4 x 10po38 (no formula)
--> 3.4e308
-->Real numbers with less no after decimal or less precision values
--> Hence called Single Precision - 65.23

---

6. **Double: 8** bytes or 64 bits

double a;

-->Range: -1.7x10po308 to +1.7x10po308 (real numbers dont have formula to find range)
--> 1.7e308
-->Real numbers with more number after decimal or more precision values
--> Hence called Double precision - 65.2345678

NOTE : Real number literal : 45.5 - Double not float
so to treat make java treat as float will use (float)45.5 or 45.5;

--> There are **128 symbols**/characters according to **america**
--> If no of symbols is 128 - that is 2 power 7 - then 7 is the length of code for each character.
--> i.e 0000000 to 1111111(128th char)
--> To find binary value of a char they have given **ASCII** value for each char, for eg A- 65 ASCII value. We divide by 2 -base 2 format to get the binary value which will be stored in RAM.

--> But since there are different languages not only english **IEEE** added other languages which gave us the total of **65536 symbols** which is 2 power 16 , hence length of the code is 16
00000000 00000000 to 11111111 11111111 - This table is **UNICODE table** with several lang symbols
--> Now java follows UNICODE char table

Note : ASCII - American Standard Code for Information Interchange.
Refer ASCII table if needed.

7. **Char :** 2 bytes or 16 bits is the size of char in java **(in C char's size is 1 byte since it follows ASCII table whereas Java follows UNICODE table hence char in java is of 2 bytes)**

- 16 bits Unicode character
-->char a = 'A';
-> Character type data represents characters such as letters, digits, and symbols.

8. **Boolean** : bytes or bits

boolean a = true; (1) -1 bit takes up entire 1 byte space
boolean b = false; (0) - 1 bit takes up entire 1 byte space
-->Here 0 /1 doesnt take only space of a bit but entire 1 byte by filling other bits as 0 - 0000 000**0/1**
Since for microprocessor playing with a byte then a bit is more speed efficient.

Note :

1. Decimal Integer : 0 1 2 3 4 5 6 7 8 9
2. Octal Integer : 0 1 2 3 4 5 6 7
3. Hexadecimal Integer : 0 1 2 3 4 5 6 7 8 9 A B C D E F
4. Binary : 0 1

Prefix and it Meanings:

1. No-prefix : Decimal
2. 0 : Octal
3. 0x : Hexa decimal
4. 0b : Binary

Note : Assignment is right to left i.e lets say b=a; Right to left - assigning a to b not b to a

---

45 - is not just integer its a Decimal Integer Literal

Interview ques: OCTAL TO DECIMAL

```
//what will be the output of the below
            int a =045;
            System.out.println(a); //37
```

--> Output is 37 because 045 is Octal not decimal as java thinks if there is 0 then its octal not decimal
--> but java displays output that is decimal because it converts octal 045 to decimal
>To convert octal to decimal -convert nos octal to binary format ignoring 0, then convert that to deci

- 045 (ignore 0) - 45
- 4 - 100 (divide by 2 till we get 1 and write the reminder)
- 5 - 101
- Attach both nos binary format - 100101 - Assign each 2^5, 2^4, 2^3, 2^2, 2^1, 2^0 respectively
- And those has on add those - 2^5+2^2+2^0=32+4+1= 37

--> OR another way to convert octal to decimal:

```
             int a = 045; // Octal literal (base 8), 045 = 4×8 + 5 = 37 in decimal
            System.out.println(a); // Prints 37 (Octal 045 converted to decimal)
```

Interview ques : HEXA DECIMAL to DECIMAL

```
            int c = 0x45; // Hexadecimal literal (base 16), 0x45 = (4×16) + 5 = 69 in decimal
            System.out.println(c); // Prints 69 (Java auto-converts hex to decimal)
```

Or : 4- 0100, 5- 0101, - 01000101 - assign 2 power and add all - 69

Interview Ques : Binary to decimal - 0b -means binary

```
            int d = 0b00000010; // Binary literal (base 2), 0b00000010 = 2 in decimal
            System.out.println(d); // Prints 2 (Java auto-converts binary to decimal)
```

Or : 1 is at 2^1 - 2- prints 2

Interview Ques : What will be the output?

```
                int e = 078; // ❌ Error: 078 is invalid because 8 is not allowed in octal (base 8
    uses digits 0–7 only)
                System.out.println(e); // Compilation error (invalid octal literal)
```

TYPE CASTING : (Type conversion) -> Converting from one datatype to another

- Implicit type casting
- Explicit type casting

1. **Implicit Typecasting**(also called **widening**)**:** Lower to higher size datatype conversion
   Implicit - internal type casting done by java only

- Byte > short > int > long > float > double

char - /^ (char > int)

- Long to float - 8 to 4 is also possible because it uses different format
  // Long (64-bit integer) to float (32-bit floating-point) is allowed
  // Because float uses a different representation (IEEE 754) and can approximate large integers

```
byte a = 45;
double b;
b = a; //right to left assignment always, assigning a to b,, not b to a;
System.out.println(a); //45
System.out.println(b); //45.0 -> since its double
```

- Char to int implicit conversion:

```
char d = 'A';          // 'A' has ASCII/Unicode value 65
int e;
e = d;                 // Implicit widening from char to int
System.out.println(d); // Prints A (char)
System.out.println(e); // Prints 65 (int value of 'A')
```

Char A - unicode value 65 will be stored as int in int type.
If its a - 97

```
char x = '0';           // '0' is a character, not the number zero
int y = x;              // Implicit widening: '0' → 48 (ASCII/Unicode value)
System.out.println(x);  // Prints 0 (the character)
System.out.println(y);  // Prints 48 (its Unicode integer value)
```

Char x has 0 as a char not int, and when we assign it to y and print y, the int unicode value will be printed, 0's unicode value is 48

Honeywell Interview Question:

```
char o = '0';
int O = o;
System.out.println(O); //48
System.out.println(o); //0
```

---

2. **Explicit Type casting**(also called **Narrowing**): Highest to lowest size datatype conversion
   Since java cannot process this we externally convert using (dattype) in right side

- Double > float > long > int > short > byte
- Int > char

```
double a = 45.5;
byte b = (byte)a;
System.out.println(a); //45.5
System.out.println(b); //45
```

CODE SNIPPETS I: Incrementation and decrementation

```
//          ++ : increment
//          -- : decrement

//          ++a, --a : signs before var : pre increment
//          a ++, a-- : signs after var : post incremenet
```

```java
//            1)
              int x = 5;           // x is initialized to 5
              int y;               // y declared
              y = ++x;             // Pre-increment: x becomes 6, then assigned to y → y = 6
              System.out.println(x); // 6 → because x was incremented before assignment
              System.out.println(y); // 6 → because y = incremented value of x

//            2)
              int j = 5;           // j is initialized to 5
              int i;               // i declared
              i = j++;             // Post-increment: i = 5 first, then j becomes 6
              System.out.println(j); // 6 → because j was incremented after assignment
              System.out.println(i); // 5 → because i got the original value of j before increment

//            3)
              int c = 5;                // c is initialized to 5
              int d;                    // d is declared
              d = ++c + ++c;            // First ++c → 6, second ++c → 7 → d = 6 + 7 = 13
              System.out.println(c);    // 7 → c incremented twice
              System.out.println(d);    // 13 → result of 6 + 7


//            4)
              int e=5;
              int f;
              f = e++ + e++; // 5+6 = 11
              System.out.println(e); //7
              System.out.println(f); //11

//            5)
              int g = 6;
              int h;
              h = ++g + g++; //7 + 7
              System.out.println(g); //8
              System.out.println(h); //14

//            6)
              int n =5;
              int m;
              m= n++ + ++n + n-- + --n + ++n + --n; //5 + 7 + 7 + 5 + 6 + 5 =35
              System.out.println(n); //5
              System.out.println(m); //35

//            7)
              int o =5;
              int p;
              p = o++;
              System.out.println(o); //o is 6
              System.out.println(p); // 5

//            8)
              int q = 5;
              q = q++;             // Post-increment: q++ returns 5, but it's reassigned to q, so q
remains 5
              System.out.println(q); // 5 → because the incremented value was not stored anywhere


              System.out.println("-----------------------------------------------------");

              int a = 5;
              int b;
               b = a++ + ++a + a++ + --a + a-- + a++ + a++;
              System.out.println(a); // 8 → final value of a
              System.out.println(b); // 46 → sum of all used values


              // Step-by-step evaluation of:
              // b = a++ + ++a + a++ + --a + a-- + a++ + a++;

              // a = 5 initially
              // a++ → use 5, then a = 6
```

```
// ++a → a becomes 7, use 7
// a++ → use 7, then a = 8
// --a → a becomes 7, use 7
// a-- → use 7, then a = 6
// a++ → use 6, then a = 7
// a++ → use 7, then a = 8

//b =   5   +  7   +  7   +  7   +  7   +  6   +  7;  // = 46
```

CODE SNIPPETS II :

Note:

1. Whenever we do arithmetic operations on integer types like byte, short, int, long Java promotes them to int by default.

So byte + byte, or short + short → result is int (even if both are byte or short).
But if one operand is long → result will be long.

1. **Truncation** means cutting off the decimal part of a number without rounding.
   E.g., 12.9 → 12, -7.6 → -7

   int g = 25;
   int h = 2;
   int i = g / h;
   System.out.println(i); // 12 → result is int (25/2 = 12.5 but decimal part is truncated)

   ✅ Explanation:
   In integer division (int / int), Java drops the decimal part, not round.
   So 25 / 2 = 12.5 becomes 12 → only the integer part is kept.

1. In case we are doing int/int and assigning it to float then it would be implicit so there will no error and no need of explicit casting as already implicit casting is done by java internally.

```
int n = 25;
int m = 2;
float o = n / m;  // int / int = 12 → assigned to float → o = 12.0
// No casting needed: Java does implicit casting from int to float (widening)
System.out.println(o);  // 12.0 → decimal but no fraction as int division happened 1st


//          1)
            byte b = 127;     // Max value byte can hold is 127
            b++;//-128              // Overflow: wraps around to -128
            b++;//-127              // Increments to -127
            System.out.println(b); // -127 → because of byte overflow

//          2)
            byte a = 1;
            byte c = 1;
            byte d = (byte)(a + c);  // a + c → promoted to int, so we cast back to byte
            System.out.println(d);   // 2 → valid output after casting otherwise error without
casting

            // Whenever we do mathematical operations on the integer datatypes byte, short , int ,
long will always give int, as above

//          3)
```

```
                int g = 25;
                int h =2;
                int i = g/h;
                System.out.println(i); //Since whenever we do mathematical operation on integer types
    returns int, even if it is 12.5 as it shoul be int it will print 12
    //          And this is called truncation. 12 → result is int (25/2 = 12.5 but decimal part is
    truncated)
    //          Truncation means cutting off the decimal part of a number without rounding.

    //          4)
                int n = 25;
                int m = 2;
                float o = n / m;  // int / int = 12 → assigned to float → o = 12.0
                // No casting needed: Java does implicit casting from int to float (widening)
                System.out.println(o);  // 12.0 → decimal but no fraction as int division happened
    first


    //          5)
                double y = 128;
                byte v = (byte)y;   // Explicit cast: 128 is out of byte range (-128 to 127), so wraps
    to -128
                System.out.println(v); // -128 → because of overflow during narrowing conversion

                //byte range: -128 to 127
                // since its our of range by 1 it will come to -128 if its 2 out of range it would
    have been -127
    //          Casting 128 to byte causes overflow:
    //          128 - 256 = -128  → wraps around

                double f=129;
                byte u = (byte)f;
                System.out.println(u); //-127  //byte out of range by 2 so it goes to -127 next
    to -128
```

CODE SNIPPET III:

Note : Java does follow BODMAS , but since we have increment and decrement java follows **OPERATOR PRECEDENCE or PRIORITY TABLE:**

Table :

1. () paranthesis
2. ++, -- Increment / decrement
3. /, *, % --> Division Multiplication, Modulus
4. +, - --> (Addition and subtraction)
5. =, +=, -=, *=, /=, %= --> Assignment

```
            int m=10, n=10;
            int res = m++ / (++n * n--) / --m; // --> 10 /(11 * 11) / 10 = 10/121 * 1/10 = 1/121
            System.out.println(res); //0 ==> 10 / 121 = 0 (int division, decimal truncated), ==> 0 / 10
= 0
```

Here first solve whats inside bracket -> increment and decrement -> then multiply
Then m++ and --m then division
Then first 10/121 then the result of that 0/10 - 0

```
//          1)
```

```
            int m=10, n=10;
            int res = m++ / (++n * n--) / --m; // --> 10 /(11 * 11) / 10 = 10/121 * 1/10 = 1/121
            System.out.println(res); //0 ==> 10 / 121 = 0 (int division, decimal truncated), ==> 0 / 10
= 0


//          2)
            int x=001, y=010, z=100; // x= 0x8+1=1, y=1x8+0=8, z=100
            int i = --x + y++ - z-- - --z + ++y - --x + y-- - --x; // 0 + 8 - 100 - 98 + 10 + 1 + 10 +
2
            System.out.println(x); //-2
            System.out.println(y); //9
            System.out.println(z); //98
            System.out.println(i); // -167

//          3)
            int ch = 'A'; // A unicode value 65 is assigned to int as integer , char > int : implicit
//          System.out.println(ch++); //65
            System.out.println(++ch); //66
            System.out.println(ch); //66

//          4)
            int c = 'A', C = 'a'; //c is 65 and C is 97
            System.out.println(c++ + ++C); // 65 + 98 = 163

//          5)
            byte j = 5;
//          j = j*10; // will give error since int cant be put into byte - needs explicit casting
            j = (byte)(j*10);
            System.out.println(j); //50

//          6)
            double a = 150.0;
            byte b = (byte)a; //needs explicit casting to work
            // 150 converted into byte is out of range by 23, so from 128 we should subtract23-1 =22
i.e 128-22 = 106, answer is -106
            System.out.println(a); //150.0
            System.out.println(b); //-106
```

VARIABLES:

✅ What is a Variable in Java?

A **variable** is a name given to a memory location that holds a value which can change during program execution.

🧠 Simple Definition:

A **variable** stores data that your program can use and modify while it runs.

🧱 Syntax:

dataType variableName = value;

✅ Example:

int age = 25; // 'age' is a variable of type int
String name = "Ram"; // 'name' is a variable of type String

🔍 Key Points:

- Variables must be **declared before use**.
- Their **type defines** what kind of data they store (int, double, char, etc.).
- They can **change values** during program execution.

📦 Types of Variables in Java:

Type	Scope

Local Variable	Declared inside methods or blocks ,those vars inside main method are also local variables.
Instance Variable	Declared in a class, outside methods (non-static)
Static Variable	Declared with static, shared among all objects

Difference between all these 3 vars, consider first 2 mainly

Feature / Type	Instance Variable	Local Variable	Static Variable
Declared inside	Inside a class, but outside methods (non-static)	Inside a method, constructor, or block	Inside a class, with the static keyword
Memory location	Heap memory	Stack memory	Method area (shared memory for class)
Lifetime	Exists as long as the object exists	Exists only during method/block execution	Exists until program ends or class is unloaded
Accessed using	Via object reference	Directly inside method	Via class name or object
Shared between objects	❌ No (each object has its own copy)	❌ No (separate for each method call)	✅ Yes (one shared copy for all objects)
Default value	✅ Gets default value (e.g., 0, null, false)	❌ Must be initialized manually	✅ Gets default value
Use case	Store data for each object instance	Temporary variables inside methods	Store constants or shared values for all objects

---

PASS BY VALUE AND PASS BY REFERNECE:

---

1. Pass by Value:

- A copy of the actual value is passed to the function; changes do not affect the original variable.
- Java is **always pass by value** — but for objects, the *value of the reference* is passed. So object contents can be modified, but the reference itself cannot be changed.

```
//Pass By value:
int a = 1000; //Also called value type assignment
int b;
b = a; //Here assigning the value of a to b is called VALUE TYPE ASSIGNMENT.
System.out.println(a); //1000
System.out.println(b); //1000

b=2000; //This is also called value type assignment
System.out.println(a); //1000
System.out.println(b); //2000
```

2. Pass by reference:

- A reference (memory address) is passed; changes affect the original variable.
- Class is imaginary, object is real, object of a class is built by JVM
  Car a = new Car();
- JVM creates a object for class Car and assigns the objects address to reference variable a
- Reference variable is a variable which stores/holds the address of value of that object
- And this ref var points towards object always since it holds the address of the object

```
class Car{ //class is blueprint of the object which has state and properties
	//Instance variables declared in class instantiated through object
	String name;
	int noOfSeats;
```

```
        float cost;
}

public class PassbyValue_PassByReference {

        public static void main(String[] args) {


                // Pass by reference:
                Car a = new Car();  // JVM creates an object of class Car and assigns its address to
reference variable 'a'
                // Object is an instance of a class

                // To assign values to the instance variables, we go to the object via its reference
variable 'a'
                a.name = "Suzuki";
                a.noOfSeats = 4;
                a.cost = 43000f;

                System.out.println(a.name);  // Suzuki

                // We can create more than one reference for the same object
                Car d; // To make 'd' refer to the same object as 'a', assign 'a' to 'd'
                // 'a' holds the object's address, so now 'd' will point to the same object
                d = a;  // This is called pass by reference or reference type assignment (assigning address
to another reference)

                d.name = "KIA";           // Changes the 'name' of the object both 'a' and 'd' refer to
                System.out.println(d.name); // KIA → updated value printed
                System.out.println(a.name); //KIA //Because both a and d point to the same object!
    }
}
```

METHODS:

- A method is a block of code that performs a specific task.
- It is defined inside a class and can be called (invoked) to perform operations.
- Methods help in code reuse, structure, and readability.

```
Return_type methodName(parameters){
      Body
}
```

Difference between Method and Function:

- Function: A block of code that performs a task and returns a value (used in general programming languages).
- Method: A function that is defined inside a class (specific to OOP languages like Java).
- So, in Java, we say "method" because everything must be inside a class.

Note :

Types of Methods in Java:

1. **Instance Method**
   - Belongs to an object of the class
   - Needs object to be called
   - Can access instance and static variables
   - Example: `obj.show();`

2. **Static Method**
   - Belongs to the class, not object

- Can be called using class name
- Can only access static variables
- Example: `ClassName.methodName();`

3. **Parameterized Method**
   - Takes parameters to work with input values
   - Example: `void sum(int a, int b) { ... }`

4. **Method with Return Type**
   - Returns a value to the caller
   - Must use `return` keyword
   - Example: `int getAge() { return age; }`

5. **Constructor (special method)**
   - Same name as class
   - Used to initialize objects
   - No return type

✅ Based on Return Type and Parameters:

Return Type	Parameters	Type Description	Example
No	No	Method without return & parameters	void greet() { ... }
No	Yes	Takes input but doesn't return anything	void greet(String name) { ... }
Yes	No	Returns a value, takes no input	int getCount() { return 5; }
Yes	Yes	Takes input and returns value	int add(int a, int b) { return a + b; }

- **Based on structure/behavior**: instance, static, final, abstract
- **Based on return type and parameters**: with/without return, with/without parameters

---

WITHOUT RETURN AND WITHOUT PARAMETERS:

```
class Calculator{

	// without return and without paramter:
	int a = 50;
	int b = 40;

	void add(){
		int c= a+b;
		System.out.println(c);
	}
}
public class Methods {

	public static void main(String[] args) {

		//Methods: with /without return and paramaters.

		//without return and without paramter:
		Calculator calci = new Calculator();
		calci.add(); // 90
	}
}
```

Note : **JVM executes our java program by calling the main method**, and whenever main method is called everytime **stack frame of main()**is created in **stack segment**.
--> Then object is created in the heap segment which has certain memory address which is stored in reference variable.
--> And **every time the method** in that class is **called** using ref var **stack frame of method_name()**

here add() is created in **stack segment**
--> Then the c local var inside the method add() is stored inside stack frame of add() inside stack segment.
--> c will have 90 and once execution of calci.add() is over the stack frame of add() in stack eg disappears and c var in it will also disappear.

- So once the stack frame of add() disappears, the stack frame of main method also disappears and calc ref var in it also disappears.
- Now the Object of Calculator has no any reference var pointing to it, such object is called as GARBAGE OBJECT.
- **Those objects that have no reference is called as Garbage Objects.**
- **And this garbage object is collected by Garbage collector** which is a thread .
- After all this code segment will also have nothing.
- And **this feature of java is called as robust** which means **java uses memory efficiently** by deleting all in segments once the program execution is over.

Why memory segments have such names:

1. Code Segment (a.k.a. Text Segment)
• Stores compiled bytecode instructions of methods and constructors.
• Name "code" because it holds the actual program code to be executed.

2. Stack Segment
• Stores local variables, method call info, and reference variables.
• Name "stack" because it works on LIFO (Last In, First Out) or FILO structure.
• Each method call creates a new "stack frame" and is removed when the method ends.

3. Heap Segment
• Stores all objects and instance variables created using 'new' keyword.
• Name "heap" because memory here is dynamically allocated and freed in any order (unlike stack).
• Managed by Java's Garbage Collector.

4. Static Segment (part of Method Area)
• Stores static variables and class-level data (loaded once per class).
• Name "static" because variables here belong to the class, not instance.

Note:
• JVM memory areas: Method Area, Heap, Stack, PC Registers, Native Method Stack.
• In simple terms, JVM internally uses: Code, Stack, Heap, and Static to manage execution.

WITH RETURN AND WITHOUT PARAMETERS:

```
class Calculator2{

	//With return and without parameters
	int a=5, b=4;
	int add (){
		return a+b;
	}
}

public class WithReturnNoParameters {

	public static void main(String[] args) {

		//With return and without parameters
		Calculator2 calci = new Calculator2();
		calci.add();
		System.out.println(calci.add());
	}
```

```
}
```

WITHOUT RETURN AND WITH PARAMETRS:

```
class Calculator3{

	int c;
	void add(int a, int b) {
		c= a+b;
		System.out.println(c);
	}
}

public class WithParametersNoReturn {

	public static void main(String[] args) {

		Calculator3 calci = new Calculator3();
		int num1 = 2;
		int num2 = 3;
		calci.add(num1, num2);
	}

}
```

WITH RETURN AND WITH PARAMETERS:

```
	int c;
	int add(int a, int b) { // this int a and int b is parameters
		c= a+b;
		return c;
	}
}

public class WithReturnWithParameter {

	public static void main(String[] args) {

		Calculator4 calci = new Calculator4();
		System.out.println(calci.add(10, 20)); //here 10, 20 are arguments for the parmaters.
	}

}
```

---

2 types of classes - Programmer defined and inbuilt classes.

INBUILT CLASSES FROM JAVA:

- Till now we have seen programmer built classes.
- James gosling and his other team members have created other around 5000 classes like Scanner, Arrays, StringBuffer, Thread which are called as **inbuilt classes.**
- To see those we can just write the name of class and ctrl+space > Open implementation.
- We use the classes as same as our classes by creating object.

Scanner class to input from user:

• Scanner is a class in java.util package used to take input from the user.
• It reads input from various sources: keyboard (System.in), files, etc.

Common Methods in Scanner class:
• nextInt() – reads int
• nextLine() – reads full line
• next() – reads a word (until space)
• nextFloat(), nextDouble(), nextBoolean(), nextByte(), nextShort(), nextLong().

Usage:

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
int age = sc.nextInt();          // Read an int
String name = sc.nextLine();     // Read a full line
```

How Scanner Works Behind the Scenes:
• Scanner wraps around an Input Stream (like System.in).
• System.in represents the standard input stream (keyboard), provided by the JRE.
• When the user types input and presses Enter:
→ InputStream captures it → Scanner parses the data (tokenizes and converts)

JVM/JRE Relation:
• JVM executes the bytecode instructions using memory (stack, heap).
• JRE provides runtime environment (System.in, Scanner class, etc.)
• Scanner uses classes from JRE libraries to perform input operations.
• JVM allocates memory for Scanner object in heap, and its reference in stack.

Important Notes:
• Always import java.util.Scanner
• Always close Scanner using sc.close() (though not mandatory for System.in)
• Using nextLine() after nextInt() may require an extra sc.nextLine() to consume leftover newline

---

📘 Scanner Input Methods in Java

• Scanner is used to read user input from console (System.in)

Import:

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

Create Scanner:

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

🔢 Numeric Inputs:

```
int a = sc.nextInt();        // reads int
float b = sc.nextFloat();    // reads float
double c = sc.nextDouble();  // reads double
long d = sc.nextLong();      // reads long
short e = sc.nextShort();    // reads short
byte f = sc.nextByte();      // reads byte
```

🔤 String / Char Inputs:

String line = sc.nextLine(); // reads entire line (including spaces)
String word = sc.next();   // reads one word (until space)

char ch = sc.next().charAt(0); // reads a single character (first char of word)

✅ Boolean Input:

---

boolean b = sc.nextBoolean(); // reads true or false

🛑 Extra Tip:

• When mixing nextInt() and nextLine(), consume leftover newline:

int x = sc.nextInt();
sc.nextLine(); // consume \n before reading full line
String line = sc.nextLine();

ARRAYS: Arrays are the objects created in Heap segment

- When there is more data storing it in traditional approach is difficult hence we use Array approach.
- Gives us Dimensionality, can store homogeneous data , regular data
- Can declare int in diff types : int[] a, int a[] int []a; int[] a[];
- 2D : int[][] a; int a[][]; int [][]a; int[] a[]; int [] []a;
- 3D : int[][][] a; int[] a[][]; int a[][][]; int[][] a[]; int[] []a[]; int []a[][]; int [][][]a;
- Invalid declaration: []int a; [][]int a; []int []a; []int a[]; []int[][] a; (i.e bracket should not start at the beginning)

```
/*
 * Arrays in JAVA:
If we want to store a multiple value of the same data type we use array
 *
 * How to declare the array: Array creation
 * <datatype> [] array_name= new <datatype>[size];
 * int [] array1 = new int[4];
 * Right part called Array instantiation / Object creation, Left part - Array
declaration
 * ======================
 *
 * How to declare and initialize the array:
 * int [] arr = {1,2,3,4,5};
 *========================
 *
 * TYPES:
 *
 * 1. 1D -> single row.
 * int [] array1 = new int[4];
 *
 * 2. 2D -> in the form of table.
 * int [][] array1 = new int[4][3]; --> 4*3-->12
 * */
```

- We dont create object in the traditional way for Array as it is to be protected for security purpose we create in this way new int[5];
- The JVM then goes to array class and creates object with no of partitions as given by the programmer as the size inside [] in heap memory and its address stores in the reference var in stack memory.

```
//a.length - length of the array
Scanner sc = new Scanner(System.in);
System.out.println("Enter the size of an array:");
int size = sc.nextInt();
int [] array = new int[size];

System.out.println("Enter the values for array:");
for(int i=0; i<array.length; i++) {
	System.out.println("Enter" +i+ "th element:");
	array[i] = sc.nextInt();
}

System.out.println("Array elements are: ");
for(int j=0; j<array.length; j++) {
	System.out.println(array[j]+" ");
}
```

```
System.out.println("Enter the size of the character array:");
int size1 = sc.nextInt();
sc.nextLine();
char[] Array = new char[size1];

System.out.println("Enter the characters for the array:");
for (int i = 0; i < Array.length; i++) {
//      System.out.print("Enter character " + (i + 1) + ": ");
    Array[i] = sc.next().charAt(0);
}

System.out.println("Character array elements are:");
for (char ch : Array) {
    System.out.print(ch + " ");
}

sc.close();
```

2D - ARRAY:

- 2 dimension
- Homogeneous Data
- Regular Data

```
int[][] a = new int [2][5]; // 2 rows - each row has 5 columns
a[1][0] = 23; // accessing 2nd rows' first column
```

Eg Program: Take input from user and display:

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

System.out.println("Enter the number of rows: ");
int row = sc.nextInt(); //Taking input from user for no of rows

System.out.println("Enter the number of cols: ");
int cols = sc.nextInt();
int [][] a = new int[row][cols];
```

```
		// To give the input to array:
		System.out.println("Enter the array elements now: ");
		for (int i=0; i<row; i++) {
			for(int j=0; j<cols; j++) {
				System.out.println("Enter the ["+i+"]["+j+"] th element of the array: ");
				a[i][j] = sc.nextInt();
			}
		}

		// To print the elements of array:
		System.out.println("Array elements are: ");
		for (int i=0; i<row; i++) {
			for(int j=0; j<cols; j++) {
				System.out.print(a[i][j]+" ");
			}
			System.out.println();
		}
```

--> a.length = rows , a[i].length = cols

3D ARRAY :

- 3 dimension
- Homogeneous Data
- Regular Data

```
		//3D Array:

		//For eg : consider the following situation:
		// Create an array to store the ages of students belonging to 2 schools having 3 classrooms with 5 students each.
		int [][][] a = new int[2][3][5];
		// 2 blocks - has 3 rows with each row having 5 cols.

		a[1][0][2] = 21; //To access the 2nd block 1st row and 3rd col
		a[0][0][2] = 21; //To access the 1st block 1st row's 2rd col
```

Eg: Take input from user and display the array elements.

```
		// Create an array to store the ages of students belonging to 2 schools having 3 classrooms with 5 students each.
		Scanner sc = new Scanner(System.in);

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

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

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

		int [][][] a = new int[block][rows][cols];

		System.out.println();

		//Take input of students from block 0 and class 0
		System.out.println("Enter the elements of array: ");
		System.out.println();

		for(int i=0; i<block; i++) {
			for(int j=0; j<rows; j++) {
				for(int k=0; k<cols; k++) {
					System.out.println("Enter the Student " +(k+1)+"'s age from class "+(j+1)+" of school " + (i+1)+": ");
					a[i][j][k] = sc.nextInt();
```

```
                }
            }
        }

        //Print the elements of array
        System.out.println("The elements of array are: ");
        for(int i=0; i<block; i++) {
            for(int j=0; j<rows; j++) {
                for(int k=0; k<cols; k++) {
                    System.out.print(a[i][j][k]+" ");
                }
                System.out.println();
            }
        }
```

--> a.length = block - i , a[i].length = rows - j, a[i][j].length - cols - k

---

JAGGED ARRAY:

- Used to store irregular (non-rectangular) data in 2D form
- Homogeneous
- 2 dimension/ could be 3d
- Jagged data - Irregular data

```
// 2D Jagged array
//          int[][] a = new int[2][5];
            // Example: 2 classrooms → one has 3 students, another has 5
            // So we take 2 rows (classrooms), and columns = max students = 5

            // ⚠️ But this approach is not memory optimal
            // → First row only needs 3, but we're allocating 5 columns for both
            // → 2 unused int slots = 2 * 4 bytes = 8 bytes wasted

            int[][] a = new int[2][];
            a[0] = new int[3]; // first class has 3 students
            a[1] = new int[5]; // second class has 5 students
```

--> Here the array is created and its object address is assigned to the ref var a.
-->that object will be divided into 2 rows - a[0],a[1]
>And for this each row a new object is created. Then that objects address is stored inside the object a[0] and a[1] which is inside the main object which was created for array and thats address was given to the ref var a.

//3D Jagged array

```
            //Jagged array for 2 schools, 1 with 3 class rooms and the other with 2 classrooms.
            //1st school - 1st class - 2 students
            //1st school - 2nd class - 3 students
            //1st school - 3rd class - 3 students

            //2nd school - 1st class - 2 students
            //2nd school - 2nd class - 3 students

            int s[][][] = new int[2][][]; //2schools

            s[0] = new int [3][]; //1st school has 3 classes
            s[1] = new int [2][]; //2nd school has 2 classes

            s[0][0] = new int[2]; //1st school, 1st class
            s[0][1] = new int[3]; //1st school, 2nd class
            s[0][2] = new int[3]; //1st school, 3rd class

            s[1][0] = new int[2]; //2nd school, 1st class
            s[2][1] = new int[3]; //2nd school, 2nd class
```

2D array example for jagged array:

```
//Create an array to store the ages of students belonging to 2 classrooms where first classroom has 3
students
              //and second classroom has 5 students.
                Scanner sc = new Scanner(System.in);

                // Ask for number of classrooms
                System.out.print("Enter number of classrooms: ");
                int rows = sc.nextInt();

                int[][] arr = new int[rows][]; // jagged array

                // For each classroom, ask number of students and initialize sub-array
                for (int i = 0; i < rows; i++) {
                    System.out.print("Enter number of students in Classroom " + (i + 1) + ": ");
                    int cols = sc.nextInt();
                    arr[i] = new int[cols];

                    // Input student ages
                    System.out.println("Enter ages for Classroom " + (i + 1) + ":");
                    for (int j = 0; j < cols; j++) {
                        System.out.print("Student " + (j + 1) + ": ");
                        arr[i][j] = sc.nextInt();
                    }
                }

                // Print the array
                System.out.println("\nAges of Students:");
                for (int i = 0; i < arr.length; i++) {
                    System.out.print("Classroom " + (i + 1) + ": ");
                    for (int j = 0; j < arr[i].length; j++) {
                        System.out.print(arr[i][j] + " ");
                    }
                    System.out.println();
                }

                sc.close();
```

DRAWBACKS OF ARRAY:

- Arrays can only store homogeneous data
- Array cannot grow or shrink in size once declared - Fixed size.
- Array requires Contiguous memory allocation not dispersed.
- Hence to overcome we use collection framework and ArrayList or Linked List in that.

```
              /** Advantages of array:
               * 1. Random access.
               * 2. type safety - the elements inside the array is based on one datatype.
               * 3. type casting - To print the element in arrays - no need to type cast.
               * */
```

Different type of declaration and instantiation of an array:

```
              //Different type of declaration and instantiation for 1D
              int [] a = new int[5];
              //or
              //When we already know the data use this and below method, otherwise we use the same as
above
              int [] b =new int[] {1,2,3,4,5};
              //or
              int [] c = {1,2,3,4,5};

              //-----------------------------------------------------------------
```

```
		//Different type of declaration and instantiation for 2D
		int [][] d = new int[3][4];
		//or
		//When we already know the data use this and below method, otherwise we use the same as
above
		int [][] e ={{1,2,3,4}, {5,6,7,8}, {1,2,3,4}};

		//---------------------------------------------------------------

		//Different type of declaration and instantiation for 2D Jagged Array
		int [][] f = new int[3][];
		d[0] = new int[4];
		d[1] = new int[6];
		d[2] = new int[3];
		//or
		//When we already know the data use this and below method, otherwise we use the same as
above
		int [][] g ={{1,2,3,4}, {5,6,7,8,9,10}, {1,2,3}};

		//---------------------------------------------------------------

		//Different type of declaration and instantiation for 3D Array
		//When we already know the data use this and below method, otherwise we use the same as
above
		int [][][] h ={{{1,2,3,4},{4,5,6,7}},{{1,2,3,4},{5,6,7,8}}};
```

OBJECT IN ARRAYS:

- Can array store objects ?

```
package arrays;

class Dog{
	String breed;
	int price;
}

public class ObjectInArrays {

	public static void main(String[] args) {

		Dog maxy = new Dog();
		Dog kia = new Dog();
		Dog simba = new Dog();

		//Now we have to store these dog objects in Array?
		//Can we store object in array? Yes

		Dog [] dogs = new Dog[3];
		dogs[0] = maxy; //now maxy objects address will be refered or stored in dogs[0] , now
dog[0] has address of maxy object
		dogs[1] = kia;
		dogs[2] = simba;

		kia.breed="lab";
		//or
		dogs[1].breed = "lab";
		dogs[0].breed = "gold";
		dogs[2].breed = "goldador";

		System.out.println(dogs[1].breed);
		//or
		for(int i=0; i<dogs.length; i++) {
			System.out.println(dogs[i].breed);
		}
	}

}
```

STRINGS:

```
//Strings:
/* => Strings are sequence of characters and enclose in " ".
 * => Strings are objects of inbuilt final class String in Java.
 * => Strings are immutable.
 * => There is both immutable and mutable, where immutable belongs to inbuilt String class
and mutable String belongs to class
 * */
```

🔹 What is a String?

• A String is a sequence of characters.
• In Java, strings are objects of class `java.lang.String`.
• Strings are immutable — once created, their value cannot be changed.
• String is widely used for storing text values, passwords, inputs, etc.
○ Strings are stored in heap memory

🔹 Ways to Create a Immutable String:

---

1. Using string literal (stored in String Pool):
String s1 = "Hello";

2. Using new keyword (stored in Heap):
String s2 = new String("Hello");

3.Using char Array

```
char[] c = {'J','a','v','a'};
String str = new String(c);
```

🔹 String Memory (Heap vs String Pool): (Heap memory has SCP and heap area)

• Java stores string literals i. Without new keyword in a special memory area called the **String Constant Pool (SCP)**. Duplicates are not allowed in SCP
• When we use `new String("text")`, it creates a new object in Heap area even if the same string already exists in SCP. Duplicates are allowed in Heap memory.

🔹 String Immutability:

• Strings are immutable: once created, you can't change the value.
• Any string operation like `concat()`, `replace()` creates a new object.
• Benefits of immutability:
- Thread safety
- Memory efficiency
- Hashcode caching
- Security (used in URLs, class loading)

📘 MUTABLE STRINGS IN JAVA

🔸 By default, Java `String` objects are **immutable**:
- Once created, their values cannot be changed.
- Any operation like `concat()`, `replace()`, `toUpperCase()` returns a new string.

🔸 To create **mutable strings** (i.e., modifiable in place), Java provides:
1. StringBuilder (non-synchronized, faster)

2. StringBuffer (synchronized, thread-safe)

🔹 1. StringBuilder (Recommended for single-thread)

📌 Declaration:

```java
StringBuilder sb = new StringBuilder("Hello");
```

📌 Common Methods:

```
sb.append(" World");         // Adds to end → Hello World
sb.insert(5, " Java");       // Inserts at index 5 → Hello Java World
sb.replace(6, 10, "Python");  // Replaces chars from 6–9 → Hello Python World
sb.delete(6, 13);            // Deletes chars from 6–12 → Hello World
sb.reverse();               // Reverses the string → dlroW olleH
```

📌 Example:

```
StringBuilder sb = new StringBuilder("Code");
sb.append("WithMe");
System.out.println(sb); // Output: CodeWithMe
```

📌 Capacity vs Length:

```
System.out.println(sb.length());   // No. of characters
System.out.println(sb.capacity()); // Default: 16 + current length
```

🔹 2. StringBuffer (Same as StringBuilder but thread-safe)

📌 Use when you are working in multi-threaded environment.

📌 Example:

```
StringBuffer sbf = new StringBuffer("Learn");
sbf.append("Java");
System.out.println(sbf); // Output: LearnJava
```

📌 All methods: .append(), .insert(), .replace(), .delete(), .reverse() etc. are available here too.

```
//Mutable String in Java:
              //=> we create mutable string by creating object of class StringBuffer, StringBuilder

              //StringBuffer
              StringBuffer s = new StringBuffer();//creates new obj in heap area which has (Has part,
Does part), and this is in has which has 16 locations
              System.out.println(s.capacity()); //16 since it has 16 locations.

              System.out.println(s.append("JAVA")); //or new StringBuffer("JAVA");
              s.append(" JAVASCRIPT ");
              s.append("JAMESGOSLING");
              System.out.println(s);
              System.out.println(s.capacity());// increased to 34 how? may be 16 *2+2. dont know actual
formula and how even this formaula works

              //deleting the chars inbetween
              s.delete(1, 10); //exclusive 10
              System.out.println(s);

              //--------------------------------------------

              //StringBuilder
              StringBuilder sa = new StringBuilder();
              System.out.println(sa.capacity()); //16
              sa.append("JAVA ");
              sa.append("JAVASCRIPT ");
```

```
                sa.append("JAMES GOSLING");
                System.out.println(sa);
                System.out.println(sa.capacity()); //34

                sa.delete(1, 10);
                System.out.println(sa);

                //difference : StringBuffer methods are syncronized, StringBuilder is not syncronized

                //Reverse a string:
                String sv = "Varshini";
                System.out.println(sv);

                //reverse method is in string buffer
                //so we need to covert string to stringBuffer first and pass the string to StringBuffer obj
                StringBuffer st = new StringBuffer(sv);
                System.out.println(st.reverse());

//              to Use lowerCase() on stringBuffer we should convert it into String then use loweCase()
                StringBuffer svt = new StringBuffer("JAVA");
//              svt.toLowerCase(); //will give error
                String s1 = new String(svt); //now we can convert to lowercase
                System.out.println(s1.toLowerCase());

                //or another method is using toString()
                StringBuffer s2 = new StringBuffer("JAVA");
                String s3 = s2.toString();
                System.out.println(s3.toLowerCase());

                //Whatever string user give as input by keyboard is immutable
                //Cant take i/p for StringBuffer and builder.

                Scanner sc= new Scanner(System.in);
                System.out.print("Enter the string: ");
                String s6 = sc.nextLine();
                System.out.println(s6);

                //To print the o/p in red color we use System.err(Used to display errors) instead of
System.out but should not use it to print o/p
                System.err.println(s6);
```

🔸 Differences: String vs StringBuilder vs StringBuffer

Feature		String	StringBuilder	StringBuffer
Mutability		❌ Immutable	✅ Mutable	✅ Mutable
Thread-Safe		✅ (immutable)	❌ Not thread-safe	✅ Thread-safe
Performance		🔻 Slower	✅ Fastest	🔻 Slower
Memory Usage		Creates new	Modifies in-place	Modifies in-place

---

🧠 INTERVIEW Qs & NOTES

❓ Why is String immutable in Java?
✅ Because it improves security (e.g., in class loading), thread-safety, and caching performance.

❓ When to use StringBuilder?
✅ When you need frequent string modifications in single-threaded apps.

❓ What is the default capacity of StringBuilder?
✅ 16 characters. It grows dynamically when needed.

❓ Can StringBuilder be converted to String?
✅ Yes. Use .toString() method.

❓ Will StringBuilder allocate new memory every time you append?
✅ No. It manages its own dynamic buffer.

📌 CONVERT BETWEEN TYPES

✔ String → StringBuilder:
StringBuilder sb = new StringBuilder("Hello");

✔ StringBuilder → String:
String str = sb.toString();

🔹 Common String Methods:

---

length() → returns string length
charAt(index) → returns character at index
substring(start, end)→ returns part of the string
concat(str) → joins strings
equals(str) → checks content equality
equalsIgnoreCase(str)→ case-insensitive equality
compareTo(str) → compares lexicographically
contains(str) → checks if substring exists
startsWith(str) → true if prefix matches
endsWith(str) → true if suffix matches
toUpperCase() → converts to upper case
toLowerCase() → converts to lower case
trim() → removes leading/trailing spaces
replace(old, new) → replaces characters
split(delim) → splits string into array

EXAMPLE:

```
// Creating a new String object with full name
String s = new String("ISHWARCHANDRAVIDYASAGAR");

// ------------------------ String methods ------------------------

// Converts all characters to lowercase
System.out.println(s.toLowerCase()); // ishwarchandravidyasagar

// Converts all characters to uppercase (already in uppercase)
System.out.println(s.toUpperCase()); // ISHWARCHANDRAVIDYASAGAR

// Returns the number of characters in the string
System.out.println(s.length()); // 24

// Returns the character at index 6 (0-based indexing)
System.out.println(s.charAt(6)); // A

// Returns ASCII value of character at index 5
System.out.println(s.codePointAt(5)); // 82 (ASCII of 'R')

// Finds the first occurrence index of 'R'
System.out.println(s.indexOf('R')); // 5

// Finds the last occurrence index of 'R'
System.out.println(s.lastIndexOf('R')); // 22

// Checks if the string contains the sequence "PARVATHI"
System.out.println(s.contains("PARVATHI")); // false

// Checks if the string contains the sequence "SAGA"
System.out.println(s.contains("SAGA")); // true
```

```java
        // Checks if the string is empty (length == 0)
        System.out.println(s.isEmpty()); // false

        // Checks if the string is blank (all characters are whitespace)
        System.out.println(s.isBlank()); // false

        // Returns substring from index 13 to the end
        System.out.println(s.substring(13)); // VIDYASAGAR

        // Returns substring from index 13 to 18 (18 exclusive)
        System.out.println(s.substring(13, 18)); // VIDYA

        //_____________________________________________________________________________

        // String Splitting Example
        String s2 = new String("ISHWAR,CHANDRA,VIDYA,SAGAR");

        // Directly printing split() returns an array's address (not readable)
        System.out.println(s2.split(",")); // Not useful - prints memory address

        //  Use traditional for loop to print each element after splitting by comma
        String[] arr = s2.split(",");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]); // prints each word: ISHWAR, CHANDRA, ...
        }

        // Use enhanced/modern for loop (also called "for-each" loop)
        for (String word : arr) {
            System.out.println(word);
            // Useful when: you only want to read values, not change them
            // Limitations: cannot go backwards, cannot get index directly
        }

        //_____________________________________________________________________________
        // Converting String to Character Array
        // Reason: Working with arrays is easier for individual character manipulation
        char[] array = s.toCharArray(); // Creates char array with size equal to the length of the
string
        for (char a : array) {
            System.out.print(a); // Prints each character in the string one by one
        }

        System.out.println(); // Just for spacing after above output

        // _____________________________________________________________________________

        // Taking a String input from user using Scanner
        Scanner sc = new Scanner(System.in);

        // Takes input until space is encountered (i.e., only first word)
        String string = sc.next();
        System.out.println(string); // Prints the input string

        // Note:
        // If you want to take the full line as input (including spaces), use:
        // String string = sc.nextLine();
```

🔹 String Comparison:

---

1. `equals()` → compares value
2. `==` → compares reference (address)
3. `compareTo()` → strings are compared character by character.

returns:

- 0 if equal
- +ve if s1 > s2
- -ve if s1 < s2

4.equalsIgnoreCase() → Strings are compared based by ignoring case.

EXAMPLES:

Equals(), equalsIgnoreCase(), ==

```
		String s1 = "JAVA";
		String s2 = "JAVA";

		System.out.println(s1==s2); //compares address/reference //true since scp doesnt allow duplicates, s1 and s2 will have same address and points to same obj that has JAVA
		System.out.println(s1.equals(s2)); //compares values //since values are same it is true

		//---------------------------------------------------------------

		String s3 = "JAVA"; //stored in SCP and its ref is given to s3
		String s4 = new String("JAVA"); //stored in heap area and its ref is given to s4

		System.out.println(s3==s4); //compares address/reference //false since s3 has address of obj created in SCP , and s4 has address of obj in heap area. both 2 diff objects with diff address
		System.out.println(s3.equals(s4)); //compares values //since values are same it is true

		//---------------------------------------------------------------------

		String s5 = "JAVA";
		String s6 = "jAVA";
		System.out.println(s5.equals(s6)); //false //because java is case sensitive
		System.out.println(s5.equalsIgnoreCase(s6)); //true //ignores case

		//---------------------------------------------------------------------

		String s7 = new String("JAVA");
		String s8 = new String("JAVA");
		System.out.println(s7==s8); //false beacuse s7 and s8 creates 3 seperate new objects, since new is used it goes to heap area, and heap area allows duplicates, so 2 new objs are created
		System.out.println(s7.equals(s8)); //true

		//---------------------------------------------------------------------

		String s9 = "JAVA";
		String s10 = "java";
		System.out.println(s9==s10); //false // java case sensitive and JAVA , java will create 2 diff objects in SCP since they are not same, so when 2 obj ref is compared its diff hence false
		System.out.println(s9.equals(s10)); //false //case sensitive hence not equal
```

CompareTo()

```
		//Comparing strings using compareTo()--continuation of StringsIntro Class refer it:

		String s1 = "SACHIN";
		String s2 = "SAURAV";

		System.out.println(s1.compareTo(s2)); //-18
//Will comapre char by char -  S and S- true, A and A - true, C and U - false: subtract ASCII values of both=> 67-85= -18
//s1<s2 : -ve value will be returned (based on ASCII values : 67<85, Hence -ve value)
//Hence we understand based on the value - -ve : s1<s2, +ve : s1>s2, 0 : s1=s2

		//So if these are made vice versa will get +ve
		s1 = "SAURAV"; //// we are not modifying the string, just changing the reference variable to a new object
		s2 = "SACHIN";
		System.out.println(s1.compareTo(s2)); //18 //+ve //s2<s1

		String s3 = "SACHIN";
		String s4 = "SACHIN";
		int m = s3.compareTo(s4);
		System.out.println(m);// 0 //since they both are equal

		if(m>0) {
```

```
            System.out.println("string 1 is greater than string 2");
        }else if(m<0) {
            System.out.println("String 2 is greater than string 1");
        }else {

        }

        // Case 1: Unequal length, compared char-by-char, no difference found, so compare by length
        String s5 = "JAVA";     // After matching 'A', there are 0 remaining chars
        String s6 = "JAVAC";    // After matching 'A', there is 1 more char 'C'
        System.out.println(s5.compareTo(s6)); // -1 → 0 - 1 (s5 is shorter) → s5 < s6

        // Case 2: Reversed from above
        String s7 = "JAVAC";    // After matching 'A', 1 more char
        String s8 = "JAVA";     // After matching 'A', 0 more char
        System.out.println(s7.compareTo(s8)); // 1 → 1 - 0 → s7 > s8

        // Case 3: More difference in length
        String sa = "JAVAJAVA"; // Same up to "JAVA", then has 4 extra chars
        String sb = "JAVA";     // Ends after "JAVA"
        System.out.println(sa.compareTo(sb)); // 4 → sa has 4 extra chars → sa > sb

        // Case 4: concat() doesn't modify original string unless reassigned
        String sc = "JAVA";
        System.out.println(sc); // JAVA
        sc.concat("Programming"); // Creates new String "JAVAPROGRAMMING" but not assigned
        System.out.println(sc); // JAVA → original string unchanged (immutable)
```

Q: If strings are immutable, how can we reassign them like s1 = "NewValue"?

📜 **Answer**:

Strings are immutable, so their content cannot be modified.
But string **reference variables** like s1 can point to different string objects.
When we do s1 = "NewValue", we are **not modifying** the original object,
we're just **changing the reference** to a new object.

🔹 What is String Concatenation?

• String concatenation is the operation of joining two or more strings together to form a single combined string.
• In Java, this can be done using:
  1. `+` operator
  2. `concat()` method
  3. `StringBuilder` or `StringBuffer`

🔹 1. Using `+` Operator (Operator Overloading)

• Java overloads the `+` operator for strings.
• If any operand is a string, Java converts the others to string and joins them.

Example:

```
String name = "Java" + "Program";
System.out.println(name); // JavaProgram

int a = 10;
String b = "Hello";
System.out.println(a + b); // 10Hello
```

• Internally, + uses StringBuilder for efficiency in recent Java versions.

EXAMPLE:

```java
//String concatenation:
            //==> Adding to 2 Strings which results in new String, String is immutable
            String s1 = "JAVA";
            String s2 = "PYTHON";
            String s3 = "JAVA"+"PYTHON";
            String s4 = "JAVA"+"PYTHON";

            System.out.println(s3==s4);//true JAVAPYTHON in scp is already there, so s3 and s4 will
have same address.
            System.out.println();

            s3 = s1+s2;
            s4 = s1+s2;
            System.out.println(s3==s4);
            //false because when we concatenate  " "+" "  with values/that object will be created on
SCP and will have same address, since they are equal
            // but when we concatenate using reference s1+s2 then brand new obj is created in heap
area, so s3 and s4 will have diff addresses.
            System.out.println();

            s3 = s1+"PYTHON";
            s4 = s1+"PYTHON";
            System.out.println(s3==s4);
            //false because when we concatenate  ref+" " or " "+ref with ref and value , new obj will
be created in heap area., s3 and s4 will have diff obj address, hence false.
            System.out.println();
```

🔹 2. Using concat() Method

• The concat() method joins two strings.
• It only works with strings — not numbers or characters.

Example:
String a = "Hello";
String b = "World";
System.out.println(a.concat(b)); // HelloWorld

❗ Note: concat() does not modify original string — strings are immutable.

```java
            //Concatenation using concat()
            String s1 = "JAVA";
            String s2 = "PYTHON" ;

            //Whenever we use concat() new obj is created inside heap area in heap segment not in
StringsConstantPool
            String s3= "JAVA".concat("PYTHON");
            //Whenever we use .concat() the brand new object is created in Heap Segment's Heap area and
address is stored in s3 which is inside Stack segment.
            String s4= "JAVA".concat("PYTHON"); //same goes with this new obj is created in heap area,
and s4 stores the address of it.

            System.out.println(s3==s4); //false, since ref vars have diff address
            System.out.println(s3.equals(s4)); //true //since values are same

            s3 = s1.concat("PYTHON");
            s4 = s1.concat("PYTHON");
            System.out.println(s3==s4); //false, since ref vars have diff address (using concat hence
new obj for both in heap area)
            System.out.println(s3.equals(s4)); //true //since values are same

            s3 = s1.concat(s2);
            s4 = s1.concat(s2);
            System.out.println(s3==s4); //false, since ref vars have diff address (using concat hence
new obj for both in heap area)
            System.out.println(s3.equals(s4)); //true //since values are same
```

🔹 3. Using StringBuilder (Best for Loops)

• Best choice for repeated concatenation (like in loops).
• Mutable → doesn't create new objects for each concat.

Example:
StringBuilder sb = new StringBuilder("Hello");
sb.append("World");
System.out.println(sb); // HelloWorld

🔹 4. Using StringBuffer

• Same as StringBuilder, but synchronized (thread-safe).

Example:
StringBuffer sb = new StringBuffer("Java");
sb.append("Rocks");
System.out.println(sb); // JavaRocks

🔹 Concatenation with Null

String s = null;
System.out.println(s + "Test"); // nullTest
System.out.println("Test" + null); // Testnull

🔹 Performance Comparison:

Method	Thread Safe	Mutable	Best Use
+ Operator	No	No	1-2 strings only
concat()	No	No	Only strings
StringBuilder	No	Yes	Repeated concat
StringBuffer	Yes	Yes	Multi-threaded apps

STRING TOKENIZER CLASS - INBUILT:

➤ What is StringTokenizer?

StringTokenizer is a class in `java.util` package used to break a string into tokens.

➤ Why use it?

It is helpful when you want to break a long string (like comma-separated or space-separated values) into multiple smaller strings (tokens), especially for parsing.

➤ Syntax:
---
import java.util.StringTokenizer;

StringTokenizer st = new StringTokenizer("your_string", "delimiter");

➤ Constructor Variants:
---
1. StringTokenizer(String str)
   - Delimiter: default is whitespace (space, tab, newline)
2. StringTokenizer(String str, String delimiter)
   - Custom delimiter like ",", "-", etc.
3. StringTokenizer(String str, String delimiter, boolean returnDelims)
   - If `true`, delimiters are also returned as tokens

➤ Commonly Used Methods:

---
1. boolean hasMoreTokens()
  - Returns true if there are more tokens left to be read

2. String nextToken()
  - Returns the next token

3. int countTokens()
  - Returns the number of tokens remaining

EXAMPLES:
=========

Example 1: Split a sentence using default delimiter (space)

```
import java.util.StringTokenizer;

public class Main {
   public static void main(String[] args) {
      StringTokenizer st = new StringTokenizer("Java is easy");

      while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
      }
   }
}
```

Output:

```
Java
is
easy
```

Example 2: Split a string using comma as delimiter

```
StringTokenizer st = new StringTokenizer("Apple,Mango,Banana", ",");

while (st.hasMoreTokens()) {
   System.out.println(st.nextToken());
}
```

Output:

```
Apple
Mango
Banana
```

Example 3: Count tokens

```
StringTokenizer st = new StringTokenizer("One Two Three");

System.out.println("Total tokens: " + st.countTokens());
```

Output:

```
Total tokens: 3
```

Example 4: Return delimiter as token

```
StringTokenizer st = new StringTokenizer("A,B,C", ",", true);

while (st.hasMoreTokens()) {
   System.out.println(st.nextToken());
}

Output:
-------
A
,
B
,
C
--------------------------------------------------------------------------------
```

```java
        // ---------------------- StringTokenizer ----------------------
        // It is a class in `java.util` package used to split a string into tokens.
        // Useful for parsing data separated by spaces, commas, or other delimiters.
        // Each token is like a "word" split from the original string using the delimiter.

        // ---------------------- Example 1 ----------------------
        StringTokenizer st = new StringTokenizer("JAVA AI ML", " "); // Delimiter: " " (space) // Will split the string wherever space occurs
        while(st.hasMoreTokens()) { // Checks if there is another token available
            System.out.println(st.nextToken());
        }

        // ---------------------- Example 2 ----------------------
        StringTokenizer s1 = new StringTokenizer("JAVA,AI,ML", ","); // Delimiter: "," (comma) // Will split the string wherever comma occurs
        while(s1.hasMoreTokens()) {
            System.out.println(s1.nextToken());
        }

        // ---------------------- Example 3 ----------------------
        StringTokenizer s2 = new StringTokenizer("COMMMMMMMMMUNICATION", "M");
        // Delimiter: "M"
        // Every 'M' is treated as a separator. It doesn't care how many times it appears.
        while(s2.hasMoreTokens()) {
            System.out.println(s2.nextToken());
            // Output:
            // CO
            //
            //UNICATION
            // (Empty tokens are between multiple M's)
        }

        // ---------------------- Interview Point ----------------------
        // Q: Is StringTokenizer faster than split()?
        // A: No. `String.split()` is fast and takes less memory so it is preferred in modern Java (it supports regex too).
        //    StringTokenizer is a legacy class from before Java 1.4.

        // Q: When would you still use StringTokenizer?
        // A: If you need basic, fast tokenizing without regex overhead – especially in older Java codebases.

        // ---------------------- Bonus Method ----------------------
        // You can also use countTokens() to know how many tokens exist:
        StringTokenizer s3 = new StringTokenizer("A B C", " ");
        System.out.println("Total Tokens: " + s3.countTokens()); // Output: 3

        // ---------------------- Comparison Summary ----------------------
        // ➤ StringTokenizer
        //    - Legacy (older)
        //    - No regex support
        //    - Slower than split()
        //    - Still useful for simple token parsing
        //
```

```
        // ➤ split()
        //    - Newer, recommended
        //    - Uses regex
        //    - More powerful and flexible
```

INTERVIEW-FOCUSED NOTES:
==========

🟩 Q1: When would you use StringTokenizer over split()?

• When performance matters (faster for simple tokenization)
• When you want to tokenize using single character delimiters only
• Less memory overhead than split()

🟩 Q2: Difference between split() and StringTokenizer?

Feature	StringTokenizer	split()
Package	java.util	Defined in String class
Delimiter type	Only character	Supports regex
Returns	Tokens one by one	Returns String[] array
Performance	Slightly faster	Can handle complex cases
Is Deprecated?	Not deprecated but less used	Preferred in modern Java

Split() is faster then stringTokenizer, tokenizer also takes more memory, so st() is best.

🟩 Q3: Is StringTokenizer still recommended?

• For simple token splitting: YES
• But in modern Java (>= 1.5), prefer using `String.split()` or `Scanner` for better flexibility

🟩 Q4: Can StringTokenizer return the delimiter as a token?

Yes, use the constructor:
→ `StringTokenizer(str, delim, true);`
It will return delimiters also.

🟩 Q5: Can you give a one-liner to count tokens in a sentence using StringTokenizer?

```
new StringTokenizer("This is Java").countTokens();  // returns 3
```

LIMITATIONS OF STRINGTOKENIZER:
==========

1. Only works with single character delimiters
2. Doesn’t support regular expressions (unlike split())
3. Doesn’t store tokens - can only iterate once
4. Slightly outdated; not used much in modern code

ALTERNATIVE: Using split() method (recommended in modern Java)

```
String str = "Red,Green,Blue";
String[] colors = str.split(",");

for(String color : colors) {
   System.out.println(color);
}
```

💡 When to use what?
==========

• Use StringTokenizer for legacy systems or simple space/comma-separated strings
• Use split() when dealing with regex or when you need more flexibility
• Use Scanner when taking token input from user or file

REAL-TIME USE CASES:
==========

• Parsing CSV-like data
• Breaking command line arguments
• Tokenizing user input
• Parsing logs

IMPORTANT POINTS TO REMEMBER:
==========

• Tokens are returned one-by-one
• It ignores consecutive delimiters (unlike split())
• Doesn’t store tokens; only provides sequential access
• Works well for light-weight tokenization needs

_____

📌 INTERVIEW THEORY QUESTIONS + ANSWERS
_____

Q1: Is + operator overloaded in Java?
A: Yes, for strings only. It's internally converted to StringBuilder.

Q2: What is the difference between + and concat()?
A:
• + can combine strings with any type (int, char, etc.).
• concat() only works with strings.

Q3: Which is better for concatenation in a loop?
A: StringBuilder → avoids multiple object creation.

Q4: Is concatenation using + efficient?
A: No, in loops it creates multiple objects → use StringBuilder instead.

Q5: Does concat() change the original string?
A: No. It returns a new string. Strings are immutable.

Q6: What will happen if I do null + "text"?
A: Treated as string literal → gives output: "nulltext"

_____

🧠 MCQ QUESTIONS:
_____

Q1:
String s = "abc";
s.concat("def");
System.out.println(s);
A) abcdef

B) abc ✅
C) def
D) Compilation Error

Q2:
String s = "a" + "b" + "c";
System.out.println(s);
A) abc ✅
B) a b c
C) a+b+c
D) Compilation Error

Q3:
Which of the following is thread-safe?
A) String
B) StringBuilder
C) StringBuffer ✅
D) concat()

Q4:
String str = null;
System.out.println(str + "Test");
A) nullTest ✅
B) Error
C) NullPointerException
D) Testnull

🔍 INTERVIEW CODE SNIPPETS:

🔸 Concatenating using a loop (bad way):

```
String result = "";
for (int i = 0; i < 5; i++) {
   result += i; // creates new object each time
}
System.out.println(result); // 01234
```

🔸 Efficient version using StringBuilder:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
   sb.append(i);
}
System.out.println(sb); // 01234
```

🔸 Concatenating int, float and string:

```
int x = 10;
float y = 5.5f;
String s = "Value";
System.out.println(s + x + y); // Value105.5
```

🔹 Mutable String Alternatives:

---

1. `StringBuilder` → Fast, not thread-safe
2. `StringBuffer` → Slower, thread-safe

String vs StringBuilder:

Feature	String	StringBuilder
Mutable	❌ No	✅ Yes
Thread Safe	❌ No	❌ No

| Performance | Slow | Fast |

🔹 Why Strings Are Immutable in Java?

• For caching and SCP reuse
• For thread safety
• For use in security-critical operations like file paths, class loading
• To store hashcode and improve performance

🔹 Example Code Snippets:

```java
String s = "Java";
System.out.println(s.length());          // 4
System.out.println(s.charAt(0));         // J
System.out.println(s.toUpperCase());     // JAVA
System.out.println(s.substring(1,3));    // av
System.out.println(s.contains("av"));    // true
```

📌 INTERVIEW THEORY QUESTIONS WITH ANSWERS:

Q1: What is the difference between == and equals() in Strings?
A: == compares references (memory address), equals() compares actual content.

Q2: Why are Strings immutable in Java?
A: To ensure thread-safety, allow safe sharing, and optimize performance via caching.

Q3: What is the String Constant Pool?
A: A special memory area where Java stores string literals to reuse and save memory.

Q4: Difference between String, StringBuffer and StringBuilder?
A:

- String → Immutable
- StringBuffer → Mutable, Thread-safe (slow)
- StringBuilder → Mutable, Not Thread-safe (fast)

Q5: What happens when we do: s1 = s1 + "new"?
A: It creates a new string object with the new value. The old object remains unchanged.

Q6: Can two string literals with same content have different references?
A: No. If both are string literals (not using new), they point to the same reference in SCP.

Q7: What is intern() method?
A: It adds the string to the SCP and returns the canonical representation.

🧠 MCQ-STYLE QUESTIONS:

Q1: What is the output?

```
String s1 = "Java";
String s2 = new String("Java");
System.out.println(s1 == s2);
```

Answer: false (Different memory references)

Q2: What is the output?

```
String str = "Hello";
str.concat("World");
System.out.println(str);
```

Answer: Hello (concat creates new string, original unchanged)

Q3: What does this return?

```
"hello".charAt(1);
```

Answer: 'e'

Q4: Which method is NOT in the String class?
A) length()
B) append() ✅
C) equals()
D) substring()

Q5: What is the output?

String s = "abc";
String t = "abc";
System.out.println(s == t);
Answer: true (both literals point to same SCP memory)

🎯 SHORT CODING SNIPPETS ASKED IN INTERVIEWS:

🔸 Reverse a String:

```
String input = "hello";
String reversed = new StringBuilder(input).reverse().toString();
```

🔸 Check Palindrome:

```
String s = "madam";
System.out.println(s.equals(new StringBuilder(s).reverse().toString())); // true
```

🔸 Count Vowels in a String:

```
String s = "Developer";
int count = 0;
for (char ch : s.toLowerCase().toCharArray()) {
  if ("aeiou".indexOf(ch) != -1) count++;
}
System.out.println(count);
```

🔸 Remove All Spaces:

```
String s = " a b c ";
System.out.println(s.replace(" ", "")); // abc
```

🔸 Split by Space and Print Words:

```
String s = "Java is fun";
String[] words = s.split(" ");
for (String word : words) System.out.println(word);
```

🔸 Compare Strings Without Case:

```
String s1 = "Java";
String s2 = "java";
System.out.println(s1.equalsIgnoreCase(s2)); // true
```

CONVENTION IN JAVA:

CONVENTIONS IN JAVA:
Java has a set of standard naming conventions to make code clean, consistent, and readable.

1. Class Names – PascalCase Convention

- Class names should begin with a capital letter and follow PascalCase (each word starts with uppercase).

```
class StudentDetails {
}
class BankAccountManager {
}
```

2. Method Names – camelCase Convention

- Method names should start with a lowercase letter, and follow camelCase (next words start with uppercase).

```
void calculateSum() { }
int getStudentId() { return id; }
```

3. Variable Names – camelCase Convention

- Start with a lowercase letter, and use camelCase for multiple words.

int totalMarks;
String firstName;

4. Constant Names – UPPERCASE snake convention with underscores

- Constants (static final variables) should be in uppercase with words separated by _.

static final int MAX_LIMIT = 100;
static final String DB_URL = "localhost";

5. Package Names – lowercase only

- All package names should be in lowercase to avoid conflicts.

package com.corejava.basics;
package studentportal;

6. Interface Names – PascalCase Convention

- Same as class names; usually nouns or adjectives.

interface Drawable { }
interface Printable { }

7. Enum Names – PascalCase and Constants in UPPERCASE

- Enum type names follow class naming conventions.
- Enum constants use all UPPERCASE letters.

enum Days { MONDAY, TUESDAY, WEDNESDAY }

8. Boolean Getters – isXxx() Naming Convention

- If a method returns boolean, prefer isXxx() over getXxx().

boolean isEmpty() { return true; }

WHY FOLLOW CONVENTIONS?

- Makes code predictable and easy to read
- Improves team collaboration
- Avoids naming conflicts
- Ensures consistency across large projects

COMMAND LINE ARGUMENTS:

METHOD OVERLOADING IN JAVA
==========

Method Overloading is one of the key concepts of **Polymorphism** in Java (compile-time polymorphism).
It allows multiple methods in the same class to have the **same name** but **different parameters** (in type, number, or order).

WHAT IS METHOD OVERLOADING?

- Method Overloading means **defining multiple methods with the same name** but with **different parameter lists**.
- This allows calling the same method in different ways, depending on what arguments are passed.

WHY METHOD OVERLOADING?

- Improves **code readability**.
- Avoids creating multiple method names for similar actions.
- Provides **flexibility** for method calls with different inputs.

Compile-Time Polymorphism means the method to be called is **determined by the compiler** at the time of **compilation**, **not during runtime**.

🔍 In Case of Method Overloading:

When you overload methods (i.e., same method name, different parameters), the **compiler knows exactly which version of the method to call** based on the **number and type of arguments** passed.

→ Since this decision is made **statically** (without executing the code), it is called:

- Static Binding : because method overloading is handled by JVM since its static when converting bytecode to binary
- Early Binding : Since JVM compiles first hence early
- Compile-time Polymorphism

RULES OF METHOD OVERLOADING

---

1. The method name **must be the same**.
2. The parameter list **must be different**:
   - By number of parameters
   - By data type of parameters
   - By order of parameters (if types differ)
3. Return type **does NOT matter** in overloading (only parameters matter).
4. Access modifiers can be same or different – they don't affect overloading.

EXAMPLES

```
// Example 1: Overloading by number of parameters
public class Demo {
    void show() {
        System.out.println("No arguments");
    }

    void show(int a) {
        System.out.println("One argument: " + a);
    }

    void show(int a, int b) {
        System.out.println("Two arguments: " + a + ", " + b);
    }
}

// Example 2: Overloading by type of parameters
class Calculate {
    int sum(int a, int b) {
        return a + b;
    }

    double sum(double a, double b) {
        return a + b;
    }
}

// Example 3: Overloading by order of parameters
class Printer {
    void print(String s, int n) {
        System.out.println("String and int: " + s + ", " + n);
    }

    void print(int n, String s) {
        System.out.println("Int and string: " + n + ", " + s);
    }
}
```

INVALID OVERLOADING EXAMPLE (Only return type differs)

```
class Example {
   int show(int a) {
      return a;
   }

   // This will cause compile-time error
   double show(int a) {
      return a * 1.0;
   }
}
// Reason: Parameter list is same; return type doesn't help in overloading.
```

INTERVIEW-ORIENTED QUESTIONS (WITH ANSWERS)

Q1: What is method overloading in Java?
A1: It is a feature that allows multiple methods with the same name but different parameter lists in the same class.

Q2: Can we overload methods by changing the return type only?
A2: No. The compiler does not consider return type for method overloading.

Q3: Is method overloading a part of polymorphism?
A3: Yes. It is an example of compile-time (static) polymorphism.

Q4: Can overloaded methods have different access modifiers?
A4: Yes. Overloading is based only on parameter lists. Access modifiers can be different.

Q5: Can overloaded methods throw different exceptions?
A5: Yes. The exception list does not affect method overloading.

KEY DIFFERENCE: OVERLOADING VS OVERRIDING

Feature	Overloading	Overriding
Based on	Different parameter list	Same method name and parameters
Happens in	Same class	Parent-child (inheritance)
Return type	Can differ (but not alone)	Must be same or covariant
Polymorphism Type	Compile-time	Runtime
Access modifier	Can be different	Must be same or more visible
Static methods	Can be overloaded	Cannot be overridden

BEST PRACTICES

- Keep method names meaningful and consistent in overloaded forms.
- Don't rely on return type alone for overloading.
- Avoid overloading with too many variations — can reduce readability.
- Prefer using optional parameters or builder pattern if overloads get too many.

REAL-LIFE EXAMPLE (FOR BETTER UNDERSTANDING)

Example: Print function with different inputs

```
class Printer {
   void print(int n) {
      System.out.println("Printing number: " + n);
```

```
    }

    void print(String s) {
        System.out.println("Printing string: " + s);
    }

    void print(double d) {
        System.out.println("Printing decimal: " + d);
    }
}

Usage:
Printer p = new Printer();
p.print(5);          // Printing number: 5
p.print("Hello");     // Printing string: Hello
p.print(3.14);       // Printing decimal: 3.14

--------------------------------------------------
SUMMARY
--------------------------------------------------
✔ Same method name
✔ Different parameter lists (type, number, order)
✗ Cannot overload using return type alone
✔ Improves readability and usability
✔ Example of Compile-Time Polymorphism


class Demo{
        void add(int a, int b) {
                System.out.println(a+b);
        }

        void add(int a, int b, float c) {
                System.out.println(a+b+c);
        }

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

                Demo d = new Demo();  // Creating an object of Demo class

                d.add(1, 10);         // Calls method with two int parameters
                d.add(1, 10.5f);      // Calls method with int and float parameters
                d.add(1, 10, 10.5f);  // Calls method with int, int, and float parameters

                // At compile time, Java checks the method signatures and chooses the correct version of
the method to call.
------
class Demo {

        void add(char a, int b) {
                System.out.println(a + b); // Adds char and int → both get promoted to int → prints sum
        }

        void add(int a, char b) {
                System.out.println(a + b); // Adds int and char → both get promoted to int → prints sum
        }

        public static void main(String[] args) {
                Demo d = new Demo();
//                  d.add('a', 'b');
                // ❌ This gives **compile-time ambiguity error**
                // Because both 'a' and 'b' are chars, and Java can't decide:
```

```
            // → Should it match (char, int) by promoting 2nd char to int?
            // → Or match (int, char) by promoting 1st char to int?
            // JVM is confused between both valid options → **Ambiguous call**
        }
}
```

Can you method overload for main method?

Yes

✅ Valid Overloading of main:

```
public class MainOverload {

    // JVM looks for this exact method signature to start execution
    public static void main(String[] args) {
        System.out.println("Main with String[] args");
        //until we call other main methods as below we get whats inside our main main method i.e Main
with String[] args

        // You can call other overloaded main methods from here
        main(5);
        main("Java");
    }

    // Overloaded version 1
    public static void main(int a) {
        System.out.println("Overloaded main with int: " + a);
    }

    // Overloaded version 2
    public static void main(String s) {
        System.out.println("Overloaded main with String: " + s);
    }
}
```

🧠 Important Notes:

- ✅ You can **overload main like any other method** (change the parameter types/number).
- ❌ But **JVM only invokes**:

  java
  CopyEdit
  public static void main(String[] args)
- Other versions won't be called automatically — you must call them explicitly.

🎯 Interview Tip:

Q: Can you overload the main method in Java?
A: Yes, we can overload the main method like any other method, but the JVM only calls the main(String[] args) version during program startup.

```
// ---------------------------------------------
// 🚀 COMMON INBUILT OVERLOADED METHODS IN JAVA
// ---------------------------------------------

// 1. System.out.println() → PrintStream class
// Overloaded to print any type: int, float, char, boolean, object, etc.
System.out.println(10);          // Prints integer
System.out.println("Hello");     // Prints string

// 2. System.out.print() → PrintStream class
// Similar to println() but does not move to a new line after printing.
System.out.print(10);            // Prints integer
System.out.print("Hello");       // Prints string
```

```
// 3. substring() → String class
// Extracts part of string; single index or start–end indexes.
String s = "PROGRAM";
System.out.println(s.substring(3));    // "GRAM"
System.out.println(s.substring(2, 5)); // "OGR"

// 4. indexOf() → String class
// Finds index of character or substring; can specify start index.
System.out.println(s.indexOf('O'));      // 2
System.out.println(s.indexOf("RA", 3));   // 4

// 5. valueOf() → String class (static method)
// Converts any datatype to String (int, char, float, boolean...).
System.out.println(String.valueOf(100));   // "100"
System.out.println(String.valueOf(false)); // "false"

// 6. replace() → String class
// Replace character or substring.
System.out.println(s.replace('R', 'X'));     // "PXOGXAM"
System.out.println(s.replace("PRO", "DEV")); // "DEVEGRAM"

// 7. Math.max() → Math class
// Returns larger value; works for int, float, double, long.
System.out.println(Math.max(5, 10));     // 10
System.out.println(Math.max(2.5, 3.9));  // 3.9

// 8. Math.min() → Math class
// Returns smaller value; works for all numeric types.
System.out.println(Math.min(4, 1));      // 1
System.out.println(Math.min(7.2, 6.3));  // 6.3

// 9. Math.abs() → Math class
// Returns absolute value; works for int, float, long, double.
System.out.println(Math.abs(-9));       // 9
System.out.println(Math.abs(-5.4f));     // 5.4

// 10. Arrays.sort() → Arrays class
// Sorts array of any data type or given range.
int[] arr = {3, 2, 1};
Arrays.sort(arr);                  // Full array sort
Arrays.sort(arr, 0, 2);              // Partial sort: index 0 to 1

// 11. Arrays.fill() → Arrays class
// Fills entire array or part of it with given value.
Arrays.fill(arr, 5);                // Fills all with 5
Arrays.fill(arr, 1, 3, 9);           // Index 1 to 2 with 9

// 12. compareTo() → String class
// Compares 2 strings lexicographically.
System.out.println("A".compareTo("B"));    // -1
System.out.println("Z".compareTo("Z"));    // 0

// 13. equals() → Object class (overridden by String and others)
// Compares content of objects (not address).
System.out.println("Hi".equals("Hi"));     // true
System.out.println("Hi".equals("hello"));   // false

// 14. charAt() → String class
// Returns character at given index.
System.out.println("JAVA".charAt(2));      // V

// 15. split() → String class
// Splits string based on delimiter and returns String array.
```

```
String[] parts = "A,B,C".split(",");
System.out.println(parts[1]);            // B

// 16. String.format() → String class
// Formats values into a formatted string (like printf).
System.out.println(String.format("%s %d", "Age:", 21)); // Age: 21

// 17. printf() → PrintStream (System.out)
// Similar to C-style printf formatting.
System.out.printf("%s = %d", "Count", 10); // Count = 10

// 18. getBytes() → String class
// Converts string to byte array.
byte[] bytes = "ABC".getBytes();
System.out.println(bytes[0]);            // 65

// 19. StringBuilder.append() → StringBuilder class
// Appends any type (int, string, char, float...) to string.
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(123);
System.out.println(sb);                  // Hello123

// ---------------------------------------------
```

🔶 COMMAND LINE ARGUMENTS IN JAVA

• Command Line Arguments are parameters passed to the Java program during execution via the terminal or command prompt.
• These arguments are passed as Strings to the `main()` method.

🔹 Syntax of main() method:

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

• `args` is an array of `String` that holds the command line arguments.
• These arguments are separated by space and indexed like an array.
• You can name it anything (e.g., `String[] myArgs`) – the name doesn't matter, only the type and position matter.

🔹 Example: Simple program to print arguments

```java
public class CmdDemo {
   public static void main(String[] args) {
      System.out.println("Number of arguments: " + args.length);

      for (int i = 0; i < args.length; i++) {
         System.out.println("Argument " + i + ": " + args[i]);
      }
   }
}
```

→ To run from terminal:

```
javac CmdDemo.java
java CmdDemo.java Hello 123 World
```

→ Output:
Number of arguments: 3
Argument 0: Hello
Argument 1: 123
Argument 2: World

🔹 Notes:

• Arguments are passed as strings by default.
• You must convert them if using as numbers:
→ Integer.parseInt(args[0])
→ Float.parseFloat(args[1]) etc.

🔹 Example with type conversion:

```
public class AddTwo {
   public static void main(String[] args) {
      int a = Integer.parseInt(args[0]);
      int b = Integer.parseInt(args[1]);
      int sum = a + b;

      System.out.println("Sum: " + sum);
   }
}
```

→ Run using:

```
java AddTwo 10 20
```

→ Output:
Sum: 30

🔹 What if no argument is passed?

→ You will get ArrayIndexOutOfBoundsException if you try to access args[i] and args.length is 0.

→ Always check args.length > 0 before accessing.

🔹 Interview Questions:

Q1: Can we change the name of 'args'?
→ Yes. Only the type matters (String[]), not the variable name.

Q2: What is the data type of command-line arguments?
→ String[] (Array of Strings)

Q3: Where are command line arguments stored?
→ In the args[] array of the main method.

Q4: Can we pass multiple arguments?
→ Yes. Each space-separated value becomes one element in the args array.

Q5: How do you convert args to numbers?
→ Use wrapper class methods like Integer.parseInt(), Float.parseFloat() etc.

🔹 Advantages of Command Line Arguments:

• Allows flexibility – users can pass different inputs without modifying code.
• Used for dynamic input from terminal/shell scripts.

• Useful in automation or scripting tools.

🔹 Common Errors:

• Forgetting to parse String → int
• Accessing args[i] when args.length = 0
• Using args.length without checking bounds
• Spaces in arguments without double quotes

→ Use " " around multi-word arguments:

java Program "Hello World" 123

📝 Summary:

• Passed during program run via terminal
• Stored as String[] in main(String[] args)
• Must be parsed if using as numeric data
• args.length tells how many were passed
• Handle exceptions using try-catch or check args.length

PACKAGES IN JAVA

What is a Package?

- A package in Java is a group of related classes, interfaces, and sub-packages.
- It helps in organizing code and preventing class name conflicts.
- It also provides access protection and namespace management.

Why Use Packages?

- To group similar types of classes.
- To avoid name conflicts between classes.
- To manage files easily in large projects.
- To provide controlled access using access modifiers.

Types of Packages:

1. Built-in packages: Provided by Java (like java.util, java.io, java.sql).
2. User-defined packages: Created by developers for their own classes.

CREATING A USER-DEFINED PACKAGE

Step 1: Create a Java file with package statement at the top

```
// File: MyClass.java
package mypackage;
public class MyClass {
public void display() {
System.out.println("Hello from MyClass in mypackage");
}
}
```

Step 2: Compile the class using -d option

```
javac -d . MyClass.java
```

This creates a folder named 'mypackage' and places the class inside it.

Step 3: Use this class in another Java file

```
// File: Test.java
import mypackage.MyClass;
public class Test {
public static void main(String[] args) {
MyClass obj = new MyClass();
obj.display();
}
}
```

Step 4: Compile and Run

```
javac Test.java
java Test
```

IMPORTING PACKAGES

1. import packagename.ClassName; // imports one specific class
2. import packagename.*; // imports all classes in package
3. import static packagename.ClassName.member; // imports static members

Example:
import java.util.Scanner;
import java.util.*;

ACCESS MODIFIERS WITH PACKAGES
public - Accessible everywhere
protected - Accessible in same package or subclass in another package
default - Accessible only within same package
private - Accessible only inside that class

BUILT-IN PACKAGES
java.lang - contains core classes (String, Math, Object, etc.)
java.util - utility classes like ArrayList, Scanner, HashMap
java.io - input/output classes like File, BufferedReader
java.net - networking classes like URL, Socket
java.sql - JDBC classes for database interaction

SUBPACKAGES

- Packages inside other packages are called subpackages.
  Example:
  package com.company.project.module;
- The subpackage must be imported explicitly. Importing the parent doesn't import subpackages.

STATIC IMPORT
Used to import static members of a class without class name.
Example:
import static java.lang.Math.*;
Then you can use:
System.out.println(sqrt(16));
System.out.println(pow(2, 4));

DEFAULT PACKAGE

- If no package is specified, the class is considered in the default package.
- Default package classes can't be imported by classes in named packages.

INTERVIEW QUESTIONS WITH ANSWERS
Q1: What is a package in Java?
A: A package is a mechanism to group related classes, interfaces, and sub-packages to organize code efficiently.
Q2: What are the types of packages?
A: Built-in packages (provided by Java) and User-defined packages (created by the programmer).
Q3: How do you compile a class inside a package?
A: Use javac -d . ClassName.java to generate proper folder structure.
Q4: Can we import a class from a subpackage by importing the parent package?
A: No, we must import the subpackage explicitly.
Q5: What is the use of static import?
A: It allows accessing static members of a class without class name.
Q6: Can a class belong to multiple packages?
A: No. A class can belong to only one package.
Q7: Difference between import pack.* and import pack.ClassName?
A: First imports all classes in the package; second imports only the mentioned class.
Q8: Why is java.lang package automatically imported?
A: Because it contains core classes needed in almost every Java program like String, Object, System, etc.
Q9: Can we create nested packages?
A: Yes, and they must follow directory structure (e.g., com.company.project).
Q10: How is package different from folder?
A: Folder is at file-system level, whereas package is a namespace recognized by the JVM and compiler.

SUMMARY

- Packages help modularize code and prevent naming conflicts.

- Java supports both built-in and user-defined packages.
- Use -d to compile package-based code.
- Proper use of access modifiers and import statements ensures better code organization.

CONSTRUCTORS IN JAVA

• A **constructor** is a special method used to initialize objects of a class.
• It is **automatically called** when an object is created using the new keyword.
• The **name of the constructor must be the same as the class name**.
• A constructor **does not have a return type**, not even void.
• Constructors can be **parameterized or non-parameterized**.
• If no constructor is explicitly defined, **Java provides a default constructor** (no-arg).

SYNTAX:

```
class ClassName {
   // Constructor
   ClassName() {
      // constructor body
   }
}
```

TYPES OF CONSTRUCTORS

1. **Default Constructor (No-arg Constructor)**
    - A constructor that takes **no parameters**.
    - If no constructor is defined, Java provides one automatically.

```
java
CopyEdit
class Demo {
   Demo() {
      System.out.println("Default Constructor called");
   }
public static void main(String[] args) {
      Demo d = new Demo(); // Constructor is called here
   }
}
```

- When a local variable (like a constructor or method parameter) has the same name as a class-level variable, it hides (shadows) the class variable.
- This causes confusion and bugs, as the local variable takes priority, and the class variable is not accessed or updated as intended.
- Shadowing happens commonly in constructors or setters where parameter names match instance variables.
- To resolve this, use the this keyword to clearly refer to the class variable (e.g., this.name = name;).

---

1. **Parameterized Constructor**
    - A constructor that accepts **arguments** to initialize an object with specific values.

```
java
CopyEdit
class Student {
   String name;
   int age;
Student(String n, int a) {
      name = n;
      age = a;
   }
```

```
void display() {
      System.out.println("Name: " + name + ", Age: " + age);
   }
public static void main(String[] args) {
      Student s = new Student("Rahul", 20);
      s.display();
   }
}
```

1. **Copy Constructor**
    - Java does not have a built-in copy constructor like C++, but you can define your own.

```
java
CopyEdit
class Student {
   String name;
   int age;
Student(String n, int a) {
      name = n;
      age = a;
   }
// Copy constructor
   Student(Student s) {
      name = s.name;
      age = s.age;
   }
void display() {
      System.out.println("Name: " + name + ", Age: " + age);
   }
public static void main(String[] args) {
      Student s1 = new Student("Amit", 21);
      Student s2 = new Student(s1); // Copy constructor
      s2.display();
   }
}
```

IMPORTANT CHARACTERISTICS OF CONSTRUCTORS

- Constructors **cannot be static, final, or abstract**.
- Can be **overloaded** (same name, different parameters).
- Cannot be inherited but can be called from child class using super().
- Can call another constructor in the same class using this().

CONSTRUCTOR OVERLOADING

- Defining **multiple constructors** with different parameter lists.

```
class Car {
   String model;
   int price;
Car() {
      model = "Unknown";
      price = 0;
   }
Car(String m) {
      model = m;
      price = 0;
   }
Car(String m, int p) {
      model = m;
      price = p;
   }
void display() {
      System.out.println(model + " - " + price);
```

```
  }
public static void main(String[] args) {
      Car c1 = new Car();
      Car c2 = new Car("Honda");
      Car c3 = new Car("BMW", 50000);
      c1.display();
      c2.display();
      c3.display();
  }
}
```

CALLING CONSTRUCTOR FROM ANOTHER CONSTRUCTOR IN SAME CLASS - LOCAL CHANINg

• Use this() to call another constructor from the same class.

```
class Box {
  int length, breadth, height;
Box() {
      this(10, 10, 10); // Calls parameterized constructor
  }
Box(int l, int b, int h) {
      length = l;
      breadth = b;
      height = h;
  }
void showVolume() {
      System.out.println("Volume: " + (length * breadth * height));
  }
public static void main(String[] args) {
      Box b = new Box();
      b.showVolume();
  }
}
```

DIFFERENCE BETWEEN CONSTRUCTOR AND METHOD

Constructor	**Method**
Called automatically	Called manually
No return type	Must have return type
Same name as class	Any name
Initializes object	Performs action
No static, final, abstract	Can be static, final, etc.

WHEN TO USE CONSTRUCTOR

• When you want to **set default or initial values** while creating an object.
• When you want to **force user to provide input** while creating objects.
• Useful when creating **multiple objects with different data**.

---

local CHAINING (Local Chaining using this())

• Constructor Chaining means calling one constructor from **another constructor in the same class**.
• This is achieved using the special keyword **this()**.
• **Purpose**: To avoid code duplication and reuse initialization logic across multiple constructors.
• This is also called **Local Chaining** or **Constructor Overloading with Chaining**.

RULES of Constructor Chaining (this()):

1. this() must be the **first line** in the constructor when used.
2. If used after any statement, it will give a **compile-time error**.
3. Can only be used **within the same class**, not across classes.
4. You can chain **as many constructors** as needed using this().
5. If chaining is done, only **one object** is created — just multiple constructors are executed.

Example 1: Basic Local Constructor Chaining

```
java
CopyEdit
class Student {
   String name;
   int age;
   String course;
// Constructor 1: No-arg constructor
   Student() {
      this("Unknown", 0, "Not Assigned"); // Calls constructor 3
      System.out.println("Default constructor called");
   }
// Constructor 2: Parameterized constructor with 2 params
   Student(String name, int age) {
      this(name, age, "Java"); // Calls constructor 3
      System.out.println("2-arg constructor called");
   }
// Constructor 3: Fully parameterized
   Student(String name, int age, String course) {
      this.name = name;
      this.age = age;
      this.course = course;
      System.out.println("3-arg constructor called");
   }
void display() {
      System.out.println(name + " | " + age + " | " + course);
   }
public static void main(String[] args) {
      Student s1 = new Student(); // Triggers all three constructors via chaining
      s1.display();
      Student s2 = new Student("Rahul", 22); // Triggers 2-arg and 3-arg
      s2.display();
   }
}
```

Output:

```
sql
CopyEdit
3-arg constructor called
Default constructor called
Unknown | 0 | Not Assigned
3-arg constructor called
2-arg constructor called
Rahul | 22 | Java
```

Example 2: Error if this() is not the first line

```
java
CopyEdit
class Demo {
   Demo() {
      System.out.println("Before this()"); // Compile-time error if you use this() after this
      this(5); // Error: Constructor call must be the first statement
   }
Demo(int x) {
      System.out.println("Value: " + x);
   }
}
```

Best Practices:

- Always place this() as the **first statement** in a constructor.
- Only use this() for **chaining between overloaded constructors**, not methods.
- Use constructor chaining to simplify and reduce redundancy when initializing different sets of data.

Why use Constructor Chaining?
• Code Reusability
• Reduces Redundancy
• Makes constructor logic maintainable
• Allows better flexibility during object creation

Real-Life Analogy:
Imagine you go to a hotel and order a full thali. Instead of the waiter preparing each item individually, they use a combo pack — one method to bring everything. That's constructor chaining — reusing existing preparation logic.

```
package objectOrientedProgramming;

class Customer{
	private int cId;
	private String cName;
	private int cNum;


//Noargs constructor
public Customer() {
	cId = 1;
	cName = "Arul";
	cNum = 1234;

}

public Customer(int cId, String cName, int cNum) {
	//to call the above constructor
	this(); //points to current classes constructor(same class), by this we can call other
constructor to another constructor.
	//So this() is used for local chaining
	this.cId  = cId;
	this.cName = cName;
	this.cNum = cNum;
}

public int getcId() {
	return cId;
}

public String getcName() {
	return cName;
}

public int getcNum() {
	return cNum;
}

}

public class LocalChaining {

	public static void main(String[] args) {

		Customer c = new Customer(2, "Alex",123);
		System.out.println(c.getcId());
		System.out.println(c.getcName());
		System.out.println(c.getcNum());
	}

}
```

Constructors are typically declared as public to allow the creation of objects from anywhere in the code

---

==========

CONSTRUCTOR CHAINING

==========

> Constructor Chaining means calling one constructor from another constructor — either within the same class or from the parent class — during object creation.

> It ensures that **all constructors involved in the inheritance chain are executed** in proper order, allowing all fields (including inherited ones) to be initialized.

🔹 There are **two types** of constructor chaining in Java:

---

1. **Within the same class** — using `this()`
2. **Between parent and child class** — using `super()`

🔹 1. Constructor Chaining WITHIN SAME CLASS — using `this()`

✔ `this()` is used to call another constructor **in the same class**
✔ It must be the **first statement** inside the constructor

🔸 Example:

```
class Sample {
    Sample() {
        this(10);  // calls Sample(int)
        System.out.println("No-arg constructor");
    }

    Sample(int x) {
        System.out.println("Parameterized constructor: " + x);
    }
}
```

Output:
Parameterized constructor: 10
No-arg constructor

⛔ ERROR if `this()` is **not the first statement**

🔹 2. Constructor Chaining BETWEEN CLASSES — using `super()`

✔ `super()` is used to call the **parent class constructor**
✔ Always the **first statement** inside the subclass constructor
✔ If not written explicitly, Java **adds `super()` by default**

🔸 Example:

```
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
}

class Child extends Parent {
```

```
    Child() {
        super();  // calls Parent() — added implicitly if omitted
        System.out.println("Child constructor");
    }
}

class Main {
    public static void main(String[] args) {
        new Child();
    }
}
```

Output:
Parent constructor
Child constructor

▶️ Why super() first? Because parent's data must be initialized before child's.

🧠 JVM FLOW WHEN OBJECT IS CREATED (Memory + Execution Perspective)

---

1. Class loading happens if not already loaded (static blocks run here)
2. Memory allocated → all instance variables set to default values
3. Constructor is called →
   - `super()` triggers Parent constructor
   - Then child's constructor body runs
4. Object is fully initialized

So,
✔ Parent constructor runs before child
✔ Even if subclass has `this()`, Java ensures parent constructor is called

Example Chain:
- SubClass() → this(int) → super(int)

🔥 KEY RULES — this() vs super()

Feature	this()	super()
Purpose	Call same class constructor	Call parent class constructor
Used in	Constructor	Constructor
Position	First statement only	First statement only
Can both be used?	❌ Not together	Only one allowed per constructor
Implicitly called?	❌ No	✅ Yes (super())

✅ PRACTICAL USE CASES / ADVANTAGES OF CONSTRUCTOR CHAINING

✔ Reduces code duplication (common init logic moved to one constructor)
✔ Better readability and maintenance
✔ Ensures that all constructors participate in initialization

---

💡 REAL EXAMPLE: MULTI-LEVEL CONSTRUCTOR CHAINING

```
class A {
    A() {
        System.out.println("A()");
```

```
    }
}

class B extends A {
    B() {
        this(5); // calls B(int)
        System.out.println("B()");
    }

    B(int x) {
        super(); // calls A()
        System.out.println("B(int): " + x);
    }
}

class C extends B {
    C() {
        System.out.println("C()");
    }
}

class Main {
    public static void main(String[] args) {
        new C();
    }
}
```

Output:
A()
B(int): 5
B()
C()

🧠 Flow:
C() → B() → B(int) → A()

⛔ COMMON MISTAKES & INTERVIEW TRAPS

❌ Calling `this()` or `super()` anywhere other than **first line** → Compile Error
❌ Trying to use both `this()` and `super()` in the same constructor → Error
❌ Assuming `super()` is needed only in multi-level inheritance — No, even one level uses it
❌ Forgetting that Java implicitly inserts `super()` if not written

📌 WHEN TO USE `this()` vs `super()`

✅ Use `this()` → when you want to reuse another constructor logic from same class
✅ Use `super()` → when child constructor must initialize parent's part too
✅ Combine `this()` and `super()` in chaining chain across classes
(But not in the same constructor)

🎭 WHY JAVA MANDATES `super()` AS FIRST STATEMENT?

Because if the parent object is not constructed properly, child cannot rely on it.
Initialization order ensures that all fields and methods from parent are valid before child uses them.

```
package objectOrientedProgramming;

class Test{
    int x, y;
```

```
    //Constructors are typically declared as public to allow the creation of objects from
    anywhere in the code
    public Test() {
        //the super() from child class comes here and executes the below, here also there
        will be super() given by JVM which will go to super class constructor i.e Object
        class. then executes the constructor of Object executes whats there in it comes back
        and executes x and y below lines and goes back to child class
        //super();
        x=100;
        y=200;
    }

    public Test(int x, int y) {
        //super();
        this.x = x;
        this.y = y;
    }
}

class Test2 extends Test{

    int a,b;

    public Test2() {
        //super(); // refers to the constructor of super/parent class. If we dont give then
        the JVM will give super() automatically always
        //since no parameters its go to default constructor.

        //this and super cant be together
        this(9,99); //this will go to same class another parameterized constructor
        a= 300;
        b=400;
    }

    public Test2(int a, int b) {
        //super(a,b); //This goes to parameterized constructor of Parent class.
        super(); //this will goto default constructor of parent class //local chaining is
        happening here then constructor chaining .
        this.a = a;
        this.b = b;

    }

    void disp(){
        System.out.println(x);
        System.out.println(y);
        System.out.println(a);
        System.out.println(b);
    }
}

public class ConstructorChaining {

    public static void main(String[] args) {

        Test2 t = new Test2(8,88); //100,200, 8, 88 //There are parameters so it goes to
        parameterized constructor
        t.disp();

        Test2 ti = new Test2(); //100,200,300,400 //no parameters go to zero parameter
        constructor
        ti.disp();

        Test2 tii = new Test2(8,88); //8, 88, 8, 88 //parameterized here and also in
        constructor by giving a, b as parameters : super(a,b); hence it goes to parameterized
```

```
            constructor of parent class and assigns,x and y with a and b
            tii.disp();

            //constructor and local chaining together -
            Test2 t2 = new Test2(); //100,200,300,400
            t2.disp();

            //Programmers can give this(), this(a,b), super(), super(a,b)
            //JVM can only give super() only if programmer has not given anything
            //first line of constructor can be either this or super not both.

      }

}
```

```
package objectOrientedProgramming;

class Demo11 {
      Demo11(){
            System.out.println("Inside parent constructor");
      }
}


class Demo22 extends Demo11{

      //Constructors will never get inherited to child class but will get executed because of super()
call.
      //Because if the Demo11 comes to child class name of constructor should be same as class name
which will not happen, but however gets executed because super() in the child class will make execute
the constructor from parent class

      Demo22(){
            System.out.println("Inside child constructor");
      }
}

public class ConstructorChaining2 {

      public static void main(String[] args) {

            Demo22 d = new Demo22();

      }

}
```

- Constructors are NOT inherited in Java.
  So, Demo11() is not inherited by Demo22.

- But when an object of Demo22 is created,
  the parent constructor Demo11() is executed automatically
  because of an implicit call to super() in Demo22 constructor.

- Reason:
  Every constructor (child or parent) must ensure its
  superclass is fully constructed first → Java does this by calling super()

- Note:
  Even though Demo11's constructor is not inherited,
  it still runs when you create a Demo22 object —
  because the compiler adds `super();` as the first line of Demo22()

PLAIN OLD JAVA OBJECT (POJO)

WHAT IS A POJO?

- POJO stands for "Plain Old Java Object".
- A POJO is a simple Java object that doesn't depend on any special libraries, frameworks, or annotations.
- It's mainly used to hold data in a clean, readable format.
- POJOs follow standard Java object conventions:
    - Private variables (fields)
    - Public getters and setters
    - A no-argument constructor (optional)
    - May override toString(), equals(), hashCode() methods (optional)

WHY USE POJOs?

- To transfer data in a structured way (e.g., between layers in an application).
- Reduces complexity and coupling.
- Used widely in Spring, Hibernate, REST APIs, ORM frameworks, etc.

BASIC RULES OF POJO

1. Must be in a public class.
2. Should not extend pre-specified classes or implement interfaces (except for Marker Interfaces).
3. Fields must be private.
4. Should have public getter and setter methods.
5. Should not contain business logic or methods other than data access.

EXAMPLE: POJO CLASS

```
java
CopyEdit
// Student.java
public class Student {
private String name;
   private int age;
   private String course;
// No-arg constructor
   public Student() {
   }
// All-args constructor (optional)
   public Student(String name, int age, String course) {
      this.name = name;
      this.age = age;
      this.course = course;
   }
// Getters
   public String getName() {
      return name;
   }
public int getAge() {
      return age;
   }
public String getCourse() {
      return course;
   }
// Setters
   public void setName(String name) {
      this.name = name;
   }
public void setAge(int age) {
      this.age = age;
   }
public void setCourse(String course) {
      this.course = course;
   }
// Optional toString() method
   @Override
```

```
  public String toString() {
    return "Student [name=" + name + ", age=" + age + ", course=" + course + "]";
  }
}
```

USES OF POJO IN REAL PROJECTS

- Used in JavaBeans
- Used for Data Transfer Objects (DTO)
- Used in ORM frameworks like Hibernate to map database tables to Java classes
- Used in REST API models (for request/response mapping)

ADVANTAGES OF POJO

- Simple and clean design
- Easy to test and reuse
- No dependency on any framework
- Makes code readable and maintainable

POJO vs JAVA BEAN

Feature	POJO	Java Bean
Access	Any access modifier	Must be public
Constructors	May not have constructors	Must have no-arg constructor
Serializable	Not required	Should be Serializable
Getter/Setter	Optional	Mandatory
Extends class	Can extend other classes	Should not extend concrete classes

POSSIBLE INTERVIEW QUESTIONS

1. What is a POJO class in Java?
2. How is POJO different from JavaBeans?
3. Why are fields private in POJO?
4. Can a POJO class have business logic?
5. Can we make a POJO class immutable?

ANSWERS:

1. A POJO (Plain Old Java Object) is a simple Java class that follows standard conventions with private fields, and public getters/setters, used mainly to hold and transfer data.
2. JavaBeans are a stricter form of POJO that must be public, have a no-arg constructor, and be serializable.
3. Fields are private to achieve encapsulation.
4. No, POJO should not contain business logic, only data.
5. Yes, by making fields final, initializing them via constructor, and not providing setters.

MCQs ON POJO

Q1. What does POJO stand for?
a) Plain Object of Java
b) Plain Old Java Object
c) Pure Object Java Output
d) None of the above
Answer: b) Plain Old Java Object
Q2. Which of the following is TRUE about POJO?
a) POJO can only contain static variables
b) POJO should implement Runnable
c) POJO should not contain business logic
d) POJO must extend Object class
Answer: c) POJO should not contain business logic
Q3. Which is NOT a characteristic of POJO?
a) Has private variables
b) Must implement an interface
c) Has public getters/setters
d) Is framework-independent
Answer: b) Must implement an interface

Bean Class in Java:

```
// A JavaBean is a special type of POJO that follows certain stricter rules.

            //Bean class is a class which implements a empty interface Serializable  which gives

// Rules to be a Bean class:
// 1. Must be public.
// 2. Must have a no-argument constructor.
// 3. All fields (variables) should be private.
// 4. Must have public getter and setter methods for accessing private fields.
// 5. Should implement java.io.Serializable interface (optional, but common).

// Why Serializable?
// Serializable is a marker interface (an interface with no methods).
// It tells the JVM that the object of this class can be converted into a byte stream.
// This is useful for saving the object's state to a file, transferring it over a network, etc.

// Without Serializable, Java cannot persist or transfer the object using streams.

import java.io.Serializable;

public class EmployeeBean implements Serializable {

   private int id;
   private String name;
   private double salary;

   // No-argument constructor (required)
   public EmployeeBean() {
   }

   // Parameterized constructor (optional)
   public EmployeeBean(int id, String name, double salary) {
      this.id = id;
      this.name = name;
      this.salary = salary;
   }

   // Getter and Setter methods
   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }

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

   public double getSalary() {
      return salary;
   }

   public void setSalary(double salary) {
      this.salary = salary;
   }

   // Optional: toString method for display
   @Override
```

```
  public String toString() {
    return "EmployeeBean [id=" + id + ", name=" + name + ", salary=" + salary + "]";
  }
}
```

WHAT IS A BLOCK IN JAVA?

==========

→ A **block** in Java is a group of statements enclosed within **curly braces `{}`**.
→ It defines a **scope** — where variables can be declared and used.
→ Blocks can exist in methods, conditionals, loops, constructors, static blocks, etc.

TYPES OF BLOCKS IN JAVA

==========

1. INSTANCE BLOCK (Non-static block)

→ Runs **every time** an object is created (before the constructor).
→ Used for common initialization code.

Example:

```
class Demo {
  {
    System.out.println("Instance Block");
  }

  Demo() {
    System.out.println("Constructor");
  }
}
// Output:
// Instance Block
// Constructor
```

2. STATIC BLOCK

→ Runs **only once** when the class is first loaded into memory.
→ Used to initialize static variables or perform setup code.

Example:

```
class Test {
  static {
    System.out.println("Static Block");
  }

  public static void main(String[] args) {
    System.out.println("Main Method");
  }
}
// Output:
// Static Block
// Main Method
```

3. METHOD BLOCK

→ Regular block inside a method.
→ Used to group code together with its own scope.

Example:

```
void show() {
  {
    int x = 10;
```

```
        System.out.println(x);
    }
    // x cannot be accessed here (block scope)
}
```

4. CONSTRUCTOR BLOCK

→ Blocks inside constructors — behave like method blocks.

5. SYNCHRONIZED BLOCK

→ Used in multithreading to prevent thread interference.
→ Ensures only one thread can execute a block of code at a time.

Example:
```
synchronized (this) {
    // critical section
}
```

6. LOOP & CONDITIONAL BLOCKS

→ Blocks used inside loops, if-else, switch, etc.

Example:
```
if (a > b) {
    System.out.println("A is greater");
}
```

7. INITIALIZER BLOCKS

→ A general term for both static and instance blocks that initialize class members.

SUMMARY:
========
→ A block = code in `{}` with its own scope.
→ Useful for organizing code, limiting variable scope, and controlling execution order.
→ Most common: static block, instance block, and method block.

STATIC IN JAVA - class members

==========

→ `static` is a keyword used to define members (variables, methods, blocks, nested classes) that belong to the class itself rather than any specific object.

->Why static block is used?
-> Static is used when we want to execute instructions just before main method
>Static block is used to initialize a static variable
->We can create as many as blocks we want and will execute in the order we have written

->Why static method is used?
-> we use when method is same for different objects : for eg: milToKm method for all car will be same , then Car.milToKm

->Why static variables is used?: When one var is common for all objects, we give as static to use memory efficiently.
-When we use same variable with same value many times instead of using many times , to save memory we can just use static var .
Static variables is executed immediately after class is loaded and memory allocation is in method area.
-Static can be accessed by both static and non-static members, but instance members can only be accessed by non-static/instance (object-level) code.

WHY USE STATIC?

- Saves memory - only one copy is created and shared among all objects.
- Can be accessed directly using the class name (no need for an object).
- Useful for constants, utility methods, counters, etc.

TYPES OF STATIC MEMBERS

---

1. Static Variable: Static can be given only to instance var not local vars that can be inside method
   - Belongs to class, not instance
   - Shared by all objects
   - Memory is allocated in Method Area (Static Segment)

2. Static Method:
   - Can access only static data directly
   - Cannot access `this` or instance variables directly
   - Called using class name: `ClassName.methodName()`

3. Static Block:
   - Runs once when class is loaded
   - Used for static initialization

4. Static Class:
   - Inner class declared as static
   - Can only access static members of outer class
   -

ORDER OF EXECUTION:

---

1. Static Variable Declaration
2. Static Block Execution
3. Static Method (main()) Execution

STATIC MEMORY VIEW (JVM PERSPECTIVE)

When you run a Java program, JVM divides memory into segments:

- Code Segment
- Static Segment-not used at all (**Method Area** - prefered)(or **Meta space**)(Old name: **Permanent Generation(PermGen))**
- Heap
- Stack

→ All static members of a class are stored in Static Segment (Method Area)
→ Objects created using `new` are stored in Heap
→ Reference variables and method calls are stored in Stack

CLASS LOADER SUBSYSTEM (NOT "class load stack")

→ JVM has a Class Loader Subsystem which does the following:

1. **Loads** classes (.class files)
2. **Links** them (verifies, prepares)
3. **Initializes** static blocks & variables

→ When you create an object using `new`, JVM first checks if the class is already loaded:
   - If not, the **Class Loader** loads the class from disk into memory (Method Area)
   - Static variables are initialized
   - Then object is created in Heap
   - Its reference is stored in Stack

EXAMPLE TO UNDERSTAND STATIC + CLASS LOADER

```
public class Demo {
   public static void main(String[] args) {
      Car c = new Car();  // Creates Car object
      c.drive();

      System.out.println(Car.company); // Static variable accessed without object
      Car.showCompany(); // Static method accessed without object
   }
}

class Car {
   static String company = "TATA";  // static variable
   int speed;

   void drive() {
      System.out.println("Driving...");
   }

   static void showCompany() {
      System.out.println("Company: " + company);
   }

   static {
      System.out.println("Static block: Class Car is loaded");
   }
}
```

MEMORY FLOW

- Demo.class is loaded → main() runs
- Car c = new Car();
  - JVM loads Car.class (if not already)
  - Executes static block
  - Static variable `company` → Method Area (static segment)
  - Object c created → Heap
  - Reference c → Stack

OUTPUT:

```
Static block: Class Car is loaded
Driving...
TATA
Company: TATA
```

KEYWORDS:

- static = class level
- new = heap allocation
- Class Loader Subsystem = loads .class files
- Method Area = static segment (shared)
- Stack = stores method calls and local variables
- Heap = stores object data

INTERVIEW NOTES:

- JVM loads class only once.
- Static block runs once when class is loaded.

• Static methods cannot access non-static data directly.
• Class loader dynamically loads classes as needed.

7. COMMON INTERVIEW QUESTIONS:

Q1. Why is the `main()` method static?
Ans. So that JVM can call it without creating an instance of the class.

Q2. Can we overload static methods?
Ans. Yes. Static methods can be overloaded but not overridden.

Q3. Can a static method access non-static variables?
Ans. No. Because static context does not have access to instance-level (non-static) variables.

Q4. Can we have multiple static blocks in a class?
Ans. Yes. They will execute in the order in which they appear in the source code.

Q5. Is it possible to call a static method from a non-static method?
Ans. Yes. Static methods can be called from anywhere.

8. REAL-WORLD USE CASES:

• Counting number of objects created (using static counter)
• Utility methods like Math.max(), Math.sqrt() etc.
• Shared configurations or constants (e.g., static final PI = 3.14)
• Factory methods (like valueOf() in wrapper classes)

9. IMPORTANT MCQ SNIPPETS:
---
1) What will be the output?

```
class A {
   static int x = 10;
   static {
      x = x + 5;
   }
}
public class Test {
   public static void main(String[] args) {
      System.out.println(A.x);
   }
}
```

Ans: 15

2) Which of the following is true about static methods?
a) Can access static and non-static both
b) Cannot access static variables
c) Can access only static variables
d) Needs object to access

Ans: c) Can access only static variables

3) When is a static block executed?
a) When object is created
b) After main()
c) Before main()
d) At the end of execution

Ans: c) Before main()

Interview Questions:

```java
package accessModifiers;

class Test{

	//static vars
	static int a ,b;

	//static block
	static
	{
		a = 10;
		b = 10;
		System.out.println("Inside Static block");
	}

	//static function
	static void function1 (){
		System.out.println("Inside static method");
	}

	//instance vars:
	int x,y;

	//non-static/instance block:
	{
		System.out.println("Inside instance. non-static block");
	}

	//non-static function
	void function2() {
		System.out.println("Inside non static/instance method");
	}

	//default / non parameterized constructor
	public Test() {
		System.out.println("Inside constructor");
		x = 30;
		y = 40;
	}
}

public class StaticIntro {

	public static void main(String[] args) {

		//Java called as DYNAMIC language because : It loads class only when we need it but not all
together at once

		//Static :
		//JVM first will look for static varibales
		//Then looks for static blocks
		//next will look for static methods that is main method

		//Instance: And once statics are done , when we create object , first obj is created then
instance block is executed and then constructor and then the method(function2) called using object

		//No need of obj creation since we have used static keyword
		Test.function1();

		//Needs obj creation since its instance method
		Test t = new Test();
		t.function2();

		//o/p order : first static block >> static method(if its called) >> instance block >>
constructor >> instance method called using ref var of obj
	}

}



package accessModifiers;
```

```java
class Demo{
	 static int a;
	//int a;

	//Static block and instance block both can access the static variable
	static {
		System.out.println(a);
//		method1(); // calling the method1
	}

	//Static method and instance method both can also access the static variable
	static void method1() {
		System.out.println(a);
	}

	//static var can be accessed on constructor also
	public Demo() {
		System.out.println(a);
	}

	//Instance vars can be accessed by instance block ,instance method, constructor

	//Why cant instance var be accessed by static block and static method ?
	//: Because when class is loaded ,  static variables memory is allocated first in method area, but for instance its memory is allocated only when object is created, so when we try to access instance var inside static method or block before obj creation it shows error

}

public class Static {

	public static void main(String[] args) {

		Demo d = new Demo();
	 Demo.method1();
	}

}



package interviewCodeSnippets;

public class CodeSnippetTricky {

	int i = 1; // instance variable (belongs to object, not the class)

	public static void main(String[] args) {

//		CodeSnippetTricky c = new CodeSnippetTricky();
//		System.out.println(c.i++);

		//Uncomment below
		//System.out.println(i++); // ❌ Error: Cannot access non-static variable 'i' from a static context
					// 'main' is static → no object is created yet
					// To fix: make 'i' static → static int i = 1;
					// OR create object: new CodeSnippetTricky().i
	}

}



	package interviewCodeSnippets;

	class Em{
		static {
			System.out.println("Hi");
		}
```

```
	}
	public class CodeSnippetOnStatic {

		public static void main(String[] args) {

	//		Em e = new Em();

			System.out.println("Hello");


			//Output: Hello , because first static main method will be loaded and Hi will be executed only if its needed that e class is loaded i=here in main method by creating object.
			//But if we create a object for e, static block will be first executed first then Hello
		}

	}


class NextQues {

	    int a; // instance variable (belongs to the object, not class)

	    static {
	        System.out.println(a); // ❌ Error: Cannot make a static reference to non-static variable 'a'
	                               // Explanation: 'a' is tied to an object (instance of the class),
	                               // but static blocks run **before** any object is created,
	                               // so there's no instance memory available to refer to 'a'.
	    }

	    {
	        System.out.println(a); // ✅ Valid: This is an instance initializer block
	                               // It runs *when an object is created*, so instance variable 'a' exists
	    }
	}



package accessModifiers.Static;

import java.util.Scanner;

class Si{
	float si;
	float p;
	float t;
	static float r;
//	static float r=15.2f; can also give value to r like this but preferred method is giving value in static block


	static{
		r = 15.2f;
	}

	//we cant put the sentences hanging without method

	void accceptInput() {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the prinicipal amount: ");
		p=sc.nextFloat();

		System.out.println("Enter the tenure: ");
		t=sc.nextFloat();

//		r = 15.2f; //since it remains same throughout we can make it static which saves memoey also
	}

	void calculate() {
		si = (p*t*r)/100;
```

```
	}

	void display() {
		System.out.println(si);
	}

	//Java compiler gives default constructor if we dont give.
}

public class StaticExercise {

	public static void main(String[] args) {

		Si si = new Si();
		si.accceptInput();
		si.calculate();
		si.display();

		Si si1 = new Si();
		si1.accceptInput();
		si1.calculate();
		si1.display();

		Si si2 = new Si();
		si2.accceptInput();
		si2.calculate();
		si2.display();


	}

}



package accessModifiers.Static;

class Demo1{

	static int a;

	static{
		System.out.println("Inside static block");
	}

	static void method() {
		System.out.println("Inside static method");
	}
}



class OuterClass{ //Outer class cant be static

	static class innerClass{ //Inner class can be static which can access only static members of
outer class.

	}
}

public class StaticClass {

	public static void main(String[] args) {

		//Since we are accessing the static using class name not object we call it as Class
members.
		//Similarly instance var, blocks, methods are called as Object members/Instance
members(Object is Instance o     f class) since they need Object creation to execute.
		System.out.println(Demo1.a);
		Demo1.method();
```

```
            //Integer.pareseInt() - Integer is a class and parseInt is a static method hence called
directly using class name.
            //Integer.MAX_VALUE - MAXVALUE is also a static method

            System.out.println(Integer.MAX_VALUE);
            System.out.println(Integer.MIN_VALUE);
      }

}
```

Object oriented programming System:

- Oops is a system with Class , object, encapsulation, inheritance, polymorphism, abstraction
- Object-Oriented Programming is a programming paradigm based on the concept of "objects".
- These objects contain **data** in the form of **fields (attributes/variables)** and **code** in the form of **methods (functions/behaviors)**.
- OOP focuses on **modularity, reusability, scalability, and abstraction**.

Main Principles of OOP

1. **Class**
   - A blueprint or template from which objects are created.
   - It defines variables and methods common to all objects of that type.
2. **Object**
   - An instance of a class.
   - It has its own state and behavior defined by the class.
     Example:
     Car c = new Car();
     c.color = "Red";
     c.drive();
3. **Encapsulation**
   - Wrapping up of data (variables) and methods into a single unit (class).
   - Keeps data safe from outside interference and misuse.
   - Achieved using access modifiers like private, public, protected.
4. **Abstraction**
   - Hiding complex implementation details and showing only the essential features.
   - Helps reduce programming complexity.
   - Achieved using abstract classes and interfaces.
5. **Inheritance**
   - One class can inherit properties and methods from another class.
   - Promotes code reusability.
   - extends keyword is used in Java.
     Example:
     class Animal { void sound() {} }
     class Dog extends Animal { void sound() { System.out.println("Bark"); } }
6. **Polymorphism**
   - Ability of an object to take many forms.
   - Two types:
     a) Compile-time Polymorphism (Method Overloading)
     b) Runtime Polymorphism (Method Overriding)

Benefits of OOP

- Modularity – Each object forms a separate entity
- Code Reusability – Inheritance lets us reuse existing code
- Flexibility – Polymorphism allows the same method to behave differently
- Maintainability – Code is easier to update or debug
- Scalability – Makes codebase more manageable for large projects

PILLARS OF OOPS:

1. Encapsulation

2. Inheritance
3. Polymorphism
4. Abstraction

ENCAPSULATION:

- Encapsulation is one of the four fundamental OOP principles.
- It is the process of **wrapping data (variables)** and **code (methods)** together into a **single unit** (class).
- It is used to **hide the internal details of an object** and only expose what is necessary.
- In simple terms, Encapsulation is like a capsule — it binds everything (data + behavior) into one unit and protects it from outside interference.

Key Features of Encapsulation

- Data hiding using private keyword.
- Access provided using public methods (getters/setters).
- Prevents unauthorized access.
- Improves code maintainability and security.

Why Use Encapsulation?

- To protect data from being accessed directly (unauthorized access).
- To control how values are set or retrieved.
- To make the code flexible and easy to maintain.
- To follow the real-world concept of "information hiding".

How to Achieve Encapsulation in Java?

Encapsulation is implemented in Java by following these 4 steps:

1. Declare variables of a class as private.
2. Provide public **getter** methods to read the values.
3. Provide public **setter** methods to modify the values.
4. Keep logic inside setter to apply conditions (validation if needed).

Syntax Example: Encapsulation in Java

```
java
CopyEdit
// Class with encapsulated members
class Student {
   // Step 1: private variables (data hiding)
   private String name;
   private int age;
// Step 2: public getter method for name
   public String getName() {
      return name;
   }
// Step 3: public setter method for name
   public void setName(String newName) {
      name = newName;
   }
// Getter for age
   public int getAge() {
      return age;
   }
// Setter with validation for age
   public void setAge(int newAge) {
      if (newAge > 0) {
         age = newAge;
      } else {
         System.out.println("Age must be positive.");
      }
   }
}
```

```
java
CopyEdit
// Accessing the encapsulated class
public class Main {
   public static void main(String[] args) {
      Student s = new Student();
s.setName("Ananya");
      s.setAge(21);
System.out.println("Name: " + s.getName());
      System.out.println("Age: " + s.getAge());
   }
}
```

Real-World Analogy

Think of ATM Machine:

- You don't see how money is processed inside.
- You only interact through input (PIN, amount) and get the output (cash).
- The internal logic is encapsulated.

Benefits of Encapsulation

- **Code security**
- **Code maintainence.**
  - Improved data protection.
  - Prevents accidental modification of data.
  - Makes the code modular and reusable.
  - Easier to test and debug.
  - You can change internal logic without affecting other classes (since outside classes use only public methods).

Rules to Remember

- Always make variables private in encapsulated classes.
- Always access them using public methods.
- Use validation inside setter methods if needed.

INHERITANCE:

1. WHAT IS INHERITANCE?
- Inheritance is an OOP principle where one class (child) acquires properties and behaviors (fields and methods) from an existing /another class (parent).
- It supports reusability and method overriding (polymorphism).
- Java supports single, multilevel, and hierarchical inheritance directly.
- Multiple inheritance using classes is not allowed, but is achieved using interfaces called as Hybrid Inheritance.

2. KEY TERMINOLOGY:
- Superclass (Parent/Base class): The class whose features are inherited.
- Subclass (Child/Derived class): The class that inherits features from the parent class.
- extends: Keyword used to inherit from a class to class and interface to interface.
-implements : class to interface
- super: Keyword used to refer to the parent class (constructor, methods, variables).

3. SYNTAX:

```
class Parent {
   int x = 10;
   void display() {
      System.out.println("Parent class");
   }
}
```

```
class Child extends Parent {
   void show() {
      System.out.println("Child class");
   }
}

public class Test {
   public static void main(String[] args) {
      Child c = new Child();
      c.display(); // inherited method
      c.show();    // child method
   }
}
```

TYPES OF INHERITANCE IN JAVA:

1.Single Inheritance: One child inherits from one parent.

2.Multilevel Inheritance: A class inherits from a class that inherits from another class.

3.Hierarchical Inheritance: Multiple child classes inherit from a single parent.

4.Hybrid Inheritance: Combination of multiple types; Java supports it only through interfaces.

5.Multiple/Diamond/Diamond path Inheritance (with classes) is NOT supported in Java to avoid ambiguity (Diamond Problem).

```
             //Multiple/Diamond/Diamond path Inheritance/Diamond Problem : one child inheriting from
multiple classes which is not possible in java

             //Called as diamond because:
             //Even when we dont extend a class to any other class, JVM by itself will extend to a
class called Object Class which is a root/parent class that every class in Java inherits.
             // ParentC and Parent B will extend to Object, and Child21 extends to ParenrC and ParentA,
which creates diamond shape, hence called diamond problem.
```

6. Cyclic Inheritance: Two classes extending to each other is called as cyclic inheritance which is not possible.

```
//Here Alpha extends to Beta and Beta extends to Alpha

//class Alpha extends Beta{
//     float height  = 5.8f;
//     void readBooks() {
//            System.out.println("I love to read books");
//     }
//}
//
//class Beta extends Alpha{
//     float height  = 5.8f;
//     void readBooks() {
//            System.out.println("I love to read books");
//     }
//}
```

WHY JAVA DOESN'T SUPPORT MULTIPLE INHERITANCE USING CLASSES?

-Java avoids ambiguity caused by multiple inheritance (Diamond Problem).
-Instead, Java uses interfaces to support multiple inheritance behavior safely.

SUPER KEYWORD USAGE:

super.variableName – Access parent class variable.

super.methodName() – Call parent class method.

super() – Call parent class constructor (must be first statement in constructor).

Example:

```
class A {
   A() {
      System.out.println("A constructor");
   }
}

class B extends A {
   B() {
      super(); // optional, auto-called
      System.out.println("B constructor");
   }
}
```

CONSTRUCTOR CALLING IN INHERITANCE:

- When a subclass object is created, Java first calls the constructor of the parent class.
- This is done implicitly using super(), even if not written.
- This ensures proper initialization of the inherited part of the object.

METHOD OVERRIDING (Dynamic Polymorphism):

- A subclass can provide its own version of a method from the parent class.
- Signature must be the same.
- Enables runtime polymorphism.

Rules:

- Method name, return type, and parameters must be identical.
- Access level cannot be more restrictive than the superclass method, only can use the same access specifier level or increase the visibility not decrease.
- Cannot override final, static, or private methods.
- Co-variant return types: While overriding, we can change the return type to a subclass (co-variant) of the original return type, **only if the return type is a class (not primitive), i.e those classes must have inheritance only then we can change the return type to child class from parent class.**

```
//Rule 3: Co-variant return type: If we want to change the return type(class) while overriding we
can do it using co-variant return Type(i.e class/object), extending the classes

//If we want to change the return type while overrding a method from a class we make it co-variant
as below
class Animal{
}

class Tiger extends Animal {
}
```

```
	class Parent1 {
		Animal disp(int s) {
			Animal a = new Animal();
			return a;
		}
	}

	class child1 extends Parent1{


//Will show error as the return type has changed, but can make it happen by making it co-variant i.e
making the Class of the object we are creating to change the type extend to the class of the object
which was supposed to be same
//	@Override
	Tiger disp(int s, int a) { //can't give 2 parameters but can give if we remove override because
it changes to method - Overloading
		Tiger t = new Tiger();
		return t;
	}

//now will child 1 will have 2 methods :1 is inherited method and the other is method overloaded
method(When no @override annotation), (If override annotation is there then its just one overriden
method and must have same no. of parameters.
}
}
```

===================================
Example:

```
class Animal {
  void sound() {
    System.out.println("Animal makes sound");
  }
}

class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Dog barks");
  }
}
```

>>

Note: Methods can not only return datatypes/primitive datatypes but also non primitive datatypes/ Objects like Arrays, Strings, StringBuffer, builder and also Scanner object and also our own object of our own class.

```
class Alpha{
	String method() {
		return "Methods in Java Can also return Objects like Array String, Buffer, builder Scanner
object other than primitive datatypes";
	}

	StringBuffer method1() {
		StringBuffer sb = new StringBuffer();
		return sb;
	}

	Scanner scan() {
		return new Scanner(System.in);
	}

	Beta b() {
		return new Beta();
	}
}
```

>>

What is keywords in Java? - Inbuilt /reserved words like package, class, void, int float, public, static, private, final etc.

Final keyword: When we want a var, method or class to be not changed or to be constant we use final keyword

```
class keyw{
	final int a= 1;
	void change() {
//		a = 10; //error as a is final, cant change and will always be constant that is 1
	}
}
```

FINAL KEYWORD IN INHERITANCE:

- final class: Cannot be inherited. (Inbuilt - Math class)
- final method: Cannot be overridden. - Can inherit final method from parent class but cannot override that in child class.
- final variable: Cannot be changed (constant).- Can Inherit final var to child class but can't change (Inbuilt variable like PI in Math class)

STATIC METHODS IN INHERITANCE:

- Static variables and methods can be inherited from parent class but static methods cant be overridden.
- Static methods are not overridden but hidden when we try to override and is called **Method Hiding**.
- Accessed using class name or object, but behave according to the reference type.

```
package accessModifiers.Static;

class SuperClass{

	static int a =10;

	static void disp() {
		System.out.println("Inside static method in Super class");
	}

	static{
		System.out.println("Inside static block of superclass.");
	}

	{
		System.out.println("Inside instance block of superclass");
	}

	public SuperClass() {
		System.out.println("Inside SuperClass");
	}

}

class SubClass extends SuperClass{

	static{
		System.out.println("Inside static block of subclass.");
	}

	{
		System.out.println("Inside instance block of subclass");
	}

	public SubClass() {
```

```
		super();
		System.out.println("Inside SubClass constructor");
	}

	// Static method cannot be overriden
//	@Override
//	static void disp() {
//		System.out.println("Inside static block");
//	}

	//Now its not overridden its a new method in this class and called specialized methods
	static void disp() {
		System.out.println("Inside static method of sub class");
		//Now we have only one method, since inherited method has same name and same no of
parameters as this method no overloading also,
		//hence that parent class method will not be inherited but will be hidden inside sub class
called method hiding.
		//And that parent class method can be accessed through the subclass, only through super
class.
	}

//	@Override
//	public SuperClass() {
//	Cant override the constructor since the name of the class will not be anymore
//	}
}


public class InheritanceAndStatic {

	public static void main(String[] args) {

		//static variables get inherited by the sub class.
//		System.out.println(SubClass.a);

		//Static method also gets inherited but cant be overridden and if we try it will be hidden
and cant be accessed by the subclass and called as Method hiding
//		SubClass.disp();

//		SuperClass.disp();
//		SuperClass s = new SuperClass();

		SubClass su = new SubClass(); //Comment all other lines and understand the output order of
execution //First parents inherited block followed by subclass block like that


	}
}
```

IS-A RELATIONSHIP:

Inheritance represents an IS-A relationship.

Example: Dog IS-A Animal

HAS-A RELATIONSHIP (Composition):

When a class contains another class instance as a field.

Preferred over inheritance in many real-world use cases.

Promotes flexibility and loose coupling.

ACCESS MODIFIERS & INHERITANCE:

public: accessible everywhere.

protected: accessible in same package and subclasses (even outside package).

default (no modifier): accessible only within same package.

private: NOT accessible in subclasses directly.

```
package accesssModifiers;


public class AcessModifiers {

		/*
		 * Access Modifiers:
		 * private , public, protected , default(no modifier)
		 * */

	public int a; //can be accessed in any class in any package
	protected int b; //can be accessed only when in same package and in other packages only if it
inherits this class.
	int c; // Any class but in the package only //package/default
	private int d; //only in the same class

	void disp()  {
		System.out.println(a);
	}


}
```

OBJECT CLASS IN JAVA:

All classes in Java implicitly inherit from the Object class.

It provides common methods: toString(), equals(), hashCode(), getClass(), clone(), finalize() etc.

ABSTRACT CLASS VS INTERFACE:

Abstract class: partial abstraction (can have non-abstract methods).

Interface: full abstraction, multiple inheritance possible.

Use abstract class when:

You want to share code among closely related classes.

You need constructors, instance variables.

Use interface when:

You expect unrelated classes to implement it.

You want multiple inheritance.

INHERITANCE AND POLYMORPHISM:

Inheritance is the base for runtime polymorphism (method overriding).

Parent reference can hold child object:

Parent p = new Child(); // Allowed (upcasting)

CASTING IN INHERITANCE:

Upcasting: Child object is referred by parent class reference (safe).

Downcasting: Parent class reference is converted to child class (requires explicit cast, may cause ClassCastException).

Example:

Parent p = new Child(); // upcasting
Child c = (Child)p; // downcasting

ADVANTAGES OF INHERITANCE:

Simple : Code reusability, Reduces development time and effort, Increases profitability

- Code reuse: Common code in superclass.
- Less redundancy, easy maintainability.
- Enables polymorphism and extensibility.

DISADVANTAGES:

- Tight coupling: Subclass depends on superclass.
- Can break when superclass changes.
- Not suitable for every scenario (prefer composition in many cases).

WHEN TO USE INHERITANCE:

When there is a clear "is-a" relationship.

When reuse and polymorphism are required.

Avoid deep inheritance trees (prefer composition if things get complex).

NOTE: private vars cannot be ever inherited from the parent class to child class, because inheritance supports Encapsulation and doesnt leave private var to be inherited.

```
package objectOrientedProgramming;

class Demo11 {
	Demo11(){
		System.out.println("Inside parent constructor");
	}
}


class Demo22 extends Demo11{

	//Constructors will never get inherited to child class but will get executed because of super()
```

```
call.
	//Because if the Demo11 comes to child class name of constructor should be same as class name which will not happen, but however gets executed because super() in the child class will make execute the constructor from parent class

	Demo22(){
		System.out.println("Inside child constructor");
	}
}

public class ConstructorChaining2 {

	public static void main(String[] args) {

		Demo22 d = new Demo22();

	}

}
```

- Constructors are NOT inherited in Java.
  So, Demo11() is not inherited by Demo22.

- But when an object of Demo22 is created,
  the parent constructor Demo11() is executed automatically
  because of an implicit call to super() in Demo22 constructor.

- Reason:
  Every constructor (child or parent) must ensure its
  superclass is fully constructed first → Java does this by calling super()

- Note:
  Even though Demo11's constructor is not inherited,
  it still runs when you create a Demo22 object —
  because the compiler adds `super();` as the first line of Demo22()

Methods in Inheritance:

1. Inherited Methods: Using methods as it is from the parent class
2. Overriden Methods: Changing the methods behaviour by overriding the inherited methods
3. Specialized methods: Behaviours or methods that is found only in the child class .

```
package objectOrientedProgramming.inheritance;


//Hierarchical Inheritance
class Plane{
	void takeOff() {
		System.out.println("Plane Took off");
	}

	void fly() {
		System.out.println("Plane flight");
	}

	void land() {
		System.out.println("Plane has landed");
	}

}

class CargoPlane extends Plane{

	//This is Overridden methods
```

```
	@Override
	void fly() {
		System.out.println("Flies at lower height");
	}

	//land and takeOff are inherited methods

	//Below is specialized methods thats only inside child class
	void caryCargo() {
		System.out.println("This is a carry cargo");
	}
}


class PassengerPlane extends Plane{

	//This is Overridden methods
	@Override
	void fly() {
		System.out.println("It flies  at medium height");
	}

	//land and takeOff are inherited methods

	//Below is specialized methods thats only inside child class
	void carryPassenger() {
		System.out.println("This carries passenger");
	}
}

class FighterPlane extends Plane{

	//This is Overridden methods
	@Override
	void fly() {
		System.out.println("Flies at any height");
	}

	//land and takeOff are inherited methods

	//Below is specialized methods thats only inside child class
	void caryWeapons() {
		System.out.println("This carries Weapons");
	}
}

public class MethodsOfInheritance {

	public static void main(String[] args) {

			/*
			 * Methods of Inheritance:
			 * 1. Inherited Methods
			 * 2. Overridden Methods
			 * 3. Specialized Methods
			 * */

		System.out.println("Cargo Plane");
		CargoPlane c = new CargoPlane();
		c.takeOff();
		c.fly();
		c.land();
		c.caryCargo();
		System.out.println();

		System.out.println("Passenger plane");
		PassengerPlane p = new PassengerPlane();
		p.takeOff();
		p.fly();
		p.land();
		p.carryPassenger();
		System.out.println();
```

```
            System.out.println("Fighter Plane");
            FighterPlane f = new FighterPlane();
            f.takeOff();
            f.fly();
            f.land();
            f.caryWeapons();
            System.out.println();
      }

}
```

QUESTIONS:

1. **Can we extend our Class to a Inbuilt class like String, scanner, String buffer?**
   No because String, scanner, String Buffer are final class cannot be inherited.(can inherit Thread)

✅ Why Constructors Cannot Be Overridden:

1. Not inherited:
Constructors **are not inherited** by subclasses. Overriding only applies to **inherited methods**, so a subclass cannot override a constructor of the parent.
2. Constructor has same name as class:
Constructors must have the **same name as the class**, and since subclasses have **different names**, the constructor signature will always be different — making overriding impossible.
3. Constructor is not a method:
Even though it looks like one, a constructor is **not a regular method** — it's a **special block** to initialize an object. Overriding applies only to methods.

Polymorphism:

Polymorphism is one of the four fundamental Object-Oriented Programming (OOP) concepts (others: Inheritance, Encapsulation, Abstraction).
"Poly" means many and "morph" means forms — so Polymorphism means many forms. - one : many

Advantages :
-> Code flexibility
-> Code reduction
-> Reduction in complexity.

In Java, polymorphism allows us to perform a single action in different ways.

Real world example:
-> I am an example for polymorphism , i am daughter my parents, sister to my brother, i am cousin to cousin
-> carbon , water

Example:

```
package objectOrientedProgrammingPolyMorphism;

class Plan{
      void fly() {
            System.out.println("Plane flies");
      }

      void land() {
            System.out.println("Plane lands");
```

```
	}
}

class CargoPlan extends Plan{

	@Override
	void fly() {
		System.out.println("Cargo plane Flies at lower height");
	}

	void carryCargo() {
		System.out.println("cargo");
	}
}

class PassengerPlan extends Plan{
	@Override
	void fly() {
		System.out.println("Passenger plane Flies at medium height");
	}

	void carryPassenger() {
		System.out.println("Passengers");
	}
}

class FighterPlan extends Plan{
	@Override
	void fly() {
		System.out.println("Fighter plane Flies at any height");
	}

	void carryWeapons() {
		System.out.println("Fiehter");
	}
}

public class Polymorphism2 {

	public static void main(String[] args) {


		CargoPlan c =new CargoPlan();
		PassengerPlan p = new PassengerPlan();
		FighterPlan f = new FighterPlan();

		c.fly();
		p.fly();
		f.fly();
		//Till now there is no Polymorphism. because there is tight coupling which means child Object to child reference only..
		//Loose coupling is what gives us polymorphism which means child object to Parent reference.

		System.out.println("_____________________________________________________________");

		//Below gives us polymorphism through loose coupling as below.
		Plan refPlane;
		refPlane = c;
		refPlane.fly();
		refPlane.land();
		((CargoPlan)(refPlane)).carryCargo(); //Here down casting is done to access specialized method which couldn't be accessed otherwise

		refPlane = p;
		refPlane.fly();
		refPlane.land();
		((PassengerPlan)(refPlane)).carryPassenger(); //Here down casting is done to access specialized method which couldn't be accessed otherwise

		refPlane = f;
		refPlane.fly();
```

```
            refPlane.land();


            //Here just one refPlane(parent ref) reference variable gives different o/p hence now this
code has polymorphism.

            //We can access inherited and overridden methods from the parent reference and the child
object , but cant access specialized methods using parent reference
            //Because parent class doesn't have or knows about the specialized methods the child class
has

            //Hence to access the specialized methods through the parent we use explicit type casting
of parent reference to child which is here called as Down Casting when casting Parent to child as below

            ((FighterPlan)(refPlane)).carryWeapons(); //Here down casting is done to access
specialized method which couldn't be accessed otherwise


            //Note: Down Casting - when casting Parent type  to child type (not explicit)
            //           explicit casting  -when converting higher data type to lower data type.
      }

}


package objectOrientedProgrammingPolyMorphism;

class Plane1{
      void takeOff() {
            System.out.println("Took off");
      }

      void fly() {
            System.out.println("Flies");
      }

      void land() {
            System.out.println("Land");
      }
}

class CargoPlane1 extends Plane1{

      @Override
      void takeOff() {
            System.out.println("Cargo Plane Took off");
      }

      @Override
      void fly() {
            System.out.println("Crago Flies");
      }

      @Override
      void land() {
            System.out.println("Crago Landed");
      }
}

class PassengerpLane extends Plane1{
      @Override
      void takeOff() {
            System.out.println("Passenger Plane Took off");
      }

      @Override
      void fly() {
            System.out.println("Passenger Flies");
      }

      @Override
      void land() {
```

```
		System.out.println("Passenger Landed");
	}
}

class Fighterplane extends Plane1{
	@Override
	void takeOff() {
		System.out.println("Fighter Plane Took off");
	}

	@Override
	void fly() {
		System.out.println("Fighter Flies");
	}

	@Override
	void land() {
		System.out.println("Fighter Landed");
	}
}


//2nd way of achieving the polymorphism is create one class with a method passed with Parent reference
and call the methods in the parent class in it as below
class Airport{ //Preferred method

	public void permit(Plane1 ref) { //should be public
		ref.takeOff();
		ref.fly();
		ref.land();
	}
}

public class Polymorphism3 {

	public static void main(String[] args) {

		CargoPlane1 c = new CargoPlane1();
		c.fly();

		PassengerpLane p = new PassengerpLane();
		p.fly();

		Fighterplane f = new Fighterplane();
		f.fly();

		//No polymorphism till now , there is method overriding  but there is tight coupling
	//This method where Reference of superclass pointing to object of subclass is called Up casting.
//		Plane1 ref;
//
//		ref = c;
//		ref.fly();
//
//		ref = p;
//		ref.fly();
//
//		ref = f;
//		ref.fly();
		//Now there is polymorphism , since ref has many forms through loose coupling


		//This is one way to achieve polymorphism

		//There is 2nd way to that also. and this 2nd way is preferred
		Airport a = new Airport();
		a.permit(c); //Now for permit method in airport class which has parameter as parent ref we
are passing child object - bow ref = c;
		a.permit(p);
		a.permit(f);


	}
```

```
}
```

Types of Polymorphism:
---

1. Compile-Time Polymorphism (Static Binding / Early Binding) - Method Overloading
2. Run-Time Polymorphism (Dynamic Binding / Late Binding) - Method Overriding

---
1. Compile-Time Polymorphism

Also known as: Static Polymorphism or Method Overloading

Occurs when:
- Two or more methods in the same class have the same name but different parameter lists (type, number, or order).

How the compiler knows which method to call?
→ At compile time, based on method signature. Hence called as compile time

Example:

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

   double add(double a, double b) {
      return a + b;
   }

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

Key Points:

- Return type can be different but must not be the only difference.
- Overloading can occur in the same class.
- It improves readability.

Method Overloading Validity:
- Different number of arguments
- Different type of arguments
- Different order of arguments

Invalid Overloading:
- Same method name + same parameter list + only return type different → Compile-time error

---
2. Run-Time Polymorphism

Also known as: Dynamic Polymorphism or Method Overriding

Occurs when:
- A subclass provides a specific implementation of a method that is already defined in its superclass.

Binding:
→ The method call is resolved at runtime depending on the object type, not the reference type.

Example:

```
class Animal {
   void sound() {
      System.out.println("Animal makes sound");
   }
}

class Dog extends Animal {
   void sound() {
      System.out.println("Dog barks");
   }
}

class Main {
   public static void main(String[] args) {
      Animal a = new Dog(); // upcasting
      a.sound(); // "Dog barks"
   }
}
```

Key Points:

- Happens using method overriding
- Achieved via upcasting
- Java uses Virtual Method Invocation for dynamic binding

Method Overriding Rules:

- Method name, return type, and parameters must be exactly same.
- The overriding method cannot have a more restrictive access modifier.
- We can't override a final or static method.
- The overriding method can throw narrower or fewer checked exceptions.

Note:

Method overriding is used to achieve runtime polymorphism.

Covariant Return Type (Java 5+):

- While overriding, the return type can be subclass of the return type declared in the superclass.

Example:

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

class Parent {
   Animal getAnimal() { return new Animal(); }
}
class Child extends Parent {
   Dog getAnimal() { return new Dog(); }
}
```

EXAMPLE:

```
package objectOrientedProgrammingPolyMorphism;

//1. Compile-Time polymorphism - Method Overloading
class Compiletime {

	//One/more methods in a class with same name has different number, types, order of parameters
	//called as compile time because Java compiler decides which method based on the no of inputs
while compiling - disp(2,2);
	void disp(){
```

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

	void disp(int a) {
		System.out.println(a);
	}

	void disp(int a , int b) {
		System.out.println(a+b);
	}
}


//2. Run-Time Polymorphism - Method Overriding
class Parent11 {
	void marry() {
		System.out.println("marry");
	}
}

class Child11 extends Parent11{

	//One method has many forms hence its polymorphism
	//Here when we use override and call the method based on the object of a particular class, JVM
comes , and since JVM comes during runtime its called runtime polymorphism

	@Override
	void marry() {
		System.out.println(" no");
	}
}

class Child22 extends Parent11{

	void marry() {
		System.out.println("no..");
	}
}

class Child33 extends Parent11{
	void marry() {
		System.out.println(" no...");
	}
}

public class PolymorphismTypes {

	public static void main(String[] args) {

		//There are 2 types of polymorphism:
		//1. Compile-Time polymorphism/Static binding/Early Binding - Method Overloading
		//2. Run-Time Polymorphism/Dynamic binding/Late Binding/Dynamic Method Dispatch - Method
Overriding

		//1. Compile-Time polymorphism - Method Overloading
		Compiletime c = new Compiletime();
		c.disp(2, 2); //called as compile time because Java compiler decides which method based on
the no of inputs while compiling - disp(2,2);


		//2. Run-Time Polymorphism - Method Overriding
		Child11 ch= new Child11();
		ch.marry(); //Here when we use override and call the method based on the object of a
particular class, JVM comes , and since JVM comes during runtime its called runtime polymorphism

	}

}
```

Other Notes on Polymorphism in Java

Upcasting:

- Reference of superclass pointing to object of subclass.
- Used to achieve runtime polymorphism.
Example:
Animal a = new Dog(); // valid

Downcasting:

- Casting superclass reference back to subclass type.
- Must be done carefully with instanceof check.
Example:
Dog d = (Dog) a;

Benefits of Polymorphism:

- Code reusability
- Code extensibility
- Simplifies maintenance
- Improves flexibility and scalability

Real Life Examples:

- Method overriding in GUI components like Button → OnClick()
- Sorting different types of objects by overriding compareTo()

Polymorphism vs Overloading vs Overriding:

Feature	Overloading	Overriding
Definition	Same method name, diff params	Subclass redefines superclass method
Type	Compile-time	Runtime
Scope	Same class	Superclass-subclass
Return Type	Can be different	Must be same (or covariant)
Access Level	Irrelevant	Can't reduce visibility

Limitations:

- Constructors cannot be overridden.
- Static methods are class methods, so they cannot be overridden but can be hidden (method hiding).
- Private methods are not inherited, so cannot be overridden.

Keywords Involved:

- @Override : Annotation to denote overridden method
- super : Used to call superclass version of method
- instanceof : To verify object type before downcasting

Conclusion:

Polymorphism is a powerful concept in Java that enhances code modularity, flexibility, and maintainability by allowing objects to take many forms.

Abstraction:

★ What is Abstraction?
_____

Abstraction is the OOP concept where implementation details are hidden, and only the essential functionalities are exposed to the user.

->It is about focusing on what an object does instead of how it does it.
->

- **Variable can't be abstract** because **only methods** can have undefined (abstract) behavior, but variables must always have a **definite value or type**.
- **Constructor can't be abstract** because **constructors are never inherited**, and **abstract means "to be overridden in subclass"** — which doesn't apply to constructors.

✅ Hide internal complexities
✅ Show only necessary info
✅ Achieve separation of concerns

► Real-life Analogy:
_____

- ATM Machine: You insert card, enter PIN, and withdraw money.
  You don't know internal circuitry, protocol, encryption – only the end result.
- Mobile phone: You use UI without knowing underlying OS/hardware.

► Why Use Abstraction?
_____

- **Improves security** by hiding implementation
- **Provides flexibility** for code modification without breaking users
- Makes code more **modular**, **maintainable**, and **scalable**.

HOW JAVA SUPPORTS ABSTRACTION?

Java supports abstraction in two primary ways:

1. **Abstract Class** → Partial Abstraction (0–100%)
2. **Interface** → Complete Abstraction (Until Java 7)

★ 1. ABSTRACT CLASS

➤ Declared using `abstract` keyword
➤ Can have both abstract and non-abstract methods
➤ Cannot be instantiated directly

➤ Syntax:
_____

```
abstract class Vehicle {
   abstract void startEngine();     // abstract method
   void stopEngine() {              // concrete method
      System.out.println("Engine stopped");
   }
}
```

➤ Concrete Subclass:

```
class Car extends Vehicle {
   void startEngine() {
      System.out.println("Car engine started");
   }
```

```
}
```

➤ Main Method:

```
public class Main {
   public static void main(String[] args) {
      Vehicle v = new Car();   // Upcasting
      v.startEngine();
      v.stopEngine();
   }
}
```

► Output:

```
Car engine started
Engine stopped
```

➤ Key Features:

_____

✓ Can have constructors
✓ Can have final, static, non-static, and abstract methods
✓ Can contain instance and static variables
✓ Supports inheritance using extends

➤ Rules:

_____

✗ Cannot be instantiated
✓ Subclass must override all abstract methods OR be abstract itself
✓ Can use abstract class as base class in inheritance hierarchy

★ 2. INTERFACE

➤ Declared using interface keyword
➤ Supports 100% abstraction (Until Java 7)
➤ Multiple inheritance possible via interfaces

➤ Syntax:

_____

```
interface Flyable {
   void fly();  // implicitly public + abstract
}
```

➤ Implementation:

```
class Bird implements Flyable {
   public void fly() {
      System.out.println("Bird is flying");
   }
}
```

➤ Main Method:

```
public class Main {
   public static void main(String[] args) {
      Flyable f = new Bird();
      f.fly();
   }
}
```

► Output:
Bird is flying

═══

► Features of Interface:
_____

✓ All methods are public and abstract by default (till Java 7)
✓ All fields are public static final (constants)
✓ From Java 8:
→ Can have default and static methods (with body)
✓ From Java 9:
→ Can have private methods

► Java 8 Default Method Example:

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

► Java 8 Static Method:

```
interface Logger {
   static void log(String msg) {
      System.out.println("LOG: " + msg);
   }
}
```

═══

★ DIFFERENCE: ABSTRACT CLASS VS INTERFACE

═══

Feature	Abstract Class	Interface
Keyword	abstract	interface
Methods	Abstract + Concrete	Abstract only (till Java 7)
Constructors	Allowed	Not allowed
Variables	Any type	public static final only
Access Modifiers	Any	Only public
Multiple Inheritance	Not supported	Supported via multiple interfaces
Use	Common base for inheritance	Define capability contract
Java Version Changes	Stable	Major updates in Java 8 & 9

═══

★ WHEN TO USE WHAT?

═══

► Use Abstract Class when:
✓ You need partial abstraction
✓ You want to define non-abstract methods
✓ You want to provide a base class with default behavior
✓ You need constructors or fields

► Use Interface when:
✓ You need 100% abstraction
✓ You want to implement multiple inheritance
✓ You're defining "capability" contracts (e.g., Comparable, Serializable)
✓ You want to ensure that implementing classes follow a contract strictly

═══

★ REAL-TIME USAGE OF ABSTRACTION

✔ JDBC → You use Connection, Statement (Interfaces), not actual internal driver logic
✔ Servlet API → Abstract classes like HttpServlet
✔ Android → Listeners like OnClickListener are interfaces
✔ Frameworks → Spring Beans defined using interfaces, implementations hidden

★ COMMON THEORETICAL INTERVIEW TOPICS

- ☑ What is abstraction?
- ☑ How does Java implement abstraction?
- ☑ What is abstract class? Can it have constructor?
- ☑ Can abstract class have static/final methods?
- ☑ Can we create objects of abstract classes or interfaces?
- ☑ Can an abstract class be final? (✗ No)
- ☑ Interface vs Abstract class - which one to use when?
- ☑ Real-time use cases of abstraction in Java
- ☑ What's new in Java 8 and Java 9 for interfaces?
- ☑ Can interface extend another interface? (✓ Yes)
- ☑ Can abstract class extend interface? (✓ Yes)
- ☑ Can interface implement class? (✗ No)

★ SUMMARY

- Abstraction is about hiding how and showing only what
- Java uses abstract classes and interfaces to implement abstraction
- Abstract class → Partial abstraction, base class pattern
- Interface → 100% abstraction (before Java 8), multiple inheritance
- Choose wisely depending on need: behavior extension vs capability sharing
- Supports loose coupling, flexible architecture, testability

```
package objectOrientedProgramming.abstractoin;

//Rule in Java : In a class if there is a even one abstract method then the class should be also
abstract otherwise there will be error.
//And if class dont have abstract methods or has only fully implemented methods , its our wish to make
class abstract or not, but the thing is class can be abstract even if dont have abstract methods
abstract class Alpha{
	//Abstract method/incomplete method is a method with only method signature and no method body
or implementation.
//	abstract void disp(); // method implementation is hidden

	public Alpha() {
		//abstract class can have constructor
		System.out.println("Inside Alpha constructor");
	}
}


//abstract class can be inherited by normal class but this class should be also abstract if it
inherits abstract method, otherwise no need.
// and normal class(not abstract) can also inherit the abstract class.
// and abstract class can also inherit another abstract class, so conclusion is with abstract also
inheritance works normally like other classes
```

```
class Gamma extends Alpha{
	//default constructor is given by Java compiler automatically
	public Gamma() {
		//super(); by JVM
	}

}

public class Abstraction2 {

	public static void main(String[] args) {

		//Abstractions : Hiding the implementation and showing only the essential functions to
user is called abstraction.

		//2 ways :
		//1. Abstract : Its a keyword
		//2. Interface

		//1. abstract
		//Alpha a = new Alpha(); //error because we cant instantiate the abstract class or we
cant create object for abstract class.

		Gamma g = new Gamma(); //constructor of this class has super() given by JVM which will go
to suoer class constructor
		//so constructor of abstract class is called by creating object of the subclass.

	}

}
```

```
package objectOrientedProgramming.abstractoin;

abstract class Bird{
	abstract void eat();
	abstract void fly();
}

abstract class Eagle extends Bird{

	@Override
	 void fly() {
		 System.out.println("Flies at large height");
	 }
}

class SerpentEagle extends Eagle{

	@Override
	 void eat() {
		 System.out.println("Eats ");
	 }

 //fly() is inherited method
}

class GoldenEagle extends Eagle{

	@Override
	void eat() {
		System.out.println("Eats");

	}
	 //fly() is inherited method
}
```

```
// Method overriding/ polymorphism is there but to make it have with loose coupling following way
class Sky{

	//parent
//	public void EagleFamily(Eagle ref) {
//		ref.eat();
//		ref.fly();
//	}

	//main parent
	public void BirdFamily(Bird ref) {
		ref.eat();
		ref.fly();
	}
}


public class Abstraction {

	public static void main(String[] args) {

		SerpentEagle s = new SerpentEagle();
		s.fly();
		s.eat();

		GoldenEagle g = new GoldenEagle();
		g.eat();
		g.fly();

		//Method overriding with loose coupling
		Sky sk = new Sky();
//		sk.EagleFamily(s);
//		sk.EagleFamily(g);
		sk.BirdFamily(s);
		sk.BirdFamily(g);


	}

}
```

```
package objectOrientedProgramming.abstractoin;

abstract class Gama{
//	abstract int vari; //not possible
//	void p() {
////		abstract int a; //not possible
//	}

//	abstract public void Gama(); //not possible //constructor

//	abstract final void disp() ; //not possible because we have to implement abstract methods in subclass, but final will make it unchangeable, so not possible.
}



public class Abstraction3 {

	public static void main(String[] args) {

		//Abstract method - possible
		//Abstract class - possible
		//Abstract variable - Not possible
		//Abstract constructor - Not possible

		/*Variable can't be abstract because only methods can have undefined (abstract) behavior, but variables must always have a definite value or type.
```

```
Constructor can't be abstract because constructors are never inherited, and abstract means "to be
overridden in subclass" — which doesn't apply to constructors.*/

            //abstract + final class - not possible - class Gama can be either abstract or final, not
both because if we add final we can never inherit, then we can never instantiate abstract methods
            //abstract + final method -
            //abstract + final variable - not possible

      }

}
```

Aggregation and Composition: Together called as Association in Java.

1. What is Aggregation? - loose bound relation

- **Aggregation** is a form of **association** that represents a **"Has-A" relationship** between two classes, where:
  - One class contains a **reference** to another class. Through method
  - The contained object **can exist independently** of the container object.

In simple terms:

Aggregation is a **weak form of ownership**, where the life of the contained object is **not managed** by the container.

2. What is Composition? - Tight bound relation

- **Composition** is a stricter form of Aggregation.
  - It also represents a **"Has-A" relationship**, but with **stronger ownership**. Through creating Object
  - The contained object **cannot exist without** the container object.

In simple terms:

Composition is a **strong form of ownership**, where the container class is responsible for the **life and death** of the contained object.

📘 Real-World Analogy

Relationship	Example	Object Independence?
Aggregation	Department **has** Students	Students can exist without department
Composition	House **has** Rooms	Room cannot exist without the House

🔄 3. Code Example: Aggregation

```
class Address {
   String city, state;
Address(String city, String state) {
      this.city = city;
      this.state = state;
   }
}
class Employee {
   int id;
   String name;
   Address address; // Aggregation
Employee(int id, String name, Address address) {
      this.id = id;
      this.name = name;
      this.address = address;
   }
void display() {
      System.out.println(id + " " + name);
      System.out.println(address.city + ", " + address.state);
   }
}
public class AggregationExample {
   public static void main(String[] args) {
      Address addr = new Address("Bangalore", "Karnataka");
      Employee emp = new Employee(101, "Varshi", addr); // Passing address object
      emp.display();
   }
}
```

- **Note:** The Address object can still be used by other classes. It has an **independent lifecycle**.

🔄 4. Code Example: Composition

```
class Engine {
   Engine() {
      System.out.println("Engine created");
   }
void start() {
      System.out.println("Engine started");
   }
}
class Car {
   private final Engine engine; // Composition
Car() {
      engine = new Engine(); // Object created inside the class (tight coupling)
   }
void drive() {
      engine.start();
      System.out.println("Car is driving...");
   }
}
public class CompositionExample {
   public static void main(String[] args) {
      Car c = new Car(); // Engine is created when Car is created
      c.drive();
```

```
    }
}
```

🔹 **Note:** Engine has no life outside Car. This is **Composition**.

🔍 5. Key Differences: Aggregation vs Composition

Feature	Aggregation	Composition
Relationship Type	Weak “has-a”	Strong “has-a”
Object Lifecycle	Independent	Dependent
Contained Object Created?	Outside the container class	Inside the container class
Reusability	High	Low (Tightly coupled)
Example	Student has a Laptop	Human has a Heart

⚙️ 6. Why are Aggregation and Composition important?

- Promote **code reuse** and **modular design**.
- Help build **realistic models** of real-world systems.
- Used in **Spring Framework**, **ORMs like Hibernate**, and **Design Patterns**.
- Crucial for **object modeling** and **microservices** design.

👨‍💻 7. Important Java Points for Interviews

- Constructors define **composition**.
- Setter injection or constructor arguments define **aggregation**.
- In **UML diagrams**:
  - Aggregation is shown with a **hollow diamond**.
  - Composition is shown with a **filled black diamond**.

✅ 8. When to Use?

Use Case	Use
Independent reuse required	Aggregation
Part must belong to whole	Composition
Spring, Dependency Injection	Aggregation (via setter or constructor)
Immutable classes	Composition

```java
package aggregation.composition;

class Charger{
	private String brand;
	private float Voltage;

	//Specialized setter
	public Charger(String brand, float Voltage) {
		super();
		this.brand = brand;
		this.Voltage = Voltage;
	}

	public String getBrand() {
		return brand;
	}

	public float getVoltage() {
		return Voltage;
	}
}

class Os{
	private String name;
	private float size;
```

```
	public Os(String name, float size) {
		this.name = name;
		this.size = size;
	}

	public String getName() {
		return name;
	}

	public float getSize() {
		return size;
	}
}

class Mobile {

	//To achieve composition create a object of that particular class in our class
	Os os = new Os("Andriod", 4.5f); //this puts OS inside Mobile class - Composition - Tight bound

	//We achieve aggregation we create a method like this
	void hasA(Charger c){
		System.out.println(c.getBrand());
		System.out.println(c.getVoltage());
	}

}

public class AggregationAndComposition {

	public static void main(String[] args) {

		//Aggregation and Composition: Together called as a Association in Java
		//Aggregation - loose bound has-A relationship between classes through a method
		//Composition - tight bound has-A relationship between classes through a Object

		Charger c  = new Charger("SAMSUNG",(25.0f));

		Mobile m = new Mobile();

		//Aggregation can access directly from Charger
		System.out.println(c.getBrand());
		System.out.println(c.getVoltage());

		//Composition
		System.out.println(m.os.getName());
		System.out.println(m.os.getSize());

		//Lets say Mobile lost
		m = null; //this makes garbage collector collect this, and no longer that exists

		//Charger is still present
		System.out.println(c.getBrand());
		System.out.println(c.getVoltage());

		//but Os cant be accessed the below way - Composition
		System.out.println(m.os.getName());
		System.out.println(m.os.getSize());

	}

}
```

📋 INTERVIEW QUESTIONS

🔸 A. Theory-Based Interview Questions (for Google, Amazon, etc.)

What is the difference between Association, Aggregation, and Composition?
Can you give a real-world example of Composition in Java?
How do Aggregation and Composition affect Garbage Collection?

How are they represented in UML diagrams?
Can Composition replace Inheritance?
What happens if the composed object is null?
Why is Composition preferred over Inheritance in design principles?
Explain Has-A relationship with code.
When would you choose Aggregation over Composition?
Is Composition possible without using constructors?

🔸 B. Coding-Based Interview Questions

- ✅ **Implement a Student class which aggregates a Laptop class.**
- ✅ **Build a Car class that has multiple Wheels (Composition).**
- ✅ **Create a Library class that contains a list of Books (Aggregation).**
- ✅ **Build a Company class that creates Employees internally (Composition).**
- ✅ **Detect and implement proper relationship type from a UML diagram.**

🔸 C. Multiple Choice Questions (MCQs)

Which of the following is true about Aggregation?
a) Objects are tightly coupled
b) Contained object can live independently ✅
c) Objects must be created in the same class
d) None of the above
Which diamond symbol is used for Composition in UML?
a) Hollow Diamond
b) Black Filled Diamond ✅
c) Circle
d) Triangle
In Composition, the contained object is:
a) Created externally
b) Created outside main class
c) Created within and dependent on parent ✅
d) None
Which relationship shows strong ownership?
a) Association
b) Inheritance
c) Aggregation
d) Composition ✅
Which is preferred in modern design patterns?
a) Inheritance
b) Composition ✅
c) Aggregation
d) Static methods

Summary in 5 Points

Aggregation = weak has-a (independent life)
Composition = strong has-a (dependent life)
Use Aggregation when shared reuse is needed
Use Composition for strict ownership and encapsulation
Both are better than Inheritance for flexible design

Java toString() Method

==========

---
1. WHAT IS toString() METHOD?

- `toString()` is a method in the Object class in Java.
- It is used to return a string representation of an object.
- Syntax:
  public String toString()

- Every class in Java inherits this method from java.lang.Object.
- By default, it returns:
  ClassName@Hexadecimal_Hashcode

---
2. DEFAULT BEHAVIOR OF toString()

Example:

```
class Student {
   int id = 101;
}

public class Test {
   public static void main(String[] args) {
      Student s = new Student();
      System.out.println(s);
   }
}
```

Output:
Student@1a2b3c4d <-- Not readable!

Explanation:
- `System.out.println(s);` is internally calling `s.toString();`
- Since we did NOT override toString(), it prints the default string from Object class.

---
3. OVERRIDING toString() FOR CUSTOM REPRESENTATION

Example:

```
class Student {
   int id = 101;
   String name = "Alice";

   public String toString() {
      return "Student ID: " + id + ", Name: " + name;
   }
}

public class Test {
   public static void main(String[] args) {
      Student s = new Student();
      System.out.println(s);
   }
}
```

Output:
Student ID: 101, Name: Alice

WHY override?
- To get meaningful/logical string output of the object.
- Useful for debugging, logging, printing object details etc.

---
4. WHEN toString() IS AUTOMATICALLY CALLED

- System.out.println(object)
- String.valueOf(object)
- Concatenation with string: "Student info: " + object
- Logger.info(object)

All of these implicitly call object.toString()

---
5. BEST PRACTICES WHILE OVERRIDING toString()

- Include key fields to represent the object meaningfully.
- Avoid printing sensitive info (password, keys).
- Return human-readable format.

Format Suggestion:
```
   ClassName { field1=value1, field2=value2 }
```

Example:
```
class Book {
   String title;
   String author;

   public String toString() {
      return "Book { title=" + title + ", author=" + author + " }";
   }
}
```

---
6. toString() IN IDE-GENERATED CODE (e.g., Eclipse)

- Most IDEs can auto-generate toString() using all fields.
- Example:

```
@Override
public String toString() {
   return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "]";
}
```

---
7. COMPARISON: toString() vs equals()

- toString() → for representation (String output)
- equals() → for comparison (boolean output)
- Both can/should be overridden for custom behavior.

---
8. INTERVIEW QUESTIONS (THEORY)

Q1. What is toString() method in Java?
Q2. Why should you override toString()?
Q3. What is the default implementation of toString()?
Q4. How is toString() used during debugging?
Q5. Can you override toString() in every class?
Q6. Can static classes use toString()?
Q7. Is overriding toString() mandatory?
Q8. What's the difference between using toString() and System.out.println()?

---

9. INTERVIEW QUESTIONS (CODING)

Q1. Create a class `Car` with fields `model`, `year`. Override toString() to print them.

Q2. You are given a class `User` with private fields. Use constructor and toString() to display details.

Q3. Show how `toString()` behaves if not overridden. Then override and compare output.

Q4. Implement a `toString()` in a class `Library` which contains a list of books. Return a formatted string.

Q5. Print an array of objects using Arrays.toString(). How is toString() involved?

---

10. MCQ QUESTIONS (with Answers)

Q1. What is returned by default from Object's toString()?
A. Memory address
B. Object value
C. ClassName@HexHashCode ✅
D. Null

Q2. toString() is defined in which class?
A. String
B. Object ✅
C. Exception
D. System

Q3. Which of these statements calls toString()?
A. System.out.println(obj); ✅
B. obj = new String();
C. int i = obj;
D. All of the above

Q4. What happens if toString() is not overridden?
A. Compilation error
B. Runtime error
C. Default Object output ✅
D. None

Q5. Which method is commonly used with toString()?
A. equals()
B. finalize()
C. valueOf() ✅
D. hashCode()

---

11. EXTRA TIPS

✅ Always override toString() when creating data models.
✅ Use Lombok @ToString for auto-generation.
✅ Be careful not to leak sensitive info.
✅ Useful in debugging frameworks like Log4j, SLF4J, etc.

```
package toString;

 class Employee{ //extends Object //automatically done by JVM
	 private int id;
	 private String name;

	 public Employee() {
	}

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

	public int getId() {
		return id;
	}

	public String getName() {
		return name;
	}

	public String toString() { //must be public if not visibility is reduced and hence error
		return id+" "+name; //tostring() method gives us value of object instead of hash code
	}

//	public String toString() { //must be public if not visibility is reduced and hence error
//		return "Hello";
//	}

 }

public class ToString {

	public static void main(String[] args) {

		Employee e = new Employee(1,"Varshini");
		System.out.println(e); //This gives e ref var's address
		//but instead to print 1, Varshini we use toString() method which is inherited by
Object class automatically and this will be attached to ref ar e automatically by JVM like
e.toString()
		//if we give ourself e.toStrng() it print like this -> toString.Employee@6f539caf
		//But our goal is to print 1 Varshini we have to override(without annotation also
possible) the toString() method in from the parent object class as above

		//Object, ctrl+click open declaration and click ctrl+o
	}

}
```

```
package interviewCodeSnippets;

public class CodeSnippetsIV {

    public static void main(String[] args) {

        // Create an array of integers
        int[] a = {10, 20, 30, 40, 50};

        // Printing the array reference variable directly
```

```
        System.out.println(a);
        // OUTPUT: [I@<hashcode>
        // Explanation:
        // This prints the internal memory reference (like [I@1b6d3586)
        // because arrays inherit the default toString() from Object class.
        // [I --> denotes it is an array of int
        // @1b6d3586 --> the object's hashcode in hexadecimal

        // Create a String object
        String s = "JAVA";

        // Printing the String object directly
        System.out.println(s);
        // OUTPUT: JAVA
        // Explanation:
        // String class overrides the toString() method from Object class.
        // Its toString() returns the actual string value, not memory address.

        // Q: Why does the array 'a' print the address, but String 's' prints the value?
        // ANS:
        // Because the toString() method of Object class is:
        //    public String toString() {
        //        return getClass().getName() + "@" + Integer.toHexString(hashCode());
        //    }
        //
        // Arrays do NOT override this method → so it prints address-style output.
        // BUT the String class **overrides** toString() to return the actual string value.

        // Summary:
        // → Arrays use Object's default toString() → gives class@hexcode
        // → String overrides toString() → gives actual string value
    }
}
```

OBJECT CLASS IN JAVA - root class in java

---

1. INTRODUCTION:

-> The Object class is the root class of the Java class hierarchy.
-> Every class in Java implicitly inherits from Object (either directly or indirectly).
-> It is present in the java.lang package.

Syntax:

---

public class Object

---

2. CONSTRUCTOR OF OBJECT CLASS:

-> Object class has a public no-arg constructor:
  public Object()

-> When we create an object of any class, this constructor of Object is internally invoked.

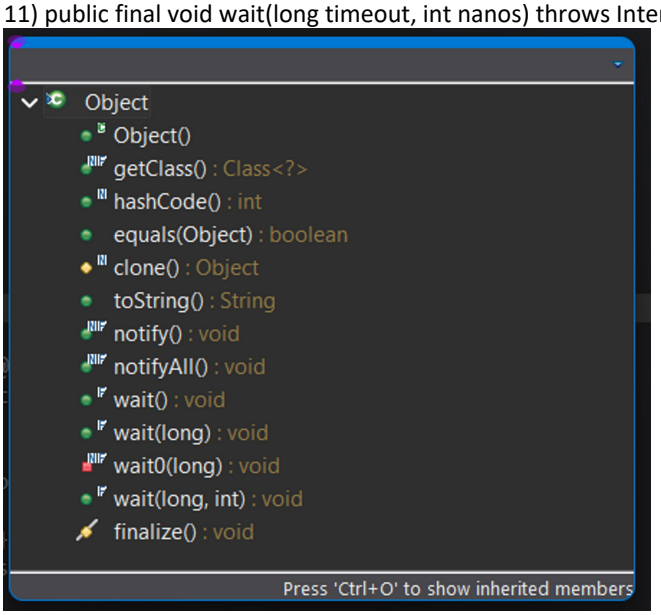
---

3. METHODS IN OBJECT CLASS (11 TOTAL):

Below are all the methods provided by the Object class:

1) public String toString()
2) public boolean equals(Object obj)

3) public int hashCode()
4) protected Object clone() throws CloneNotSupportedException
5) protected void finalize() throws Throwable
6) public final Class<?> getClass()
7) public final void notify()
8) public final void notifyAll()
9) public final void wait() throws InterruptedException
10) public final void wait(long timeout) throws InterruptedException
11) public final void wait(long timeout, int nanos) throws InterruptedException



---
4. DETAILED EXPLANATION OF EACH METHOD:
---

1) toString()

-> Returns a string representation of the object.
-> Default implementation: getClass().getName() + "@" + Integer.toHexString(hashCode())
-> Can be overridden to provide meaningful output.

Example:

```
public String toString() {
   return "Student[name=" + name + ", age=" + age + "]";
}
```

---

2) equals(Object obj)

-> Compares the current object with another object.
-> Default: checks reference equality (==).
-> Can be overridden for content comparison.

Example:

```
@Override
public boolean equals(Object obj) {
  if (this == obj) return true;
  if (obj == null || getClass() != obj.getClass()) return false;
  Student s = (Student) obj;
  return name.equals(s.name) && age == s.age;
}
```

---

3) hashCode()

-> Returns an integer hash code value for the object.
-> Used in hashing-based collections (HashMap, HashSet).
-> If equals() is overridden, hashCode() must also be overridden.

---

4) clone()

-> Creates and returns a copy (clone) of the object.
-> Class must implement Cloneable interface to use this.
-> Uses shallow copy by default.

Example:

```
public class Person implements Cloneable {
   public Object clone() throws CloneNotSupportedException {
      return super.clone();
   }
}
```

---

5) finalize()

-> Called by the garbage collector before destroying an object.
-> Used to release resources.
-> Deprecated since Java 9.

---

6) getClass()

-> Returns runtime class information of the object.

Example:

```
Student s = new Student();
System.out.println(s.getClass().getName());
```

---

7) notify(), notifyAll(), wait()

-> Used for thread synchronization.
-> Defined in Object class because every object in Java has a monitor lock.

---
5. WHY OBJECT CLASS METHODS ARE FINAL?

-> Methods like wait(), notify(), notifyAll(), and getClass() are marked final to ensure thread-safety and security.

---
6. OBJECT CLASS IN INHERITANCE:

-> Every class implicitly extends Object.
-> If no superclass is mentioned, Java automatically extends Object.

Example:

```
class A {
```

```
   // implicitly: extends Object
}
```

---

7. COMMON INTERVIEW THEORY QUESTIONS:

Q1. What is the superclass of every class in Java?
-> java.lang.Object

Q2. Why should equals() and hashCode() be overridden together?
-> To maintain the general contract for hash-based collections.

Q3. What is the default implementation of toString()?
-> getClass().getName() + "@" + hashCode()

Q4. What does getClass() method return?
-> Returns runtime class (object's class metadata)

Q5. What is the use of clone()? Can it create deep copy?
-> clone() creates shallow copy by default.
-> To perform deep copy, we override clone() manually.

---

8. CODING QUESTIONS ON OBJECT CLASS:

Q1. Override toString() for a class Student.

Q2. Create a class Employee and override equals() and hashCode() to compare name and id.

Q3. Demonstrate use of clone() with shallow and deep copy.

Q4. Use getClass() to print runtime class of an object.

Q5. Write a program using wait(), notify() with proper synchronization block.

---

9. MCQ QUESTIONS:

Q1. Which of the following is not a method of Object class?
a) equals()
b) compareTo()
c) clone()
d) hashCode()

Ans: b) compareTo()

Q2. What is the return type of getClass() method?
a) String
b) Class
c) Object
d) void

Ans: b) Class

Q3. Which interface must be implemented to use clone()?
a) Serializable
b) Cloneable

c) Comparable
d) AutoCloseable

Ans: b) Cloneable

Q4. Which of the following methods is called by garbage collector before destroying the object?
a) destroy()
b) delete()
c) finalize()
d) end()

Ans: c) finalize()

Q5. Which method is used for synchronization in Java?
a) wait()
b) notify()
c) notifyAll()
d) All of the above

Ans: d) All of the above

---
10. QUICK SUMMARY:

-> Object is the root class for all Java classes.
-> Contains 11 methods – most commonly used are toString(), equals(), hashCode(), clone().
-> For synchronization, wait(), notify(), and notifyAll() are used.
-> If you override equals(), override hashCode() too.
-> getClass() is used in reflection and debugging.
-> finalize() is deprecated – don't use in modern code.

Java Keywords: this, super, final, this(), super()

➤ 1. `this` Keyword in Java

Definition:

`this` is a reference variable in Java that refers to the **current object**.

Usage:
---
1. To refer to **current class instance variables**
2. To **invoke current class methods**
3. To **invoke current class constructor** using `this()`
4. To **pass current object** as an argument
5. To return current object from method
6. To resolve **naming conflict** between instance and local variables

Example 1: Refer instance variable

```
class Student {
   int id;
   String name;

   Student(int id, String name) {
      this.id = id;      // this.id → instance variable
```

```
        this.name = name;   // name → local variable
    }
}
```

Example 2: Call current method

```
void display() {
    System.out.println("Hello");
}
void show() {
    this.display();  // same as just calling display()
}
```

Example 3: Pass current object

```
void print(Student s) {
    System.out.println(s.name);
}
void callPrint() {
    print(this);  // passing current object
}
```

➤ 2. `this()` Constructor Call

Definition:

- Used to call another constructor in the **same class**
- Must be the **first line** inside the constructor

Example:

```
class Test {
    Test() {
        this(5);
        System.out.println("Default constructor");
    }
    Test(int x) {
        System.out.println("Parameterized constructor " + x);
    }
}
```

Output:

```
Parameterized constructor 5
Default constructor
```

Note:

- Only one `this()` call allowed per constructor
- Cannot be used inside methods (only constructors)

➤ 3. `super` Keyword in Java

Definition:

`super` is a reference variable used to refer to the **immediate parent class object**.

Usage:
---
1. To access **parent class variables**
2. To invoke **parent class methods**

3. To call **parent class constructor** using `super()`

Example 1: Access parent variable

```
class Parent {
   int num = 10;
}
class Child extends Parent {
   int num = 20;

   void print() {
      System.out.println(super.num); // prints 10
   }
}
```

Example 2: Call parent method

```
class Parent {
   void display() {
      System.out.println("Parent");
   }
}
class Child extends Parent {
   void display() {
      super.display(); // calls Parent's display
      System.out.println("Child");
   }
}
```

➤ 4. `super()` Constructor Call

Definition:

- Used to call parent class constructor from subclass constructor
- Must be the **first line** in the subclass constructor

Example:

```
class Parent {
   Parent() {
      System.out.println("Parent Constructor");
   }
}
class Child extends Parent {
   Child() {
      super(); // optional if parent has default constructor
      System.out.println("Child Constructor");
   }
}
```

Output:

```
Parent Constructor
Child Constructor
```

Note:

- If parent does not have a no-arg constructor, you **must explicitly call super(args)** with matching arguments.
- Java adds `super()` implicitly if not written explicitly (but only if parent has default constructor)

➤ 5. `final` Keyword in Java

Definition:

The `final` keyword in Java is used to create constants, prevent method overriding, and inheritance.

Uses:
---
1. Final **variable** → cannot be reassigned
2. Final **method** → cannot be overridden
3. Final **class** → cannot be inherited

1. Final Variable (constant)

```
final int x = 10;
x = 20; // Error: cannot assign value to final variable
```

- Must be initialized at declaration or in constructor (if instance variable)

2. Final Method

```
class Parent {
   final void show() {
      System.out.println("Parent");
   }
}
class Child extends Parent {
   void show() { } // Error: cannot override final method
}
```

3. Final Class

```
final class Animal {
   void eat() {
      System.out.println("eating...");
   }
}
class Dog extends Animal { } // Error: cannot inherit from final class
```

Note:

- Final methods improve performance (no dynamic binding)
- Final reference variables (objects) → reference can't change, but internal state can
  Example:
  final List<String> list = new ArrayList<>();
  list.add("Java"); // allowed

➤ Summary Table:

Keyword	Purpose	Usage Scope
this	Refers to current class object	Variables, Methods
this()	Calls constructor in same class	Constructor only
super	Refers to parent class object	Variables, Methods
super()	Calls parent constructor	Constructor only
final	Prevents modification (var/method/class)	Variable/Method/Class

---

➤ Real-World Applications:

- `this()` and `super()` used in constructor chaining
- `final` variables used for configuration/constants
- `super` used to extend parent class behaviors without rewriting code

```java
package objectOrientedProgramming.inheritance;

class SuperClass{
	int i = 1;

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

class SubClass extends SuperClass{
	int i = 10;

	void disp() {
//		System.out.println(i); //i = 10 , because its from this subclass.
		System.out.println(super.i); //o/o i =1; because we are calling i var from superclass using super keyword.
	}

	@Override
	void display() {
		System.out.println("Hi"); //prints hi
		//but if we want to print the super class's Hello then we use super keyword again as below
		super.display(); //super class's method is executed.
	}

}

public class SuperKeyword {

	public static void main(String[] args) {

		SubClass s = new SubClass();
		s.disp();

		s.display(); //"Hi"

	}

}
```

INTERFACES IN JAVA

---

1. INTRODUCTION:

- An interface in Java is a blueprint of a class & is used to achieve pure abstraction and standardization and multiple inheritance.
- It is a contract that specifies what a class should do, but not how.
- All methods in an interface are implicitly abstract and public (till Java 7).
- Interfaces support full abstraction in Java.
- The class implementing the interface must provide an implementation for all methods.

RULES :

1. Interface is used to achieve pure abstraction and standardization

2. Interface promotes polymorphism by allowing an interface ref to point to objects of implementing classes, This achieves loose coupling , reduces code and provides code flexibility.
3. Methods in interface are automatically public and abstract.
4. Specialized methods cant be accessed directly using interface type reference but can access using down casting the interface reference to subclass type
5. If a class partially implements interface, it must be declared as abstract since it will inherit the unimplemented abstarct method.
6. A class can implement multiple interfaces because diamond shape problem does not exist as interface don't have parent.
7. An interface cant implement another interface, because interface cannot provide methods with bodies inside it.
8. A interface can extend another interface. Not only this it can inherit from multiple interface because diamond problem doesn't exist. Multiple inheritance in java can be indirectly achieved by making use of the in the interfaces.
9. A class can both extend another class as well as implement an interface, order should be extend first and implement later.
10. An interface can contain constant variables and method signature. A variable within an interface is automatically public static final. A method will be automatically public.
11. An empty interface in Java is referred to as Marker interface or Triggered interface and is used to provide properties to the object of the class. - **one inbuilt interface which is empty is Serializable.** (Serialization is where we can create permanent object that will be stored in Hard disk instead of RAM, classes which we create is stored in RAM which is volatile)
12. An object of interface cannot be created because interface is collection of abstract methods. However, reference can be created so that loose coupling, polymorphism and its advantages can be achieved.

---
2. SYNTAX:

```
interface InterfaceName {
   // abstract method
   void method1();

   // Java 8: default method - to backward compatibility
   default void defaultMethod() {
      System.out.println("Default method in Interface");
   }

   // Java 8: static method - to access directly through interface.
   static void staticMethod() {
      System.out.println("Static method in Interface");
   }

   // Java 9: private method (helper) - to remove duplicate code- code redundancy
   private void helper() {
      System.out.println("Helper method");
   }
}

class MyClass implements InterfaceName {
   public void method1() {
      System.out.println("Implemented method1");
   }
}
```

```
package objectOrientedProgramming.abstractoin;

import java.util.Scanner;

//interface dont have any parent by default not even object
interface Theta{ //automatically there will be abstract for both interface and methods in it , since interface will take only abstract method
	void disp(); //JVM will make any methods in interface as public and abstract by default
}

class Omega implements Theta{

	@Override
	public void disp() { //must be public because in interface all methods will be public by default , and when implementing that method we cant decrease the visibility/access level to package/default
```

```
			System.out.println("Implementing inherited abstract methos=ds from interface in this class
which is implemented to that interface.");
		}
}

//-------------------------------------------------

interface Calculator{
	public void add();
	public void sub();
}

abstract class MyCalci1 implements Calculator{
	@Override
	public void add() {
		int a = 1;
		int b = 1;
		System.out.println(a+b);
	}

	//sub() is here inherited method and since its is abstract method we will have to make class as
abstract also
}

class MyCalci2 implements Calculator{
	@Override
	public void add() {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the first number: ");
		int a  =sc.nextInt();
		System.out.println("Enter the second number: ");
		int b = sc.nextInt();
		System.out.println(a+b);
	}

	@Override
	public void sub() {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the first number: ");
		int a  =sc.nextInt();
		System.out.println("Enter the second number: ");
		int b = sc.nextInt();
		System.out.println(a-b);
	}
}

class MyCalci3 implements Calculator{
	@Override
	public void add() {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the first number: ");
		int a  =sc.nextInt();
		System.out.println("Enter the second number: ");
		int b = sc.nextInt();
		if(a>b) {
			System.out.println(a+b);
		}else {
			System.out.println("Invalid!!");
			System.exit(0); //will terminate the program, nothing will execute next
		}
	}

	@Override
	public void sub() {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the first number: ");
		int a  =sc.nextInt();
		System.out.println("Enter the second number: ");
		int b = sc.nextInt();
		if(a<b) {
			System.out.println(a-b);
		}else {
```

```
                System.out.println("Invalid!!");
                System.exit(0);
            }
        }

        public void mul() {
            int a=1, b=2;
            System.out.println(a*b);
        }
}

//polymorphism
class Calci {
        void calci(Calculator c) { //This is just ref of interface not Object and we cant create object
also
            c.add();
            c.sub();
        }
}

public class AbstractionWithInterface {

        public static void main(String[] args) {

            //2. Abstraction with interface.

            //Disadvantages of Abstraction using abstract keyword
            /*
             * 1. Can achieve pure and impure abstraction also, since in a abstract class in can have
both complete and abstract methods or only abstract methods.
             * 2. It can be used when we need to achieve the partial abstraction
             *
             * */

            /*==> Pure abstraction can be achieved by interface and also can achieve multiple
inheritance
            ==> We cant create Object for interface similar to abstract class since they have abstract
methods
            ==> We can achieve polymorphism with interface by passing child class objects to parent
interface reference.
            ==> Methods in interface are automatically public and abstract
            ==> Specialized methods cant be accessed directly using interface type reference but can
access using down casting the interface reference to subclass type
            ==> A class can implement multiple interfaces because diamond shape problem does not exist
as interface don't have parent.
            ==>
            */

            //implementing classes
//          MyCalci1 c = new MyCalci1();
            MyCalci2 cc = new MyCalci2();
            MyCalci3 ccc = new MyCalci3();

            Calci l = new Calci();
//          l.calci(c);
            l.calci(cc);
            l.calci(ccc);

            //Parent ref
            Calculator ref;
            ref = ccc;
//          ref.mul(); //we cant access the specialized method of a class extended to interface using
reference of interface parent and up casting but possible indirectly using down casting as below.
            ((MyCalci3)(ref)).mul();
        }
}
```

```
package objectOrientedProgramming.abstractoin;
```

```java
//import java.util.Scanner;
interface Calculator1{
	void add();
	void sub();
}

interface Calculator2{
	void mul();
	void div();
}

class MyCalculator implements Calculator1,Calculator2{

	@Override
	public void mul() {
		System.out.println("Multiplication");
	}

	@Override
	public void div() {
		System.out.println("Dividing");
	}

	@Override
	public void add() {
		System.out.println("Adding");
	}

	@Override
	public void sub() {
		System.out.println("Subtracting");
	}

}

//-----------------------------------------------------
//Industry preferred multiple inheritance using Interface

interface One{
	void add();
}

interface Two{
	void sub();
}

interface Three extends One, Two{
	void mul();
}

class One1 implements Three{

	@Override
	public void add() {
		System.out.println("Add");
	}

	@Override
	public void sub() {
		System.out.println("Sub");
	}

	@Override
	public void mul() {
		System.out.println("Mul");
	}
}

//-----------------------------------------------------
// A class can implement a interface and extend to a class at the same time, but first extend then
implement as below.
```

```java
interface Four{
	void add();
}

class Two2{
	void addition() {
		System.out.println("additiom");
	}
}

class Three3 extends Two2 implements Four{

	@Override
	public void add() {
		System.out.println("Add");
	}
}

//-------------------------------------------------------
//Interface can have variable butJava will automatically attach public , static and final
interface Demo{
	int COUNT = 2; //automatically java will make it as public static final int count = 2;
//	void fun(); //automatically java will make it as public void fun();
}

class Demo21 implements Demo{
	//static var will be inherited hence count will be inherited here
	void add() {
//		COUNT = 2; //will get error that final field cant be assigned
	}
}

//-----------------------------------------------------------------
//Interface which is empty is called as empty interface or marker interface or triggered interface, Tagged interface
//One inbuilt interface which is empty/marker/tagged interface is Serializable
interface Demo1{

}

//-----------------------------------------------------------------------

public class MultipleInheritanceWithInterface2 {

	public static void main(String[] args) {

		//Multiple inheritance using interface : One class can implement 1/more interfaces
		//and interface is not extended to any parent by default not even object so there will be no diamond problem also.

		MyCalculator c = new MyCalculator();
		c.add();
		c.sub();
		c.mul();
		c.div();


		//Multiple inheritance
		One1 o = new One1();
		o.add();
		o.sub();
		o.mul();
	}
}
```

---

3. FEATURES OF INTERFACE:

- Cannot instantiate interfaces directly.
- Can extend multiple interfaces (supports multiple inheritance).

- A class can implement multiple interfaces (overcomes Java's single inheritance limitation).
- All variables are:
  → public
  → static
  → final (constants)
- All methods are:
  → public
  → abstract (until Java 7)

---
4. DIFFERENCE: ABSTRACT CLASS VS INTERFACE

Feature	Abstract Class	Interface
Keyword	abstract	interface
Methods	Abstract + Concrete	Abstract only (Java 7)
Constructors	Allowed	Not allowed
Variables	Any type	public static final only
Access Modifiers	Any	Only public
Multiple Inheritance	Not Supported	Supported
Use	Common base	Define capability/contract
Java Updates	Stable	Major in Java 8 & 9

---
5. JAVA 8 & 9 ADDITIONS TO INTERFACE:

- Default methods (Java 8): Provide method body in interface (used for backward compatibility)
  → Can be overridden by implementing class.
- Static methods (Java 8): Belong to the interface, not to instances. - used to access directly with interface name
  → Called via interface name.
- Private methods (Java 9): Used for internal logic reuse among default/static methods. - to avoid duplicate code- code redundancy.

```java
package objectOrientedProgramming.abstractoin;

interface Demo11{
	void disp();

	//default and static came in Java 8
	default void disp1() { //when we want to give a body to method inside interface we can use abstract, and can be implemented/overriden in class
		System.out.println("Default");
	} //default methods are only allowed inside interfaces

	static void disp2() {
		System.out.println("..");
		//Static methods with body can be given in interface as we can access static methods without the need of creation of objects
	}

	//private came in Java 9
	private void disp3() {
		System.out.println("Private method can be given with body inside interfaces");
	}
}

class Def implements Demo11{

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

	@Override
	public void disp1() {
		System.out.println("..");
	}
}
```

```java
//--------------------------------------------------------------------

//class Bet{
//	//default methods are only allowed inside interfaces default void add() {
//		System.out.println(".");
//	}
//}

//--------------------------------------------------------------------
@FunctionalInterface
interface fun{
	void add1();
	default void add() {  //its possible to give one/more default method inside the functional interface along with one abstract method(must), i.e cant give only one default method only there must me one abstract method along with it

	}

	static void add3() {//its possible to give one/more static method inside the functional interface along with one abstract method(must), i.e cant give only one static method only there must me one abstract method along with it

	}

	private void add4() {//its possible to give one/more static method inside the functional interface along with one abstract method(must), i.e cant give only one static method only there must me one abstract method along with it

	}

	//can give as many as default and static methods, private methods and can also give them together, but there must be a abstract method
}

//--------------------------------------------------------------------------------

//To reduce redundancy in the code i.e repeating of code

interface Student{ //no need of class to implement if there is no abstract method
	static void studentLife() {
//		System.out.println("Student");
//		System.out.println("is student");
		students();
	}

	static void student() {
//		System.out.println("Student");
//		System.out.println("is student");
		students();
	}

	//both method has same so one method with same lines of code can be called inside them

	private static void students() { //when we dont want a method to be in inheritance and cant be accessed we use private
		System.out.println("Student");
		System.out.println("is student");
	}
}

//--------------------------------------------------------------------------------

interface SeaAnimals{
	void eat();
	void swim();
	//Whenever new feature/method has to be added , if there are many classes its not possible to override it in all of them, so we can give method with body along default keyword

	default void communicate() { //default is not access modifier its a keyword, public is there automatically
		System.out.println("Sea animals communicate");
	}
```

```
}

class Shark implements SeaAnimals{

	@Override
	public void eat() {
		System.out.println("Sharks eat humans");
	}

	@Override
	public void swim() {
		System.out.println("Sharks swim");
	}

}

class Dolphin implements SeaAnimals{

	@Override
	public void eat() {
		System.out.println("Dolphins eat fish");
	}

	@Override
	public void swim() {
		System.out.println("Dolphins swim");
	}

}

public class DefaultAndStaticMethodInInterface {

	public static void main(String[] args) {

		Def d = new Def();
		d.disp();
		d.disp1();
//		d.disp2(); //static methods wont participate in inheritance when its within interface but
can when its class inheritance.

		Demo11.disp2(); //static method can be accessed directly with the name of interface like
this

		//---------------------

		Shark s = new Shark();
		s.eat();
		s.swim();

		Dolphin dolp = new Dolphin();
		dolp.eat();
		dolp.swim();
	}

}
```

---

6. EXAMPLE:

```
interface Animal {
   void eat();            // abstract method
   default void sleep() {   // default method
      System.out.println("Animal sleeps");
   }
   static void breathe() {   // static method
      System.out.println("Animal breathes");
   }
}
```

```
class Dog implements Animal {
   public void eat() {
      System.out.println("Dog eats bone");
   }

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

public class Test {
   public static void main(String[] args) {
      Dog d = new Dog();
      d.eat();            // Dog eats bone
      d.sleep();          // Dog sleeps
      Animal.breathe();    // Animal breathes
   }
}
```

---
7. WHY INTERFACE?

- To achieve **100% abstraction**.
- For **loose coupling** between code.
- To achieve **multiple inheritance** in Java.
- To define common **behavior contracts** for unrelated classes.

---
8. TYPES OF INTERFACES:
---
1. **Normal Interface**: Only abstract methods (before Java 8).
2. **Functional Interface**: Has **only one abstract method** (Java 8+) → Used in Lambda.
   → Example: Runnable, Comparable, Callable
3. **Marker Interface**: Has **no methods**, just tagging.
   → Example: Serializable, Cloneable

---
9. MARKER INTERFACE:

- Interface with no methods or fields.
- Used to tag classes with some property.
- JVM/Framework checks marker to apply behavior.
  → Example: `Serializable` marks class for Java serialization.

---
10. FUNCTIONAL INTERFACE:

-Interface with exactly one abstract method which can be implemented through a independent class/inner class(provides security)/ Anonymous Inner class(with more security)/lambda expressions for more security and to reduce the lines of code.
- Can contain default/static methods.
- Used in lambda expressions, streams, etc.
    - Inbuilt functional interfaces are comparable, comparator, supplier, runnable
  → Annotated with `@FunctionalInterface`

Example:

```
@FunctionalInterface
interface Greeting {
   void sayHello();
}

public class Test {
   public static void main(String[] args) {
      Greeting g = () -> System.out.println("Hi!");
      g.sayHello();
```

```
  }
}
```

```
package objectOrientedProgramming.abstractoin;

@FunctionalInterface
interface FunctInte{
	void disp(int a); //can give parameters
//	void disp2(); // not possible because Functional Interface can have only one abstract method
}

//One way to implement Functional interface is through class and other is inner class which will provide us security and also other way is / Anonymous Inner class with more security and for still more security and less lines we use lambda expression
//class Alphaa implements FunctInte{
//
//	@Override
//	public void disp() {
//		System.out.println("..");
//	}
//
//}

public class FunctionalInterfaceInJava {

	public static void main(String[] args) {

		//Functional Interface : Interface with only one abstract method is called Functional Interface
		//This concept came from JAVA 8.0 /JDK 1.8 version

		//inner class
//	class Alphaa implements FunctInte{
//
//			@Override
//			public void disp() {
//				System.out.println("..");
//			}
//
//		}
//		Alphaa a = new Alphaa();
//		a.disp();

		// Anonymous Inner class which has no class keyword no class name and implements keyword
		//but has only Functional Interface name with brackets and object of that class with new keyword and reference to it and ; at end as below
		FunctInte d = new FunctInte(){
			@Override
			public void disp(int a) {
				System.out.println(a);
			}
		};
		d.disp(2);

		//through Lambda expression : () -> {}; // all of these are possible syntaxes
		//lambda expression is only possible for functional interface not for other interfaces
		FunctInte f = (int a) -> System.out.println(a);
		f.disp(2);

		FunctInte fu = (a) -> System.out.println(a);
		f.disp(2);

		FunctInte fun = a -> System.out.println(a);
		f.disp(2);
	}

}
```

---

11. INTERFACE VS CLASS:

- Interface cannot have constructor.
- Cannot create object of an interface.
- Interface is implemented using `implements` keyword.
- Class is instantiated using `new` keyword.

---
12. INTERFACE INHERITANCE:

- One interface can extend another using `extends`.
- A class can implement multiple interfaces.

Example:

```
interface A {
   void a();
}
interface B {
   void b();
}
interface C extends A, B {
   void c();
}
class Impl implements C {
   public void a() { }
   public void b() { }
   public void c() { }
}
```

---
13. COMMON INTERFACES IN JAVA:

- Comparable
- Comparator
- Runnable
- Cloneable
- Serializable
- Callable
- AutoCloseable

---
14. INTERVIEW THEORY QUESTIONS:
---
1. What is the purpose of an interface in Java?
2. Can we instantiate an interface?
3. How does interface support multiple inheritance?
4. What is the difference between abstract class and interface?
5. Can an interface have constructors or variables?
6. What is a functional interface?
7. What are default and static methods in interface?
8. What is a marker interface?
9. Why were private methods introduced in Java 9 interface?
10. Can interface extend a class?

---
15. MCQ QUESTIONS:
---
1) Which of the following is true about interface?
  a) Can have constructors
  b) Can be instantiated
  c) Can have static methods [✓]
  d) Can have protected methods

2) What will happen if class doesn't implement all methods of interface?
  a) Compilation error [✓]
  b) Runtime error

c) It works fine
d) Interface is ignored

3) What is the default modifier of methods in interface before Java 8?
a) private
b) public [✓]
c) default
d) protected

4) Which of these is not a type of interface?
a) Marker
b) Abstract [✓]
c) Functional
d) Normal

5) Can an interface extend multiple interfaces?
a) Yes [✓]
b) No
c) Only one
d) Only abstract ones

---
16. CODING QUESTIONS:

Q1. Create an interface Shape with area() method. Implement it in Circle and Rectangle classes.

```
interface Shape {
   double area();
}

class Circle implements Shape {
   double radius;
   Circle(double r) { radius = r; }
   public double area() {
      return 3.14 * radius * radius;
   }
}

class Rectangle implements Shape {
   double l, b;
   Rectangle(double l, double b) {
      this.l = l;
      this.b = b;
   }
   public double area() {
      return l * b;
   }
}

public class Main {
   public static void main(String[] args) {
      Shape s1 = new Circle(2);
      Shape s2 = new Rectangle(4, 5);
      System.out.println(s1.area());  // 12.56
      System.out.println(s2.area());  // 20.0
   }
}
```

---
17. TECHNICAL ROUND QUESTIONS (WITH EXPECTED ANSWERS):

Q1. Why are interface variables public, static, and final?
→ Because they are shared constants, and interfaces don't have instance-level data.

Q2. What happens if a class implements two interfaces with same method?
→ No conflict, the method needs to be implemented once in the class.

Q3. Can an interface implement another interface?
→ No. Interface can only **extend** another interface.

Q4. What is default method? Why introduced in Java 8?
→ Method with implementation in interface. To allow adding new features without breaking existing classes.

Q5. Can a class implement two interfaces with default methods of same name?
→ Yes, but it must override and resolve the conflict explicitly.

Q6. What is the use of private methods in Java 9 interface?
→ To reuse code inside default/static methods in the interface only.

Q7. Can you write lambda for interface with two methods?
→ No. Lambda works only with functional interfaces (1 abstract method).

JAVA 8 FEAUTURES:

JAVA 8 – LAMBDA EXPRESSIONS

==========

INTRODUCTION:

Lambda Expression is a new feature introduced in Java 8 that allows us to treat code as data.
It lets us create anonymous functions (i.e., functions without names) and pass them as parameters.

SYNTAX:

(parameter1, parameter2, ...) -> { function body }

If the body has only one statement, braces `{}` and return statement can be omitted:
a -> a * a

EXAMPLE:

Traditional Approach:

```
Runnable r = new Runnable() {
   public void run() {
      System.out.println("Thread running");
   }
};
```

With Lambda:

```
Runnable r = () -> System.out.println("Thread running");
```

KEY RULES:
---
1. Lambda can have 0 or more parameters.
2. If one parameter, parentheses `()` can be omitted: s -> s.length()
3. If multiple parameters, parentheses are required: (a, b) -> a + b

4. If only one statement, curly braces `{}` and return can be omitted.
5. Cannot define multiple methods inside lambda.
6. Lambda can be assigned only to a **functional interface**.

WHAT IS A FUNCTIONAL INTERFACE?

An interface with only ONE abstract method.
e.g., Runnable, Comparator, Callable, ActionListener, etc.

We can annotate it with `@FunctionalInterface` to enforce the rule.

ADVANTAGES OF LAMBDA:
---
1. Enables functional programming.
2. Reduces boilerplate code.
3. Improves readability.
4. Helps in writing cleaner code for streams, collections, threading, etc.
5. Encourages use of higher-order functions.

USING LAMBDA WITH COLLECTIONS:

```
List<String> names = Arrays.asList("Java", "C", "Python", "Kotlin");

names.forEach(name -> System.out.println(name));

names.sort((s1, s2) -> s1.compareToIgnoreCase(s2));

Thread t = new Thread(() -> System.out.println("Lambda Thread"));
t.start();
```

RETURNING VALUES FROM LAMBDA:

```
Function<Integer, Integer> square = n -> n * n;
System.out.println(square.apply(5)); // Output: 25
```

LAMBDA WITH CUSTOM FUNCTIONAL INTERFACE:

```
@FunctionalInterface
interface Calculator {
   int operate(int a, int b);
}

public class Main {
   public static void main(String[] args) {
      Calculator add = (a, b) -> a + b;
      Calculator multiply = (a, b) -> a * b;
      System.out.println(add.operate(5, 10));      // 15
      System.out.println(multiply.operate(5, 10)); // 50
   }
}
```

USAGE IN STREAMS:

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

list.stream()
   .filter(n -> n % 2 == 0)
   .map(n -> n * n)
   .forEach(System.out::println); // prints square of even numbers
```

INTERVIEW THEORY QUESTIONS (with answers)

Q1: What is a Lambda Expression in Java 8?
A: Lambda expression is a short block of code which takes in parameters and returns a value. It's used to implement functional interfaces concisely.

Q2: Can Lambda expressions be used without a functional interface?
A: No. A lambda must be assigned to a functional interface.

Q3: What is a functional interface?
A: An interface that has exactly one abstract method. It can have default and static methods.

Q4: What is the purpose of the `@FunctionalInterface` annotation?
A: It marks an interface as functional and throws a compile-time error if more than one abstract method is declared.

Q5: Can we use `this` keyword inside a lambda?
A: Yes. `this` refers to the enclosing class instance, not the lambda itself.

Q6: How do lambdas enable functional programming?
A: They allow functions to be passed as arguments and returned from methods, enabling higher-order functions.

Q7: How are lambdas better than anonymous classes?
A: They are shorter, more readable, and do not create extra classes at runtime.

Q8: What's the return type of a lambda expression?
A: Inferred from the functional interface's method signature.

INTERVIEW CODING QUESTIONS (with solutions)

Q1: Write a lambda to sort a list of strings in reverse order.

```
List<String> names = Arrays.asList("Java", "Python", "C");
names.sort((s1, s2) -> s2.compareTo(s1));
System.out.println(names);
```

Q2: Write a lambda to square numbers and print only even squares.

```
List<Integer> nums = Arrays.asList(1,2,3,4,5);
nums.stream()
   .map(n -> n * n)
   .filter(n -> n % 2 == 0)
   .forEach(System.out::println);
```

Q3: Write your own functional interface and use lambda.

```
@FunctionalInterface
interface Greet {
   void sayHello(String name);
}

public class Main {
   public static void main(String[] args) {
      Greet greet = name -> System.out.println("Hello " + name);
      greet.sayHello("Java");
   }
}
```

MULTIPLE CHOICE QUESTIONS (MCQs)
---

1. Which interface is required for a lambda expression to work?
a) Runnable
b) Any interface with >1 method

c) Functional interface
d) None of the above
Ans: c

2. What does `() -> System.out.println("Hello")` represent?
a) A method
b) A thread
c) A lambda expression
d) A stream
Ans: c

3. Which of the following is a correct lambda syntax?
a) int x -> x + 1
b) (int x) -> return x + 1;
c) x -> x + 1
d) x => x + 1
Ans: c

4. Can a lambda expression access outer class variables?
a) No
b) Yes, only static
c) Yes, if effectively final
d) Yes, always
Ans: c

5. Which Java version introduced Lambda?
a) Java 6
b) Java 7
c) Java 8
d) Java 11
Ans: c

JAVA 8 – STREAM AND STREAM API

==========

INTRODUCTION:

Stream API was introduced in Java 8 to process collections of data in a **functional style**.
It represents a **sequence of elements** supporting **sequential and parallel operations**.

IMPORTANT:

- Stream is **not a data structure**, but a **view of data**.
- Once consumed, a Stream **cannot be reused**.
- Stream operations are **lazy** — executed only when terminal operation is triggered.

PACKAGE:
---
java.util.stream

TYPES OF STREAMS:
---
1. **Sequential Stream** – processed in a single thread.
2. **Parallel Stream** – processed in multiple threads (internally uses ForkJoinPool).

CREATING STREAMS:

From Collection:
List<String> names = Arrays.asList("A", "B", "C");
Stream<String> stream = names.stream();

From Arrays:
```
int[] arr = {1, 2, 3};
IntStream stream = Arrays.stream(arr);
```

Using Stream.of():
```
Stream<String> stream = Stream.of("A", "B", "C");
```

INFINITE STREAMS:
```
Stream<Integer> infiniteStream = Stream.iterate(1, n -> n + 1);
```

STREAM PIPELINE:

A Stream Pipeline consists of:
1. **Source** – e.g., List, Set
2. **Intermediate Operations** – e.g., map, filter, sorted
3. **Terminal Operations** – e.g., collect, forEach, reduce

COMMON INTERMEDIATE OPERATIONS:
---
1. **map(Function)** – transforms each element.
2. **filter(Predicate)** – filters based on condition.
3. **sorted() / sorted(Comparator)** – sorts stream elements.
4. **distinct()** – removes duplicates.
5. **limit(n)** – limits to n elements.
6. **skip(n)** – skips n elements.

COMMON TERMINAL OPERATIONS:
---
1. **forEach(Consumer)** – performs action for each.
2. **collect(Collector)** – returns the result (List, Set, Map).
3. **toArray()** – converts stream to array.
4. **reduce()** – reduces to a single value.
5. **count()** – counts the number of elements.
6. **min()/max()** – returns min/max based on comparator.
7. **anyMatch()/allMatch()/noneMatch()** – boolean matchers.
8. **findFirst()/findAny()** – returns Optional.

USING STREAM API – EXAMPLES:
---
1. Print all even numbers from a list:
```
List<Integer> list = Arrays.asList(1,2,3,4,5);
list.stream()
   .filter(n -> n % 2 == 0)
   .forEach(System.out::println);
```

2. Square all numbers and collect into a list:
```
List<Integer> squared = list.stream()
   .map(n -> n * n)
   .collect(Collectors.toList());
```

3. Sum of all numbers using reduce:
```
int sum = list.stream().reduce(0, (a, b) -> a + b);
```

4. Find first even number:
```
Optional<Integer> firstEven = list.stream()
   .filter(n -> n % 2 == 0)
   .findFirst();
```

5. Get names starting with “A” and sort:
```
List<String> names = Arrays.asList("Ram", "Anil", "Amit");
List<String> result = names.stream()
   .filter(s -> s.startsWith("A"))
   .sorted()
```

```
   .collect(Collectors.toList());
```

6. Convert list to map:

```
List<String> items = Arrays.asList("apple", "banana");
Map<String, Integer> map = items.stream()
   .collect(Collectors.toMap(s -> s, s -> s.length()));
```

PARALLEL STREAM:

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
list.parallelStream()
   .forEach(System.out::println);
```

Be careful: order is **not guaranteed** in parallel streams.

STREAM OPERATIONS ARE LAZY:

Nothing executes until a terminal operation is called.

e.g.

```
Stream<Integer> stream = list.stream().map(n -> {
   System.out.println("Mapping: " + n);
   return n * n;
});
// No output here
stream.forEach(System.out::println); // Now mapping is done
```

LIMITATIONS OF STREAM:

---

1. Cannot reuse the same stream.
2. No short-circuiting for some operations.
3. Difficult debugging due to laziness.
4. Not always efficient with mutable/shared data.

INTERVIEW THEORY QUESTIONS (with answers)

Q1: What is the Java 8 Stream API?
A: Stream API allows us to process collections of objects in a declarative way using functional programming concepts.

Q2: Is Stream a data structure?
A: No. It is a pipeline for data transformation, not a data holder.

Q3: What are the types of Stream operations?
A: Intermediate (lazy) and Terminal (eager).

Q4: What's the difference between map and flatMap?
A: map transforms one element into another. flatMap flattens nested streams.

Q5: What is reduce() in Stream?
A: It combines elements of a stream into a single result using an accumulator function.

Q6: Can a Stream be reused?
A: No. Once a stream is consumed, it cannot be reused.

Q7: When should I use parallelStream()?
A: When your task is CPU-intensive and can benefit from multi-threading.

Q8: What’s the main benefit of using streams?
A: Code becomes more readable, concise, and easy to parallelize.

INTERVIEW CODING QUESTIONS (with solutions)

Q1: Find sum of squares of even numbers from list.

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
int sum = list.stream()
   .filter(n -> n % 2 == 0)
   .map(n -> n * n)
   .reduce(0, Integer::sum);
System.out.println(sum); // 20
```

Q2: Count names starting with "A".

```
List<String> names = Arrays.asList("Anil", "Bob", "Ajay", "Ravi");
long count = names.stream()
   .filter(s -> s.startsWith("A"))
   .count();
System.out.println(count); // 2
```

Q3: Convert a list of strings to uppercase and remove duplicates.

```
List<String> names = Arrays.asList("a", "b", "a", "c");
List<String> result = names.stream()
   .map(String::toUpperCase)
   .distinct()
   .collect(Collectors.toList());
```

Q4: Find the max element.

```
List<Integer> nums = Arrays.asList(3, 5, 1, 7, 2);
int max = nums.stream().max(Integer::compare).get();
```

MULTIPLE CHOICE QUESTIONS (MCQs)
---

1. Which of the following is a terminal operation?
a) map
b) filter
c) collect
d) sorted
Ans: c

2. What does `distinct()` do?
a) Sorts elements
b) Removes nulls
c) Removes duplicates
d) Skips first element
Ans: c

3. What type of interface is expected in a `map()` function?
a) Supplier
b) Function
c) Predicate
d) Consumer
Ans: b

4. Which stream method returns an Optional?
a) map
b) reduce
c) sorted
d) forEach
Ans: b

5. What happens if you reuse a Stream?
a) Nothing
b) It returns null
c) Throws IllegalStateException
d) Infinite loop
Ans: c

6. Which operation will short-circuit the stream?
a) filter
b) findFirst
c) map
d) collect
Ans: b

7. What does `collect(Collectors.toList())` return?
a) Stream
b) Set
c) List
d) Optional
Ans: c

==========
JAVA 8: PREDEFINED FUNCTIONAL INTERFACES
==========

Java 8 introduced several built-in Functional Interfaces in the
java.util.function package. Each interface has exactly one abstract method,
making them suitable for use with Lambda expressions and Stream API.

Functional Interface = Interface with one abstract method
(can also have static or default methods)

---
1. Predicate<T>

Definition:
Represents a boolean condition on a single argument.

Package:
java.util.function.Predicate

Abstract Method:
boolean test(T t)

Use Case:
- Filtering collections
- Used in stream filter()

Example:
Predicate<Integer> isEven = x -> x % 2 == 0;
System.out.println(isEven.test(4)); // true

Combining Predicates:
Predicate<String> startsWithA = str -> str.startsWith("A");
Predicate<String> endsWithZ = str -> str.endsWith("Z");
Predicate<String> both = startsWithA.and(endsWithZ);
System.out.println(both.test("AtoZ")); // true

---
2. Function<T, R>

Definition:
Represents a function that takes a value of type T and returns a value of type R.

Package:
java.util.function.Function

Abstract Method:
R apply(T t)

Use Case:
- Data transformation
- Mapping in streams

Example:
```
Function<String, Integer> strLength = s -> s.length();
System.out.println(strLength.apply("Java")); // 4
```

Chaining Functions:
```
Function<Integer, Integer> square = i -> i * i;
Function<Integer, Integer> add2 = i -> i + 2;
Function<Integer, Integer> result = square.andThen(add2);
System.out.println(result.apply(4)); // 18
```

---
3. Consumer<T>

Definition:
Represents an operation that takes one argument and returns no result.

Package:
java.util.function.Consumer

Abstract Method:
void accept(T t)

Use Case:
- Logging
- Performing side effects

Example:
```
Consumer<String> printer = msg -> System.out.println("Hello " + msg);
printer.accept("World"); // Hello World
```

---
4. Supplier<T>

Definition:
Represents a supplier of results. Takes no input, only returns a value.

Package:
java.util.function.Supplier

Abstract Method:
T get()

Use Case:
- Random value generation
- Lazy initialization

Example:
```
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

---
5. Comparator<T>

Definition:
Used to define custom comparison logic between two objects.

Package:
java.util.Comparator

Abstract Method:
int compare(T o1, T o2)

Use Case:
- Custom sorting

Example:
```
Comparator<Integer> reverse = (a, b) -> b - a;
List<Integer> list = Arrays.asList(4, 1, 6);
Collections.sort(list, reverse);
System.out.println(list); // [6, 4, 1]
```

Java 8 Example:
```
Comparator<Employee> byName = Comparator.comparing(Employee::getName);
```

---
6. Comparable<T>

Definition:
Defines the natural ordering of objects.
Must be implemented in the class itself.

Package:
java.lang.Comparable

Abstract Method:
int compareTo(T o)

Use Case:
- Default sorting in Collections.sort()

Example:
```
class Student implements Comparable<Student> {
   int marks;
   Student(int marks) {
      this.marks = marks;
   }
   public int compareTo(Student s) {
      return this.marks - s.marks;
   }
}
```

INTERVIEW QUESTIONS AND ANSWERS

Q1. What is the difference between Predicate and Function?
Ans: Predicate returns a boolean value; Function returns a value of any type.

Q2. Which functional interface is used to filter elements in a list?
Ans: Predicate

Q3. What is the difference between Comparator and Comparable?
Ans: Comparator defines external sorting logic, Comparable defines natural order within the class.

Q4. Can we chain predicates and functions?
Ans: Yes. Use and(), or(), negate() for Predicate andThen(), compose() for Function.

Q5. Which functional interface represents an action with no return value?
Ans: Consumer

Q6. Which interface returns a result without taking input?
Ans: Supplier

Q7. Why use Function<T, R>?
Ans: To transform data from one type to another, like mapping.

MCQ QUESTIONS WITH ANSWERS
---

1. What does Function<T, R> represent?
a. A function with no input
b. A function that returns boolean
c. A function with one input and one output [Answer: c]
d. None of the above

2. Which method is in the Supplier interface?
a. accept()
b. test()
c. get() [Answer: c]
d. apply()

3. Which interface is best for filtering?
a. Consumer
b. Supplier
c. Predicate [Answer: c]
d. Function

4. Which interface returns no result?
a. Predicate
b. Consumer [Answer: b]
c. Supplier
d. Function

5. Which interface takes no input but provides a value?
a. Predicate
b. Supplier [Answer: b]
c. Function
d. Consumer

IMPORTANT TIPS

- Predicate is used for conditions (boolean results).
- Function is used for data transformation (input to output).
- Consumer is for operations with side effects like printing.
- Supplier is for lazy generation or factory methods.
- Comparator and Comparable are for sorting, but used differently.
- Know the method signatures: test(), apply(), accept(), get(), compare(), compareTo()

==========

==========

JAVA 8: METHOD REFERENCES

==========

DEFINITION:
Method Reference is a shorthand syntax in Java 8 for calling a method using :: (double colon).
It is used to refer to a method without executing it.

SYNTAX:
ClassName::methodName
or
object::instanceMethodName

It is used with Functional Interfaces (like Predicate, Function, Consumer, etc.).

WHY USE METHOD REFERENCE?

- More readable and concise than lambda expressions
- Avoids boilerplate code
- Reuses existing methods

TYPES OF METHOD REFERENCES
---

1. Reference to a static method
   Syntax: ClassName::staticMethod

   Example:
```
public class Demo {
    public static void printMessage() {
        System.out.println("Hello from static method!");
    }

    public static void main(String[] args) {
        Runnable r = Demo::printMessage;
        new Thread(r).start();
    }
}
```

2. Reference to an instance method of a particular object
   Syntax: objectRef::instanceMethod

   Example:
```
public class Printer {
    public void print(String msg) {
        System.out.println(msg);
    }

    public static void main(String[] args) {
        Printer p = new Printer();
        Consumer<String> consumer = p::print;
        consumer.accept("Hello World");
    }
}
```

3. Reference to an instance method of an arbitrary object of a particular type
   Syntax: ClassName::instanceMethod

   Example:
```
List<String> list = Arrays.asList("apple", "banana", "cherry");
list.forEach(System.out::println); // Each string's println()
```

   Another Example:
```
Function<String, Integer> strLength = String::length;
```

```
System.out.println(strLength.apply("Java")); // 4
```

4. Reference to a constructor
   Syntax: ClassName::new

```
Example:
interface Message {
    Demo create(String msg);
}

class Demo {
    Demo(String msg) {
        System.out.println("Message: " + msg);
    }
}

public class Test {
    public static void main(String[] args) {
        Message m = Demo::new;
        m.create("Hello Constructor!");
    }
}
```

WHEN TO USE LAMBDA VS METHOD REFERENCE?

Use Method Reference when:
- The lambda is just calling a method (nothing extra)
- Code becomes shorter and cleaner

Otherwise, use lambda when:
- Additional logic is needed
- Parameters need transformation before method call

INTERVIEW QUESTIONS AND ANSWERS

Q1. What is method reference in Java 8?
Ans: It is a shorthand notation for calling methods using double colon (::), used with functional interfaces.

Q2. Can method reference replace all lambda expressions?
Ans: No. Only when the lambda is simply calling an existing method.

Q3. What are the different types of method references?
Ans:
- Static method reference
- Instance method of specific object
- Instance method of arbitrary object
- Constructor reference

Q4. What is the benefit of method references over lambda?
Ans: Improves readability and reduces code clutter when reusing methods.

Q5. Can method reference be used with custom functional interfaces?
Ans: Yes, as long as the method signature matches the abstract method in the interface.

Q6. What is the syntax for referring to a constructor?
Ans: ClassName::new

MCQ QUESTIONS WITH ANSWERS
---

1. Which operator is used for method reference?
a. ->
b. ::
c. %
d. **

Answer: b

2. What type of method reference is used in: String::length?
a. Static method reference
b. Constructor reference
c. Instance method of arbitrary object [Answer]
d. Instance method of specific object

3. Which one is not a valid method reference?
a. System.out::println
b. String::length
c. Object::equals
d. new::Demo

Answer: d

4. Which functional interface can you use with ClassName::new?
a. Runnable
b. Supplier
c. Constructor Interface
d. Any matching functional interface [Answer]

5. What will this return: Function<String, Integer> f = String::length;
a. Number of characters in a string [Answer]
b. ASCII value of first char
c. First index
d. Throws exception

CODING QUESTIONS

Q. Convert the following lambda to method reference:
Consumer<String> printer = s -> System.out.println(s);
Answer:
Consumer<String> printer = System.out::println;

Q. Replace lambda with method reference:
Function<String, Integer> lengthFunc = s -> s.length();
Answer:
Function<String, Integer> lengthFunc = String::length;

Q. Write a constructor reference using a custom class.
```
interface Factory {
   Sample create(String name);
}
class Sample {
   Sample(String name) {
      System.out.println("Name: " + name);
   }
}
Factory f = Sample::new;
f.create("Test");
```

REAL-WORLD USE CASES

- Sorting using Comparator:
  Arrays.sort(names, String::compareToIgnoreCase);

- forEach printing:
  list.forEach(System.out::println);

- Creating objects:
  Supplier<List<String>> listSupplier = ArrayList::new;

==

==
JAVA 8: OPTIONAL CLASS
==

INTRODUCTION:

`Optional<T>` is a container class introduced in Java 8 under `java.util` package.
It is used to represent optional values — that is, a value that **may or may not be present**.

MAIN PURPOSE:

To eliminate **NullPointerException (NPE)** in Java code and write cleaner, null-safe code.

==
WHY OPTIONAL?

Before Java 8:

```
String name = user.getName();
if (name != null) {
  System.out.println(name.toUpperCase());
}
```

After Java 8 using Optional:

```
Optional<String> name = Optional.ofNullable(user.getName());
name.ifPresent(n -> System.out.println(n.toUpperCase()));
```

==
CREATING OPTIONAL OBJECTS
---

1. Optional.of(value)
  --> Creates Optional with non-null value
  --> Throws NullPointerException if value is null

  Example:
  Optional<String> opt = Optional.of("Hello");

2. Optional.ofNullable(value)
  --> Creates Optional with value if non-null, else empty Optional

  Example:
  Optional<String> opt = Optional.ofNullable(null); // empty

3. Optional.empty()
  --> Returns an empty Optional

  Example:
  Optional<String> opt = Optional.empty();

==========
COMMON METHODS OF OPTIONAL
---

1. isPresent()
   --> Returns true if value is present, else false

   Example:
   ```
   if(opt.isPresent()) {
      System.out.println(opt.get());
   }
   ```

2. ifPresent(Consumer<T>)
   --> Executes lambda if value is present

   Example:
   ```
   opt.ifPresent(System.out::println);
   ```

3. get()
   --> Returns value if present, else throws NoSuchElementException

   Example:
   ```
   String s = opt.get();
   ```

4. orElse(T other)
   --> Returns value if present, else returns default

   Example:
   ```
   String s = opt.orElse("default");
   ```

5. orElseGet(Supplier)
   --> Similar to orElse(), but lazy evaluation using lambda

   Example:
   ```
   String s = opt.orElseGet(() -> "default");
   ```

6. orElseThrow()
   --> Throws NoSuchElementException if value not present

   Example:
   ```
   String s = opt.orElseThrow();
   ```

   Custom exception:
   ```
   opt.orElseThrow(() -> new IllegalArgumentException("Missing"));
   ```

7. map(Function)
   --> Transforms the value if present

   Example:
   ```
   Optional<Integer> len = opt.map(String::length);
   ```

8. flatMap(Function)
   --> Used when function returns another Optional

   Example:
   ```
   Optional<Optional<Integer>> nested = opt.map(s -> Optional.of(s.length()));
   Optional<Integer> flat = opt.flatMap(s -> Optional.of(s.length()));
   ```

9. filter(Predicate)
   --> Returns value if predicate is true, else empty

   Example:
   ```
   opt.filter(s -> s.startsWith("A")).ifPresent(System.out::println);
   ```

==========
EXAMPLES

```
Optional<String> name = Optional.of("Java");
System.out.println(name.get()); // Java

Optional<String> empty = Optional.empty();
System.out.println(empty.orElse("default")); // default

Optional<String> maybeNull = Optional.ofNullable(null);
System.out.println(maybeNull.isPresent()); // false
```

==========
BEST PRACTICES

✅ Use Optional for method return types that can be empty
✅ Avoid using Optional in fields, parameters (performance hit)
✅ Prefer orElseGet over orElse for expensive operations
✅ Never call get() without checking isPresent() — use orElse()

==========
INTERVIEW QUESTIONS & ANSWERS

Q1. What is Optional in Java 8?
Ans: Optional is a container class introduced to handle null values and avoid NullPointerException.

Q2. What is the difference between of() and ofNullable()?
Ans:
- of(): throws NullPointerException if value is null
- ofNullable(): returns empty Optional if value is null

Q3. How does Optional help avoid null checks?
Ans: Optional provides methods like orElse(), ifPresent(), and map() to handle values without explicitly checking for null.

Q4. When should you NOT use Optional?
Ans: Avoid using Optional as method parameters or class fields. Use it only for return values.

Q5. What is the difference between orElse() and orElseGet()?
Ans:
- orElse(): Always evaluates the argument
- orElseGet(): Lazily evaluates only when needed

Q6. What happens if you call get() on empty Optional?
Ans: Throws java.util.NoSuchElementException

Q7. How is flatMap different from map in Optional?
Ans:
- map(): returns Optional<Optional<T>> if mapper returns Optional
- flatMap(): flattens result into Optional<T>

==========
MCQ QUESTIONS WITH ANSWERS
---

1. Which method creates an empty Optional?
a. Optional.null()
b. Optional.of(null)
c. Optional.empty() [Answer]
d. Optional.clear()

2. What does orElse() return?
a. Always null
b. Optional object
c. Default value if empty [Answer]
d. Nothing

3. What will Optional.of(null) return?
a. Empty Optional
b. Throws Exception [Answer]
c. null
d. Optional["null"]

4. What is the output?

```
Optional<String> opt = Optional.ofNullable(null);
System.out.println(opt.isPresent());
```

a. true
b. false [Answer]
c. null
d. error

5. What does flatMap() return?
a. Optional of Optional
b. Optional<T> [Answer]
c. List<T>
d. Stream<T>

==========
CODING QUESTIONS

Q1. Convert the below null-safe logic to Use Optional:

```
if(user != null && user.getName() != null) {
   System.out.println(user.getName().toUpperCase());
}
```

Ans:

```
Optional.ofNullable(user)
        .map(User::getName)
        .map(String::toUpperCase)
        .ifPresent(System.out::println);
```

Q2. Get length of a string using Optional

```
Optional<String> name = Optional.of("Test");
int length = name.map(String::length).orElse(0);
System.out.println(length); // 4
```

Q3. Create Optional from null and handle it

```
String input = null;
Optional<String> opt = Optional.ofNullable(input);
System.out.println(opt.orElse("Default")); // Default
```

Q4. Throw custom exception using orElseThrow

```
Optional<String> data = Optional.empty();
String result = data.orElseThrow(() -> new IllegalArgumentException("No Data"));
```

==========
REAL-WORLD USE CASE

```
public Optional<User> findUserByEmail(String email) {
   return users.stream()
           .filter(u -> u.getEmail().equals(email))
```

```
            .findFirst();  // returns Optional<User>
}

Optional<User> user = findUserByEmail("abc@example.com");
user.ifPresent(u -> sendEmail(u));

==========================================


==========================================
JAVA 8: COLLECTORS (java.util.stream.Collectors)
==========================================

INTRODUCTION:
-------------
Collectors is a utility class in Java 8 found in:
   --> java.util.stream.Collectors

It provides **static methods** to collect stream elements into various **mutable containers**, such as:
- List
- Set
- Map
- String
- Grouped or partitioned collections

Collectors are typically used in the **collect()** terminal operation of the Stream API.

==========================================
SYNTAX:
--------
<T, A, R> R collect(Collector<? super T, A, R> collector)

Example:
List<String> names = stream.collect(Collectors.toList());

==========================================
MOST COMMON COLLECTORS
------------------------------------------

1. Collectors.toList()
-----------------------
Collects elements into a List

List<String> names = stream.collect(Collectors.toList());

2. Collectors.toSet()
----------------------
Collects elements into a Set (removes duplicates)

Set<String> names = stream.collect(Collectors.toSet());

3. Collectors.toMap()
----------------------
Collects elements into a Map using key and value mappers

Map<Integer, String> map =
   list.stream().collect(Collectors.toMap(
      item -> item.getId(),
      item -> item.getName()
   ));

4. Collectors.joining()
------------------------
```

Concatenates string values with optional delimiter, prefix, suffix

```
String result = list.stream().collect(Collectors.joining(", "));
String fancy = list.stream().collect(Collectors.joining(", ", "[", "]"));
```

5. Collectors.counting()

Counts the number of elements in the stream

```
long count = stream.collect(Collectors.counting());
```

6. Collectors.summingInt(), summingDouble(), summingLong()

Returns the sum of numeric properties

```
int sum = items.stream().collect(Collectors.summingInt(Item::getPrice));
```

7. Collectors.averagingInt(), averagingDouble(), averagingLong()

Returns the average of numeric properties

```
double avg = items.stream().collect(Collectors.averagingInt(Item::getQuantity));
```

8. Collectors.maxBy() / minBy()

Finds max or min using Comparator

```
Optional<Item> max = list.stream().collect(Collectors.maxBy(Comparator.comparing(Item::getPrice)));
```

9. Collectors.groupingBy()

Groups elements by classifier function

```
Map<String, List<Employee>> map =
   list.stream().collect(Collectors.groupingBy(Employee::getDepartment));
```

You can also use downstream collectors:

```
Map<String, Long> countByDept =
   list.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));
```

10. Collectors.partitioningBy()

Partitions into two groups: true or false

```
Map<Boolean, List<Employee>> partitioned =
   list.stream().collect(Collectors.partitioningBy(emp -> emp.getSalary() > 10000));
```

==========
REAL-WORLD EXAMPLES
---

1. Convert List of strings to comma-separated string:

```
  String result = list.stream().collect(Collectors.joining(", "));
```

2. Convert List of students into Map<rollNo, name>

```
  Map<Integer, String> map = students.stream()
     .collect(Collectors.toMap(Student::getRollNo, Student::getName));
```

3. Group students by department

```
  Map<String, List<Student>> grouped = students.stream()
     .collect(Collectors.groupingBy(Student::getDepartment));
```

4. Count students in each department

```
  Map<String, Long> count = students.stream()
```

```
    .collect(Collectors.groupingBy(Student::getDepartment, Collectors.counting()));

5. Get average marks per department
  Map<String, Double> avg = students.stream()
    .collect(Collectors.groupingBy(Student::getDepartment, Collectors.averagingDouble(Student::getMarks)));

6. Partition students by pass/fail (marks > 40)
  Map<Boolean, List<Student>> passFail = students.stream()
    .collect(Collectors.partitioningBy(s -> s.getMarks() > 40));
```

==========

BEST PRACTICES

✅ Use collect(Collectors.toList()) instead of manual loops
✅ Use groupingBy() for multi-level aggregation
✅ Use joining() for clean string concatenation
✅ Avoid duplicate keys in toMap() or use merge function

==========

INTERVIEW QUESTIONS & ANSWERS

Q1. What is the use of Collectors in Java 8?
Ans: It provides factory methods to create collectors for gathering Stream results into collections, strings, maps, and summaries.

Q2. Difference between groupingBy() and partitioningBy()?
Ans:
- groupingBy(): groups elements by a classifier
- partitioningBy(): splits elements into two groups based on predicate (true/false)

Q3. How do you count elements using Collectors?
Ans: Use `Collectors.counting()` with `collect()`

Q4. How to create Map from List using Streams?
Ans:
Map<K, V> map = list.stream().collect(Collectors.toMap(obj -> key, obj -> value));

Q5. What will happen if keys are duplicated in toMap()?
Ans: Throws IllegalStateException unless a merge function is provided.

Q6. How to calculate average using Streams?
Ans: Use Collectors.averagingInt(), averagingDouble(), or averagingLong()

Q7. What is downstream collector?
Ans: A collector applied after a primary collector, like:
  groupingBy(classifier, downstreamCollector)

Q8. Can you write a custom collector?
Ans: Yes, using Collector.of() but it's advanced. Most use built-in ones.

==========

MCQ QUESTIONS WITH ANSWERS

---

1. Which method collects stream into a List?
a. collect(Collectors.toCollection())
b. collect(Collectors.toList()) [Answer]
c. collect(toList())
d. collect(Stream.toList())

2. What does groupingBy() return?

a. List<List<T>>
b. Set<T>
c. Map<K, List<T>> [Answer]
d. Optional<T>

3. What is the result of collect(Collectors.counting())?
a. int
b. long [Answer]
c. Optional<Integer>
d. Stream<Long>

4. What will joining() return?
a. String [Answer]
b. List
c. Stream
d. Set

5. Which method is used for partitioning?
a. groupingBy()
b. filteringBy()
c. partitioningBy() [Answer]
d. dividingBy()

6. Which collector can calculate average salary?
a. Collectors.sum()
b. Collectors.averagingDouble() [Answer]
c. Collectors.average()
d. Collectors.mean()

7. What is needed for toMap() collector?
a. Only key extractor
b. Only value extractor
c. Both key and value extractors [Answer]
d. No arguments

8. Which of the following can cause exception in toMap()?
a. Null keys
b. Duplicate keys [Answer]
c. Empty stream
d. All of the above

JAVA EXCEPTION HANDLING

==========

HLL -> Java compiler - **Compilation time - Errors that occur here are called Compilation/Syntax error**
-> Byte code -> JVM -> MLL -> Microprocessor - Output - Execution/run time - Problems during execution is called as Exceptions

Run time errors occur during runtime and occurs due to lack of system resources. - eg: StackOveflowError, OutOfMemoryError

```
	static void disp() {
		disp(); //function calling itself is called as recursion.
//StackOverflowError will be occurred which is runtime error, since there is lack of resources, i.e the stack frames for disp will be keep repeatedly created in stack segment and once stack memory is filled and there will be overflow.
	}
```

```
int[] arr = new int[Integer.MAX_VALUE]; //OutOfMemoryError, since that much heap memory is not available.
```

```
//Can we handle the run time errors - stack over flow error? Yes we can handle runtime errors using try and catch block as below

	static void disp1() {
		try {
			disp1();
		}
		catch(StackOverflowError e) {
			System.out.println("Handled run time error using try and catch");
			System.out.println(e.getMessage());
		}
	}

catch(Error e) { //same as Exception Error is also a generic inbuilt Error class in java
			System.out.println("handled");
		}
```

❖ WHAT IS AN EXCEPTION?

=> An Exception is an abnormal condition or event that disrupts the normal flow of a program while execution when user gives faulty input.
->Java uses an object-oriented approach to handle exceptions using the Throwable class hierarchy.

- **Run time system(RTS)** - catches the exception object and searches for try catch blocks
- **Default Exception Handler(DEH)** - When RTS doesn't see any try catch block it will give by itself in console what exception it is
- **User defined Exception Handling(UDEH)** - When there is try catch block its called User defines Exception handling.

//Here a Arithmetic Exception object is created which is a instance of inbuilt-class ArithmeticException from java.lang package which is caught by the catch block with reference a of type ArithmeticException a = (New ArithmeticException();(invisible) - for all exception the same happens.
//ArithmetcException Object is caught and passed to the reference of the type inbuilt class ArithmeticException

❖ DIFFERENCE BETWEEN ERROR AND EXCEPTION:

Error:
- Serious issues, cannot be recovered.
- Handled by JVM.
- Examples: StackOverflowError, OutOfMemoryError

Exception:
- Can be recovered.
- Handled by user code.
- Examples: NullPointerException, IOException

❖ TYPES OF EXCEPTIONS:

---

1. Checked Exceptions:- (Compile time exception)
 - When : Compile time
 - Why : called method is ducking an exception.
  - Known to the compiler at compile time.
  - Must be handled or declared using throws.
  - Example: IOException, SQLException, FileNotFoundException, InterruptedException, ClassNotFoundException

```
for(int i=65; i<=69; i++) {
			System.out.println((char)i); //prints A B C D E

			//This below will cause checked exception
			//Thread.sleep(3000); //Class Thread has static sleep ( long milliseconds) method which throws exception , or which is ducking an exception.

			try {

				Thread.sleep(3000); //Class Thread has static sleep ( long milliseconds)
```

```
method which throws exception , or which is ducking an exception.
                    }
                    catch(Exception e) {
                           System.out.println(e.getMessage());
                    }
             }
```

2. Unchecked Exceptions: (Run time exception)
 - When : Run time
 - Why : occurs because of faulty input.
 - Occur at runtime.
 - Compiler doesn't force handling.
 - Example: ArithmeticException, NullPointerException, InputMismatchException

```
Scanner sc= new Scanner(System.in);
             System.out.println("Enter the a");
             System.out.println(sc.nextInt()/sc.nextInt()); //lets say we give 0 in denominator which
will be given during run time, which will cause run time/unchecked exception
```

3. Errors:
 - Irrecoverable.
 - Handled by JVM.
 - Example: OutOfMemoryError

Different ways of Handling Exception:

1. Handle the exception using try, catch blocks
2. Rethrowing an exception try, catch, throw, throws, finally
3. Ducking of an exception - throws……(try and catch are optional) - Ducking is where a method will escape from handling exception by just warning using throws Exception is called ducking.

Different Ways of Handling Exception in Java

1. **Handling Exception using try-catch blocks**
→ This is the most common and direct way to handle exceptions.

Syntax:

```
try {
   // risky code
} catch (ExceptionType e) {
   // handling code
}
```

Example:

```
public class TryCatchExample {
   public static void main(String[] args) {
      try {
         int a = 10, b = 0;
         int c = a / b;  // ArithmeticException
         System.out.println("Result: " + c);
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }

      System.out.println("Program continues...");
   }
}
```

```java
package exceptionHandling;
import java.util.*;

class E11{

	void fun1() {
		System.out.println("Connection established");
		try {
			Scanner sc = new Scanner(System.in);
			System.out.println("Enter the numerator: ");
			int a = sc.nextInt();

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

			int c = a/b; //from here only the control goes to fun2 hence there will be no connection terminated4
			System.out.println(c);

		}
		catch(Exception e) {
			System.out.println(e.getMessage());
		}
		System.out.println("Connection terminated2");
	}
}


public class DiffWaysOfHandlingException1 {

	public static void main(String[] args) {

		//First RTS goes to fun1 in class E11 then checks for try catch in main method.
		System.out.println("Connection established");
		E11 e = new E11();
		e.fun1();
		System.out.println("Connection Terminated1");
	}

}
```

---

2. Rethrowing an Exception using try-catch-throw or throws

→ This means catching an exception and throwing it again (to be handled elsewhere).

Example:

```java
public class RethrowExample {
  static void divide(int a, int b) throws ArithmeticException {
    try {
      int result = a / b;
      System.out.println("Result: " + result);
    } catch (ArithmeticException e) {
      System.out.println("Exception caught in divide(), rethrowing...");
      throw e;  // Rethrown to caller
    }
  }

  public static void main(String[] args) {
    try {
      divide(10, 0); // Call method that rethrows
    } catch (ArithmeticException e) {
      System.out.println("Handled again in main(): " + e.getMessage());
    }
  }
}
```

```java
package exceptionHandling;
import java.util.*;

class E21{

	void fun1() throws Exception{ //this warns the main method that i am re-throwing the exception handle it
		System.out.println("Connection established");
		try {
			Scanner sc = new Scanner(System.in);
			System.out.println("Enter the numerator: ");
			int a = sc.nextInt();

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

			int c = a/b; //from here only the control goes to fun2 hence there will be no connection terminated4
			System.out.println(c);

		}
		catch(Exception e) { //caught the object with exception
			System.out.println(e.getMessage());
			throw e; //then this is re-throwing the object of exception to main stack frame where this fun1 was called and this main method is unaware of it so it exception will not be handled , so this method should give the warning when fun1 is declared using throws as above
		}
		//System.out.println("Connection terminated2"); //not executed as the exception object is thrown before the control comes to this line, so to make it execute no matter what we use finally block with finally keyword
		finally{
			System.out.println("Connection terminated2");
		}
	}
}


public class DiffWaysOfHandlingException2Rethrowing {

	public static void main(String[] args) {

		//First RTS goes to fun1 in class E11 then checks for try catch in main method.
		System.out.println("Connection established");
		E21 e = new E21();
		try { //handling the re-thrown exception
			e.fun1();
		}catch(Exception ex) {
			System.out.println(ex.getMessage()+ " Handled");
		}
		System.out.println("Connection Terminated1");
	}

}
```

---

3. Ducking an Exception using throws keyword (Try-Catch is optional)

→ Ducking = Declaring that a method might throw an exception using throws, and not handling it immediately.

→ The caller must handle or further declare the exception.

Example (Checked Exception):

```
import java.io.*;

public class DuckingExample {
   static void readFile() throws IOException {
      FileReader fr = new FileReader("nonexistent.txt");  // may throw FileNotFoundException
      fr.read();
```

```
    fr.close();
  }

  public static void main(String[] args) {
    try {
      readFile();  // Caller handles it here
    } catch (IOException e) {
      System.out.println("Exception handled in main: " + e);
    }
  }
}
```

```
package exceptionHandling;
import java.util.Scanner;

class Demo1{
	void alpha() throws Exception{ //escaped exception by just warning there will be exception
but not handling it - this is called ducking
		System.out.println("Connection established.");

		Scanner sc = new Scanner(System.in);

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

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

		int c = a/b;
		System.out.println(c);

		System.out.println("Connection terminated");
		sc.close();
	}
}

public class DiffWaysHandle3DuckingException {

	public static void main(String[] args)  { //now main method is also escaping if here also
there is throws Exception, hence Default Exception handler will come in picture

		//Ducking : its escaping of the exception

		Demo1 d = new Demo1();
		try {
			d.alpha();
		}
		catch(Exception e) {
			System.out.println(e.getMessage());
		}

	}

}
```

---

4. Using finally block with try-catch
→ finally block is used to execute important code (like closing files, releasing resources) regardless of exception.

Example:

```
public class FinallyExample {
  public static void main(String[] args) {
    try {
      int a = 5, b = 0;
      int c = a / b;
```

```
            System.out.println("Result: " + c);
        } catch (ArithmeticException e) {
            System.out.println("Caught Exception: " + e);
        } finally {
            System.out.println("Finally block executed!");
        }

        System.out.println("Program continues...");
    }
}
```

Summary Table:

Way	Use Case	try/catch required?	Example Exception
try-catch	Direct handling in same method	Yes	ArithmeticException
try-catch-throw	Handle partially, rethrow to caller	Yes	ArithmeticException
throws (Ducking)	Pass exception to caller for handling	No	IOException
finally	Clean-up actions (always runs)	Usually with try	Any

❖ EXCEPTION HIERARCHY:

---

java.lang.Object
↳ java.lang.Throwable
  ↳ java.lang.Error - Has child classes like StackOverflowError, OutOfMemoryError
  ↳ java.lang.Exception
    ↳ Checked Exception **(IOException)**class name
    ↳ Unchecked Exception **(RuntimeException)**class name

IN-BUILT specialized METHODS IN THROWABLE: which will be inherited to other child classes down like Exception, RuntimeException , ArithmeticException

1. getMessage() - returns the string that says why exception has happened - Description of exception - e.getMessage()
2. printStackTrace() - void method - gives in which line exception, what exception did occur - e.printStackTrace()

```
            try {
                int c = 100/0;
            }
            catch(Exception e) { //throwable has these inbuilt specialized methods which will be inherited by its child classes like exception , error..
                System.out.println(e.getMessage());
                e.printStackTrace();
                System.out.println(e.getLocalizedMessage()); //same as e.getMessage();

                System.out.println(e); //Doesnt give address of e, it returns value i.e Exception name with description using toString() method.

                //getMessage() - gives description
                //printStackTrace - gives type of exception + description + which line
                //reference - gives type + description
            }
```

❖ KEYWORDS USED IN EXCEPTION HANDLING:

---

try - Block where risky code is written
catch - Handles the exception
finally - Executes always, cleanup block

throw   - Used to explicitly throw an exception
throws   - Declares exception in method signature

❖ BASIC SYNTAX:

```
try {
  // risky code
} catch (ExceptionType e) {
  // handle it
} finally {
  // cleanup
}
```

❖ EXAMPLES:

```
package exceptionHandling;
import java.util.*;

public class ExceptionIntro {

	public static void main(String[] args) {

		//Exceptions occur during the execution time and compilation/syntax error occur during compiling , that is as we write the code as Java compiles as we write the code.
		//An Exception is an abnormal condition or event that disrupts the normal flow of a program when user gives faulty input.

//		System.out.println(10/3); //3

		//If we give denominator as 0 it raises exception during Execution time. - ArithmeticException
		//When there is try catch block it means there is User defined Exception Handling
		System.out.println("Connection Established");
		try {
			Scanner sc = new Scanner(System.in);
			System.out.println("Enter the numerator: ");
			int a = sc.nextInt();

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

			int c = a/b;
			System.out.println(c);
		}
		catch(Exception e){ //executes only when there is exception.
			System.out.println("Enter no zero denominator");
			System.out.println(e.getMessage());
		}

		System.out.println("Connection terminated");



	}

}
```

```
// Example 1: Arithmetic Exception
try {
  int x = 10 / 0;
} catch (ArithmeticException e) {
  System.out.println("Division by zero!");
}
```

```
// Example 2: ArrayIndexOutOfBoundsException
try {
  int[] arr = new int[2];
  arr[3] = 5;
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("Invalid index!");
}
```

❖ MULTIPLE CATCH BLOCKS:

```
-------------------------
try {
  // code
} catch (IOException e) {
  // handle IO
} catch (Exception e) {
  // generic catch
}

package exceptionHandling;

import java.util.InputMismatchException;
import java.util.Scanner;
import java.util.*;

public class SingleTryMultiCatch2 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);

		System.out.println("Connection established.");

		try {
			System.out.println("Enter the numerator: ");
			int a = sc.nextInt();

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

			int c = a/b; //Here a Arithmetic Exception object is created which is a instance of inbuilt-class ArithmeticException from java.lang package which is caught by the catch block with reference a of type ArithmeticException a = (New ArithmeticException();(invisible) - for all exception the same happens.

			System.out.println(c);

			System.out.print("Enter the size of the array: ");
			int size = sc.nextInt(); //-3 - NegativeArraySizeException
			int [] arr = new int[size];

			System.out.println("Enter the data: ");
			int data = sc.nextInt(); //Varsha - InputMismatchException

			System.out.println("Enter the index where data shouldbe inserted : ");
			  int index = sc.nextInt();

			  arr = null; //- NullPointerException

			  arr[index] = data; //Lets say we give index 7 which is out of bound - ArrayIndexOutOfBoundsException
			  System.out.println(arr[index]);

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

```
            }
            //if we give generic exception as the first catch block then others it will lead to error since other Exceptions will
            become unreachable code.
//                  catch(Exception e) { //generic exception
//                          System.out.println(e.getMessage());
//                  }
            catch(ArithmeticException a){ //ArithmetcException Object is caught and passed to the reference of the type inbuilt class
            ArithmeticException
                    System.out.println("Give non zero denominator");
            }
            catch(NegativeArraySizeException n) {
                    System.out.println("Be positive");
            }
            catch(InputMismatchException i) {
                    System.out.println("Give integer input.");
            }
            catch(ArrayIndexOutOfBoundsException a) {
                    System.out.println(a.getMessage());
                    System.out.println("Be in length");
            }
            catch(Exception e) { //generic exception
                    System.out.println(e.getMessage());
            } //if we give generic exception as the first catch block then others it will lead to error since other Exceptions will
            become unreachable code.

            System.out.println("Connection is terminated");
        }

}
```

Note: Always write child exceptions first, then parent.

❖ **FINALLY BLOCK:**

- Always executes regardless of exception.
- Used for clean up (closing DB, files).
- Executes even if return exists in try/catch.

-try{}finally{} - possible, catch(){}finally{} - not possible

---

Difference between final, finally, finalize().

1. Final is a keyword which can be given to class , variable and method, class.
2. Finally is a block to make sure it is executed regardless of exception
3. Finalize() is a method in Object class which is called by Garbage collector when there is a garbage to be collected/destroyed/de allocate.

❖ **throw vs throws:**

---

throw:

- Used to throw an exception explicitly.
- Only one object can be thrown at a time.

throws:

- Declares exception in method signature.

Example:

```
void read() throws IOException {
  throw new IOException("File not found");
}
```

❖ CUSTOM EXCEPTION:

WHAT IS A CUSTOM EXCEPTION?

-A Custom Exception is a user-defined exception class in Java.
-It is used when the built-in exceptions do not adequately describe a specific application error.
-Java allows developers to create their own exception types by extending the `Exception` class (for checked exceptions) or `RuntimeException` class (for unchecked exceptions).

WHY USE CUSTOM EXCEPTIONS?

---

1. To provide more meaningful error messages.
2. To represent application-specific error conditions.
3. To improve readability and maintainability of code.
4. To enforce business logic through exception handling.

SYNTAX: CREATING A CUSTOM EXCEPTION

```
// Checked Exception
public class MyCheckedException extends Exception {
   public MyCheckedException(String message) {
      super(message);
   }
}

// Unchecked Exception
public class MyUncheckedException extends RuntimeException {
   public MyUncheckedException(String message) {
      super(message);
   }
}
```

EXAMPLE: CUSTOM CHECKED EXCEPTION

```
package exceptionHandling; // declaring the package

import java.util.Scanner; // importing Scanner class for input

// Custom exception class extending Exception
class InvalidInputException extends Exception{ // user-defined exception, no body needed for simple use
}

class Atm{
	int pin = 1234; // correct PIN known by the ATM
	int pCustomer; // PIN entered by the customer

	void acceptInput() {
		Scanner sc = new Scanner(System.in); // creating Scanner object
		System.out.print("Enter the PIN: "); // prompt to enter PIN
		pCustomer = sc.nextInt(); // storing entered PIN
	}

	void validate() throws Exception{ // this method may throw an exception
		if(pin==pCustomer) { // if entered pin matches ATM pin
			System.out.println("Collect your money"); // successful login
		}
		else { // if pin is incorrect
			System.out.println("Invalid pin"); // message to user
			InvalidInputException iie = new InvalidInputException(); // create object of custom
exception
			throw iie; // throw the exception to caller
```

```
		}
	}
}

class Bank {
	void init(){ // method to handle 3 attempts
		Atm a = new Atm(); // create object of ATM

		try { // first attempt
			a.acceptInput(); // take PIN input
			a.validate(); // validate it
		}
		catch(Exception e) { // if first attempt fails
			System.out.println("Re-enter the pin"); // prompt again
			try { // second attempt
				a.acceptInput(); // take input again
				a.validate(); // validate again
			}catch(Exception b) { // if second attempt fails
				try { // third attempt
					a.acceptInput(); // take input third time
					a.validate(); // validate third time
				}catch(Exception c) { // if third attempt fails
					System.out.println("CARD IS BLOCKED!!!!!"); // block the card after 3
failed attempts
				}
			}
		}
	}
}

public class CustomExceptionInJava {

	public static void main(String[] args) {

		// Custom Exception in Java:
		// => A Custom Exception is a user-defined exception class in Java.

		Bank b = new Bank(); // create Bank object
		b.init(); // call init to start ATM PIN process
	}

}
```

```
class InvalidAgeException extends Exception {
  public InvalidAgeException(String message) {
    super(message);
  }
}

public class TestCustomException {
  static void validate(int age) throws InvalidAgeException {
    if (age < 18) {
      throw new InvalidAgeException("Age is below 18. Not eligible to vote.");
    } else {
      System.out.println("Eligible to vote");
    }
  }

  public static void main(String[] args) {
    try {
      validate(15);
    } catch (InvalidAgeException e) {
      System.out.println("Exception caught: " + e.getMessage());
    }
  }
}
```

OUTPUT:
Exception caught: Age is below 18. Not eligible to vote.

EXAMPLE: CUSTOM UNCHECKED EXCEPTION

```
class DivisionByZeroException extends RuntimeException {
   public DivisionByZeroException(String message) {
      super(message);
   }
}

public class Calculator {
   public static int divide(int a, int b) {
      if (b == 0) {
         throw new DivisionByZeroException("Cannot divide by zero.");
      }
      return a / b;
   }

   public static void main(String[] args) {
      int result = divide(10, 0);
      System.out.println("Result: " + result);
   }
}
```

OUTPUT:
Exception in thread "main" DivisionByZeroException: Cannot divide by zero.

DIFFERENCE: CHECKED vs UNCHECKED CUSTOM EXCEPTIONS

Checked Custom Exception:
- Inherits from `Exception`
- Must be handled using try-catch or declared with throws
- Compile-time checked

Unchecked Custom Exception:
- Inherits from `RuntimeException`
- Optional handling
- Checked at runtime

BEST PRACTICES FOR CUSTOM EXCEPTIONS
---
1. Name your exception clearly (e.g., `InvalidUserInputException`)
2. Extend correct class (`Exception` or `RuntimeException`)
3. Always provide a meaningful error message
4. Optionally, provide multiple constructors (default, with message, with cause)
5. Document the usage of the custom exception

INTERVIEW CONCEPTS TO REMEMBER

✓ What is a custom exception in Java?
✓ Why and when should you create a custom exception?
✓ Difference between checked and unchecked custom exceptions?
✓ Can you create a custom exception with multiple constructors?
✓ How does custom exception improve code quality?
✓ Example scenario where custom exception is useful?

MCQ QUESTIONS

---

1. What is the base class for all exceptions in Java?
   a) Error
   b) Throwable
   c) Exception
   d) RuntimeException
   ➤ Answer: b) Throwable

2. Which class should be extended to create a checked custom exception?
   a) Throwable
   b) Exception
   c) RuntimeException
   d) Object
   ➤ Answer: b) Exception

3. Which exception is not required to be handled at compile time?
   a) IOException
   b) SQLException
   c) NullPointerException
   d) FileNotFoundException
   ➤ Answer: c) NullPointerException

4. Can custom exceptions have multiple constructors?
   a) Yes
   b) No
   ➤ Answer: a) Yes

5. What happens if you do not handle a checked custom exception?
   a) Compile-time error
   b) Runtime error
   c) Logical error
   d) No error
   ➤ Answer: a) Compile-time error

CODING QUESTION

Q: Create a custom exception named `NegativeAmountException` that should be thrown
when a user tries to withdraw a negative amount from their bank account.

ANSWER:

```
class NegativeAmountException extends Exception {
    public NegativeAmountException(String message) {
        super(message);
    }
}

class Bank {
    void withdraw(int amount) throws NegativeAmountException {
        if (amount < 0) {
            throw new NegativeAmountException("Negative amount withdrawal not allowed.");
        } else {
            System.out.println("Withdrawn: " + amount);
        }
    }

    public static void main(String[] args) {
        Bank b = new Bank();
        try {
            b.withdraw(-500);
        } catch (NegativeAmountException e) {
            System.out.println("Exception caught: " + e.getMessage());
```

```
    }
  }
}
```

OUTPUT:
Exception caught: Negative amount withdrawal not allowed.

==========

❖ **EXCEPTION PROPAGATION:**

- If methodA() calls methodB() and B throws exception, it propagates back to A.
- If not handled, goes to JVM.

```
package exceptionHandling;
import java.util.*;

class E1{

	void fun1() {
		System.out.println("Connection established");
//		try {
			Scanner sc = new Scanner(System.in);
			System.out.println("Enter the numerator: ");
			int a = sc.nextInt();

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

			int c = a/b; //from here only the control goes to fun2 hence there will be no connection terminated4
			System.out.println(c);
//		}
//		catch(Exception e) {
//			System.out.println(e.getMessage());
//		}
		System.out.println("Connection terminated4");
	}
}

class E2{

	void fun2() {
		System.out.println("Connection established");
		try {
			E1 e = new E1();
			e.fun1();
		}
		catch(Exception e) {
			System.out.println(e.getMessage());
		}
		System.out.println("Connection terminated3  ");
	}
}

class E3{

	void fun3() {
		System.out.println("Connection established");
		E2 e = new E2();
		e.fun2();
		System.out.println("Connection terminated2");
	}
}

public class ExceptionPropogation {

	public static void main(String[] args) {

		//Stack tracing happens and RTS checks for tryCatcj in fun1 then fun2 then fun3 then main
```

```
method, if even main method doesnt have the UDEH it gives to UDH

            System.out.println("Connection established");
            E3 e = new E3();
            e.fun3();
            System.out.println("Connection Terminated1");
        }

}
```

❖ NESTED try-catch:

```
--------------------
try {
  try {
    int a = 10 / 0;
  } catch (ArithmeticException e) {
    System.out.println("Inner catch");
  }
} catch (Exception e) {
  System.out.println("Outer catch");
}
```

❖ MULTI-CATCH BLOCK (Java 7+):

```
--------------------------------
try {
  // risky code
} catch (IOException | SQLException e) {
  // same handling
}
```

❖ try-with-resources (Java 7+):

```
----------------------------------
- Auto closes resources that implement AutoCloseable.

Example:
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
  String line = br.readLine();
}
```

❖ COMMONLY USED EXCEPTIONS:

```
-----------------------------
- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- NumberFormatException
- ClassCastException
- IllegalArgumentException
- IOException
- FileNotFoundException
- SQLException
```

❖ finalize() vs finally:

```
-------------------------
finalize():
- Called by GC before object is destroyed.
- Deprecated in Java 9+

finally:
- Always executes.
```

❖ BEST PRACTICES:

```
-------------------
- Catch only specific exceptions.
```

- Avoid empty catch blocks.
- Log exceptions.
- Don't catch Exception or Throwable unless absolutely needed.
- Use try-with-resources to avoid leaks.
- Use custom exceptions for clarity.

❖ MCQs (with answers):

Q1. What is the base class for all exceptions?
A. Throwable

Q2. Which exception is a checked exception?
A. IOException

Q3. What will happen if catch block doesn’t match any exception?
A. JVM terminates the program

Q4. Which block is always executed?
A. finally

Q5. Can we use try without catch?
A. Yes, if finally is used

Q6. What will be output?
```
  try { return 1; } finally { return 2; }
```
A. 2

Q7. Which of these are unchecked exceptions?
A. ArithmeticException, NullPointerException

Q8. Can constructor throw exception?
A. Yes

❖ TECHNICAL QUESTIONS WITH ANSWERS:

Q1. What’s the difference between throw and throws?
Ans:
- throw is used to explicitly throw an exception.
- throws is used in method signature to declare exceptions.
  Example:

```
void read() throws IOException {
   throw new IOException("File not found");
}
```

Q2. How does exception propagation work?
Ans:
- If an exception occurs in methodA(), and methodA() doesn't handle it, it is propagated to the caller method.
- The call stack is checked in reverse until it's either handled or the program crashes.
- Only unchecked exceptions propagate automatically.

Q3. What is try-with-resources?
Ans:
- Introduced in Java 7.
- Automatically closes resources (e.g., files, DB connections) that implement AutoCloseable.
  Example:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
   String line = br.readLine();
}
```

Q4. Can we use multiple catch blocks? Order?
Ans:
- Yes, multiple catch blocks are allowed.
- Child exceptions must be caught before parent exceptions.
  Example:

```
try {
   int a = 5 / 0;
} catch (ArithmeticException e) {
   System.out.println("Arithmetic Error");
} catch (Exception e) {
   System.out.println("Generic Error");
}
```

Q5. Can finally block override return values?

Ans:

- Yes. If there is a return statement in try or catch, and finally also has a return, then finally's return is used.

Example:

```
public int test() {
   try {
      return 1;
   } finally {
      return 2;  // this overrides
   }
}
// Output: 2
```

Q6. Can we catch multiple exceptions in one catch block?

Ans:

- Yes, from Java 7 onwards using | pipe symbol.
  Example:

```
try {
   // risky code
} catch (IOException | SQLException e) {
   e.printStackTrace();
}
```

Q7. Why is catching Throwable or Exception bad practice?

Ans:

- It hides the actual exception.
- Prevents specific handling.
- Catches unrecoverable errors (like OutOfMemoryError) which shouldn't be caught.

Q8. Explain custom exceptions with example.

Ans:

- Custom exceptions are user-defined exceptions that extend Exception or RuntimeException.
  Example:

```
class InvalidAgeException extends Exception {
   public InvalidAgeException(String msg) {
      super(msg);
   }
}
public void validateAge(int age) throws InvalidAgeException {
   if (age < 18) throw new InvalidAgeException("Age must be 18+");
}
```

Q9. Difference between Checked and Unchecked exceptions?

Ans:

Checked Exception	**Unchecked Exception**
Caught at compile time	Caught at runtime
Must be handled or declared	No need to handle explicitly
Example: IOException	Example: NullPointerException

Q10. Can we handle errors using catch?

Ans:

- Technically yes, but it's not recommended.
- Errors represent serious problems (like JVM crash).
- Catching them may lead to unstable programs.

❖ CODING QUESTIONS WITH ANSWERS:

Q1. Write a program to handle ArrayIndexOutOfBoundsException.

```
public class TestArray {
   public static void main(String[] args) {
      int[] arr = {1, 2};
      try {
         System.out.println(arr[5]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Index out of bounds!");
      }
   }
}
```

Q2. Write a program that uses try-with-resources.

```
import java.io.*;
public class ReadFile {
   public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
         String line;
         while ((line = br.readLine()) != null) {
            System.out.println(line);
         }
      } catch (IOException e) {
         System.out.println("File not found or error reading.");
      }
   }
}
```

Q3. Create a custom exception InvalidAgeException.

```
class InvalidAgeException extends Exception {
   public InvalidAgeException(String message) {
      super(message);
   }
}
public class AgeChecker {
   public static void validate(int age) throws InvalidAgeException {
      if (age < 18) {
         throw new InvalidAgeException("You must be 18+ to vote.");
      }
   }
public static void main(String[] args) {
      try {
         validate(16);
      } catch (InvalidAgeException e) {
         System.out.println("Error: " + e.getMessage());
      }
   }
}
```

Q4. Demonstrate nested try-catch-finally usage.

```
public class NestedTry {
   public static void main(String[] args) {
      try {
         try {
            int x = 5 / 0;
         } catch (ArithmeticException e) {
            System.out.println("Inner Catch: Divide by 0");
         } finally {
            System.out.println("Inner Finally");
         }
      } catch (Exception e) {
         System.out.println("Outer Catch");
      } finally {
         System.out.println("Outer Finally");
      }
   }
```

```
}
```
Q5. Simulate exception propagation through methods.
```
public class PropagationExample {
public void methodA() {
      methodB();
   }
public void methodB() {
      int x = 10 / 0;
   }
public static void main(String[] args) {
      PropagationExample obj = new PropagationExample();
      try {
         obj.methodA();
      } catch (ArithmeticException e) {
         System.out.println("Handled in main: " + e.getMessage());
      }
   }
}
```
Q6. Write a method to validate email, and throw custom InvalidEmailException.
```
class InvalidEmailException extends Exception {
   public InvalidEmailException(String msg) {
      super(msg);
   }
}
public class EmailValidator {
public static void validateEmail(String email) throws InvalidEmailException {
      if (email == null || !email.contains("@") || !email.endsWith(".com")) {
         throw new InvalidEmailException("Invalid email format");
      }
   }
public static void main(String[] args) {
      try {
         validateEmail("testemail.com");
      } catch (InvalidEmailException e) {
         System.out.println("Error: " + e.getMessage());
      }
   }
}
```

MULTI THREADING: This concept in java is to utilize CPU time efficiently.

THREAD IN JAVA

==========

HDD<->RAM<->Microprocessor

- **Program** - Its a set of instructions in Hard disk drive.
- Program is given to RAM first - **Loading**
- **Process/task** - When the program is loaded to RAM its stored and called as Process/task.

- This is managed by OS - i.e STOS - Single task operating system which loads only a single program to RAM and until the instructions of current process is executed by processor another program will not be loaded - which is time consuming.
- And even if many microprocessors were used each doing one process at same time, it could save time but its expensive

and it takes more space.
- So to make multi processes to be executed at the same time Bill gates introduced **MTOS - Multi Tasking Operating system(windows)**
- Which will divide the 1 CPU Time - 1 CPU Unit among all the tasks, (for eg: 1 min will be divided among 4 tasks, each task will be executed with that divided time one after other.) - So at last, one task will be executed at any time.
- **MULTI THREADING:** This concept in java is to utilize CPU time efficiently. - this is why we use multi threading.
- Thread is collection of instructions which is executed independently which a part of process and this phenomenon of creating multi threads in our process is called as Multi-threading (which is concurrent execution)

- **3 types of execution : Sequential, parallel execution, concurrent execution**

  1. Sequential Execution:
  - Tasks run one after another.
  - No overlap.
  - Simple and predictable.
  🧠 Example: Read → Write → Save (step-by-step)

  2. Concurrent Execution: Multi threading
  - Multiple tasks start and progress together.
  - CPU switches between them (time-sharing) when there is delay. (start - stop - resume)
  - Looks like multitasking on single core.
  🧠 Example: Cooking and replying to messages alternately.

  3. Parallel Execution:
  - Tasks run at the exact same time on multiple cores with no switch.
  - True multitasking.
  - Faster but needs multi-core CPU.
  🧠 Example: Two people cooking different dishes at once.

  QUICK COMPARISON:

Type	Same Time?	CPU Needed
Sequential	❌	Single-core
Concurrent	❌ (interleaved)	Single-core
Parallel	✅	Multi-core

-When main method is called first small stack frame for main called as main stack is created inside stack segment

==========

1. What is a Thread?

- Thread is collection of instructions which is executed independently which a part of process and this phenomenon of creating multi threads in our process is called as Multi-threading.(which is concurrent execution)
- A thread is a lightweight sub-process, the smallest unit of execution.
- A thread is a separate path of execution within a program.
- Java supports multithreading: executing multiple threads simultaneously.
- Multithreading helps in better CPU utilization.

2. Difference Between Process and Thread:

Process:
- Independent program in execution.
- Has its own memory space.
- Context switch is expensive.
- Uses IPC for communication.

Thread:
- A smaller part of a process.
- Shares memory with other threads.
- Context switch is lightweight.
- Communication is easier via shared memory.

3. Ways to Create a Thread in Java:

a) By Extending Thread class:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running");
   }
}
public class Test {
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();  // Starts the thread
   }
}
```

b) By Implementing Runnable interface:

```
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Runnable thread is running");
   }
}
public class Test {
   public static void main(String[] args) {
      Thread t = new Thread(new MyRunnable());
      t.start();
   }
}
```

Note: Runnable is preferred when extending another class (single inheritance).

4. Thread Lifecycle (States):
NEW → RUNNABLE → RUNNING → WAITING/TIMED_WAITING/BLOCKED → TERMINATED

- NEW: Thread is created but not started.
- RUNNABLE: Ready to run.
- RUNNING: Currently executing.
- BLOCKED: Waiting to acquire lock.
- WAITING: Waiting indefinitely for another thread's action.
- TIMED_WAITING: Waiting for a specific time.
- TERMINATED: Thread has completed execution or stopped.

5. Important Thread Methods:

- start(): Starts a new thread (calls run()).
- run(): Defines the thread's job.
- sleep(ms): Pauses execution for milliseconds.
- join(): Waits for another thread to finish.
- yield(): Suggests scheduler to give chance to other threads.
- isAlive(): Checks if thread is alive.
- setName(), getName(): Sets/Gets thread name.
- setPriority(), getPriority(): Sets/Gets priority.
- interrupt(): Interrupts a sleeping/waiting thread.

6. Thread Priorities:
- Range: 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY)
- Default: 5 (NORM_PRIORITY)
- thread.setPriority(Thread.MAX_PRIORITY);
Note: Priority is only a hint; actual scheduling is OS-dependent.

7. Thread Scheduler:
- Decides which thread runs next.
- Part of JVM.

- May vary between operating systems.
- No guarantee of order of execution.

8. Daemon Threads:
- Background service threads (e.g., Garbage Collector).
- Dies when all user threads are finished.
- thread.setDaemon(true); // Must be called before start()

9. Thread Synchronization:
- Prevents data inconsistency when multiple threads access shared resource.

a) Synchronized Method:

```
synchronized void method() {
   // thread-safe code
}
```

b) Synchronized Block:

```
synchronized(object) {
   // critical section
}
```

10. Inter-thread Communication:
Used when threads need to communicate with each other.

Common methods from Object class:
- wait(): Releases lock and waits.
- notify(): Wakes one waiting thread.
- notifyAll(): Wakes all waiting threads.
Note: These must be used inside synchronized blocks.

11. Thread Group:
Used to manage multiple threads together.
Example:
```
ThreadGroup group = new ThreadGroup("MyGroup");
Thread t1 = new Thread(group, new MyRunnable(), "Thread1");
```

12. Thread Pool (Executor Framework):
Used for better resource management and performance.

Example:
```
ExecutorService executor = Executors.newFixedThreadPool(5);
executor.execute(new MyRunnable());
executor.shutdown();
```

13. Java Concurrency Utilities (High-Level API):

- ExecutorService: Manages thread pools.
- Callable: Like Runnable but returns a value.
- Future: Holds result returned by Callable.
- CountDownLatch: Waits until count becomes zero.
- Semaphore: Controls access to resources.
- ReentrantLock: Explicit locking.
- CyclicBarrier: Synchronize threads at a barrier.

14. Runnable vs Callable:

Runnable:
- run() method
- Doesn’t return a result
- Cannot throw checked exceptions

Callable:

- call() method
- Returns a result
- Can throw checked exceptions

15. Common Thread Exceptions:

- IllegalThreadStateException: Calling start() on already started thread.
- InterruptedException: Thrown when thread is interrupted during sleep or wait.

16. Best Practices:

- Prefer Executor framework over manually creating threads.
- Handle InterruptedException correctly.
- Use synchronized carefully for performance.
- Prefer Immutable objects.
- Use volatile or Atomic classes for shared variables.
- Avoid blocking calls inside synchronized blocks.

JAVA MULTITHREADING

==========

📌 DEFINITION:
Multithreading is the concurrent execution of two or more threads to perform tasks simultaneously, maximizing CPU utilization and improving application performance.

- Thread is collection of instructions which is executed independently which a part of process and the phenomenon of creating multi threads in our process is called as Multi-threading (which is concurrent execution).
- **MULTI THREADING:** This concept in java is to utilize CPU time efficiently. - this is why we use multi threading.

```
//2 ways to create small stack
//1. By creating objects o those classes that are extending to Thread class
//2. Creating Thread class Objects and passing references of classes to constructor
```

Java supports multithreading at the language level using:
- java.lang.Thread class
- java.lang.Runnable interface
- java.util.concurrent package (advanced)

==========
1. BASICS OF THREADS
==========
- Thread: Smallest unit of execution within a process.
- Each thread has its own call stack but shares memory with other threads in the same process.

🧠 Key Concepts:
- Single-threaded: One task at a time (default in Java).
- Multi-threaded: Multiple tasks running concurrently or in parallel.

==========
2. ADVANTAGES OF MULTITHREADING
==========
- Better CPU utilization.
- Improves application responsiveness.
- Reduces idle CPU time (I/O wait, network delay).
- Useful in UI apps, servers, games, real-time systems.

==
3. THREAD CREATION METHODS - Diff ways of achieving multithreading
==
`Note: Thread class implements Runnable Interface`

Why there is 2nd method ? Because in first method there is disadvantage i.e we cant achieve multiple inheritance with class so there is Runnable Functional interface with run() method which can help us solve this diamond problem and also achieve multi threading.

A. Extending thread class
B. Implementing Runnable Interface.
C. Implementing callable

>>

🟩 a) EXTENDING Thread class:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread running");
   }
}
MyThread t1 = new MyThread();
t1.start();
```

🧠 Multithreading in Java – Memory & Conceptual Understanding (for first method)

🔹 **Multithreading Basics**

Multithreading = Concurrent execution of two or more threads.
Threads perform tasks **simultaneously**, improving **CPU utilization** and **application performance**.
A **Thread** is a unit of execution — a collection of instructions that runs independently.
Even though tasks seem independent, in a normal Java program they may still **run sequentially**, unless you use actual threads.
Java's main() method is also run on a thread — it's known as the **main thread**.

🔹 **Thread Creation & Memory Understanding**

To make tasks run independently, place each task inside the run() method of a class that **extends Thread**.
When you create an object of a class that extends Thread, it behaves like:

- Memory allocated in **Heap** for object.
- A **small stack (Thread stack)** is created inside the **Stack Segment** for that thread.

🔹 **Thread Stack Naming (Memory Perspective)**

Demo d = new Demo();
- A new thread stack called **Thread-0** is created in stack segment.

Demo1 d1 = new Demo1();
- Creates **Thread-1** stack in stack segment.

Demo2 d2 = new Demo2();
- Creates **Thread-2** stack in stack segment.

🔹 **Calling Threads Properly**

DO NOT directly call the run() method.
Use .start() method to start the thread — it internally calls the run() method.
Example:

- d.start(); → Creates stack frame of run() method inside **Thread-0**.
- d1.start(); → Creates stack frame of run() inside **Thread-1**.
- d2.start(); → Creates stack frame of run() inside **Thread-2**.

Instead if we call d.run() directly the stack frame will be created inside main thread/small stack of main which will result in sequential execution not concurrent, so there will be no multi threading.

🔹 **Thread Scheduler Role**

Java has a built-in **Thread Scheduler**.
It is a part of the JVM, acts like a runtime compiler (like RTS).
Thread Scheduler decides:

- **When** and **which thread** to execute.
- Switches threads based on **priority**, **delay**, or **availability**.

It improves time efficiency by **scheduling other threads during delays/sleeps**.
Once all threads complete, **JVM terminates** the execution.

```
package multithreading.injava;
import java.util.Scanner;

class Demo extends Thread{ //Thread class implements Runnable Interface
	@Override
	public void run() {
		//1st task
		System.out.println("Adding started");
		Scanner sc = new Scanner(System.in);
		System.out.println(sc.nextInt()+sc.nextInt());
		System.out.println("Adding completed");
	}
}

class Demo1 extends Thread{
	@Override
	public void run() {
		//2nd task - chars
			System.out.println("Printing characters");
			for(int i=65; i<=69; i++) {
				System.out.println((char)i);

				try {
					Thread.sleep(2000);
				}catch(Exception e) {
					e.printStackTrace();
				}
			}
			System.out.println("Printing characters completed");
	}
}

class Demo2 extends Thread{
	@Override
	public void run() {
		//3rd task - num
			System.out.println("Printing numbers");
			for(int i=1; i<=10; i++) {
				System.out.println(i);

				try {
					Thread.sleep(2000);
				}catch(Exception e) {
					e.printStackTrace();
				}
			}
			System.out.println("Completed");
	}
}


public class MultithreadingIntro1stWay {

	public static void main(String[] args) {

		//Multi-thrading - Multiple/many threads
		//Threads - Thread is collection of instructions which is executed independently which a part of process
		//Multi-threading is the concurrent execution of two or more threads to perform tasks simultaneously, maximizing CPU utilization and improving application performance.


		//Multi-threading code - Has dependent processes(even though they are independent) because , tasks are executed one after another sequentially but we can make it run concurrently using multi - threading
		//In java main method stack frame is also considered as Thread
```

```
		//to make these tasks independent we can put each of them in run() methods from Thread
class as above.

		//small stack for d will be created inside stack segment(after as usual Object memory is
allocated inside Heap) when we create object for a class that extends to Thread class, and this small
stack is Thread-0(for first object of class with first task).
		//Then similarly small stack for d1 is created in stack segment called as Thread-1
		//Then similarly small stack for d2 is created in stack segment called as Thread-2
		Demo d =new Demo();
		Demo1 d1 = new Demo1();
		Demo2 d2 = new Demo2();

		//As per principle of Multi-threading we must not call run() methods directly instead
they should be called using start method .
		//When d.start() is called this run method's stack frame is created inside Thread-0 small
stack which is inside Stack segment
		d.start();
		//When d1.start() is called this run method's stack frame is created inside Thread-1
small stack which is inside Stack segment
		d1.start();
		//When d.start() is called this run method's stack frame is created insie Thread-2 small
stack which is inside Stack segment
		d2.start();

		//there is a software like RTS, compiler called Thread Scheduler which schedules threads
which are independent.
		//Thread Scheduler tell JVM to start executing from 0, and based on delays, to save time,
it will schedule other threads to execute., after all are executed it is terminated

		//What happens if we call run() method? -
		//if we call d.run() directly the stack frame will be created inside main thread/small
stack of main which will result in sequential execution not concurrent, so there will be no multi-
threading.
	}

}
```

==========

🟩 b) IMPLEMENTING Runnable interface:

```
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Runnable thread");
   }
}
Thread t = new Thread(new MyRunnable());
t.start();
```

```
package multithreading.injava;
import java.util.Scanner;

class Adding implements Runnable{ //Runnable functional interface
	@Override
	public void run() {
		//1st task
		System.out.println("Adding started");
		Scanner sc = new Scanner(System.in);
		System.out.println(sc.nextInt()+sc.nextInt());
		System.out.println("Adding completed");
	}
}

class PrintChars implements Runnable{
	@Override
	public void run() {
		//2nd task - chars
				System.out.println("Printing characters");
				for(int i=65; i<=69; i++) {
					System.out.println((char)i);
```

```
                        try {
                                Thread.sleep(2000);
                        }catch(Exception e) {
                                e.printStackTrace();
                        }
                }
                System.out.println("Printing characters completed");
        }
}

class PrintNum implements Runnable{
        @Override
        public void run() {
                //3rd task - num
                System.out.println("Printing numbers");
                for(int i=1; i<=10; i++) {
                        System.out.println(i);

                        try {
                                Thread.sleep(2000);
                        }catch(Exception e) {
                                e.printStackTrace();
                        }
                }
                System.out.println("Completed");
        }
}


public class MultithreadingIntro2ndWay {

        public static void main(String[] args) {

                //Note: Thread class implements Runnable Interface

                //2nd way of achieving is by implementing Runnable functional interface.
                /*Why there is 2nd method ? Because in first method there is disadvantage
                 * i.e we cant achieve multiple inheritance with class
                 * so there is Runnable Functional interface with run() method which can help us solve
this diamond problem and also achieve multi-threading.
                 */

                //When we create object for classes implementing Runnable interface there will be no
separate small stacks for each class as in first method,
                //So we must create 3 objects of Thread class directly as below it will create 3 small
stacks - Thread-0 , Thread-1, Thread-2 in heap segment.
                //And pass the reference vars of the classes that are implementing runnable interface to
Constructor of the thread objects.
                Adding a = new Adding();
                PrintChars pc = new PrintChars();
                PrintNum pn = new PrintNum();

                Thread t1 = new Thread(a);
                Thread t2 = new Thread(pc);
                Thread t3 = new Thread(pn);

                //Now to make respective run() methods go sit in small stacks of each threads we use
start() - which will result concurrent execution.
                t1.start();
                t2.start();
                t3.start();

                //Then Thread scheduler tells JVM to assign schedules for each threads as same in first
method.
        }

}
```

🟩 c) IMPLEMENTING Callable + Future:

```
Callable<String> task = () -> "Result";
```

```
ExecutorService ex = Executors.newSingleThreadExecutor();
Future<String> result = ex.submit(task);
System.out.println(result.get());
```

🟩 d) Using Anonymous class:

```
Thread t = new Thread(new Runnable() {
   public void run() {
      System.out.println("Anonymous thread");
   }
});
t.start();
```

🟩 e) Using Lambda expression (Java 8+):

```
Thread t = new Thread(() -> System.out.println("Lambda thread"));
t.start();
```

🟩 f) Using Timer and TimerTask (for scheduled threads):

```
Timer timer = new Timer();
timer.schedule(new TimerTask() {
   public void run() {
      System.out.println("Timer task executed");
   }
}, 0, 1000); // Runs every second
```

==========

4. THREAD LIFE CYCLE (STATES)

==========

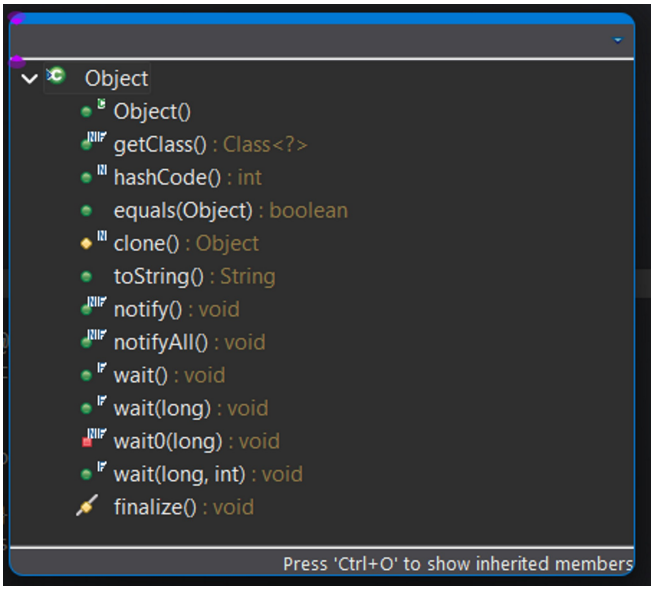
7/5 States: - 5 if 4,5,6th state considered all together Block sate by some developers
- NEW: Thread is created but not started.
Start()
- RUNNABLE: Ready to run, waiting for CPU.
- RUNNING: Currently executing. - dead if no other methods are called
- BLOCKED: Waiting to acquire lock. - resource unavailable
- WAITING: Waiting indefinitely for another thread. - wait()
- TIMED_WAITING: Waiting for a specific period. - sleep()
- TERMINATED: Finished execution or aborted. - dead

-From any given state(except new state) thread can enter dead state with the help of interrupt

-**DEADLOCK** - Its state where 2/more Threads enter Blocked state infinitely/forever due to interdependency of releasing lock
-**LIVELOCK** - Its a state where thread keeps on rotating in states without entering DEAD state or block state forever
-**STARVATION** - When thread quickly goes to dead state from block state or doesn't get resources because of other threads having highest priority . So in total thread when doesn't get resources is called starvation.

Transitions:
- start() → NEW → RUNNABLE → RUNNING
- sleep()/wait() → TIMED_WAITING
- notify()/notifyAll() → back to RUNNABLE,
- run() completes → TERMINATED

- wait() → Thread enters **WAITING** state, waits indefinitely until notify()/notifyAll() is called — from **Object class**.
- wait(long timeout) → Thread enters **TIMED_WAITING**, waits for given time, resumes after timeout or if notified — from **Object class**.
- wait(long timeout, int nanos) → More precise timed wait (rarely used), resumes after timeout or notification — from **Object class**.
- notify() → Wakes up **one** waiting thread (randomly chosen) → moves to **RUNNABLE** — from **Object class**.
- notifyAll() → Wakes up **all** waiting threads → all move to **RUNNABLE** — from **Object class**.




```java
package multithreading.injava;

class Warrior extends Thread{

	String res1 = "bramhastra";
	String res2 = "Sarpastra";
	String res3 = "Pashupatastra";

	@Override
	public void run() { // RUNNING state: When thread is scheduled and run() starts executing
		String name = Thread.currentThread().getName(); // Still in RUNNING

		if(name.equalsIgnoreCase("Arjuna")) {
			arjun(); // Depending on name, thread goes into Arjuna method
		}else if(name.equalsIgnoreCase("Karna")) {
			karna(); // Or Karna method
		}
	}

	void arjun() {
		try {
			Thread.sleep(2000); // TIMED_WAITING: Thread sleeps for 2 seconds before acquiring any lock,
			//When a thread is in sleep and finished sleep , that thread goes back to runnable state before going to running

			synchronized (res1){ // BLOCKED if another thread holds res1; else enters RUNNING after acquiring lock
				System.out.println("Arjuna acquired Bramhastra");

				Thread.sleep(3000); // TIMED_WAITING: while holding res1

				synchronized (res2){ // BLOCKED if another thread holds res2
					System.out.println("Arjuna acquired Sarpastra");

					Thread.sleep(3000); // TIMED_WAITING: while holding res1 + res2

					synchronized (res3){ // BLOCKED if another thread holds res3
						System.out.println("Arjuna acquired Pashupatastra");
						// When done: thread releases res3 lock
					} // Exiting res3 block: res3 monitor is released
				} // Exiting res2 block: res2 monitor is released
			} // Exiting res1 block: res1 monitor is released
			// Thread will eventually reach TERMINATED after completing run()
		}catch(Exception e) {
			e.printStackTrace();
		}
	}
```

```
	void karna() {
		try {
			Thread.sleep(2000); // TIMED_WAITING

			synchronized (res1){ // BLOCKED if already held, else RUNNING
				System.out.println("Karna acquired Bramhastra");

				Thread.sleep(3000); // TIMED_WAITING

				synchronized (res2){ // BLOCKED if already held
					System.out.println("Karna acquired Sarpastra");

					Thread.sleep(3000); // TIMED_WAITING

					synchronized (res3){ // BLOCKED if already held
						System.out.println("Karna acquired Pashupatastra");
						// Exiting will release res3
					}
				}
			}
			// Thread reaches TERMINATED after run() finishes
		}catch(Exception e) {
			e.printStackTrace();
		}
	}
}

public class ThreadLifeCycle {

	public static void main(String[] args) {

		// Thread Life cycle :
		/*
		 * NEW       → When thread object is created using new Thread()
		 * RUNNABLE  → After start() is called; thread is ready to run and waiting for CPU
		 * RUNNING   → When CPU picks the thread and executes run()
		 * BLOCKED   → When thread tries to enter a synchronized block but the lock is already
held
		 * TIMED_WAITING → When thread is sleeping or waiting with a timeout (e.g., sleep, join)
		 * WAITING   → Waiting indefinitely (e.g., wait())
		 * TERMINATED → When run() method completes or thread is stopped
		 */

		//Creating 2 threads for each method
		Warrior w1 = new Warrior(); // NEW: Thread object created
		Warrior w2 = new Warrior(); // NEW: Another thread object created
		w1.setName("Arjuna");
		w2.setName("Karna");

		w1.start(); // RUNNABLE: Thread w1 ready to run
		w2.start(); // RUNNABLE: Thread w2 ready to run
		// JVM Thread Scheduler will pick one or both to move to RUNNING based on availability
	}

}
```

DEADLOCK:

```
package multithreading.injava;

class Warriors extends Thread {

	String res1 = "bramhastra";
	String res2 = "Sarpastra";
	String res3 = "Pashupatastra";

	@Override
	public void run() { // RUNNING state: Thread is now executing
		String name = Thread.currentThread().getName(); // RUNNING state
```

```
		if (name.equalsIgnoreCase("Arjuna")) {
			arjun(); // Arjuna thread starts execution
		} else if (name.equalsIgnoreCase("Karna")) {
			karna(); // Karna thread starts execution
		}
	}

	void arjun() {
		try {
			Thread.sleep(2000); // TIMED_WAITING: Thread sleeps for 2 sec, then goes back to
RUNNABLE → RUNNING

			synchronized (res3) { // Tries to acquire res3 monitor
				// If Karna has already locked res3, Arjuna thread goes to BLOCKED state
				System.out.println("Arjuna acquired Bramhastra");

				Thread.sleep(3000); // TIMED_WAITING: Holds res3 while sleeping

				synchronized (res2) { // Tries to acquire res2
					// BLOCKED if Karna has already locked res2
					System.out.println("Arjuna acquired Sarpastra");

					Thread.sleep(3000); // TIMED_WAITING again

					synchronized (res1) { // Tries to acquire res1
						// If Karna is already holding res1 → DEADLOCK
						System.out.println("Arjuna acquired Pashupatastra");
					} // res1 released
				} // res2 released
			} // res3 released

			// If completed successfully → TERMINATED
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	void karna() {
		try {
			Thread.sleep(2000); // TIMED_WAITING: Sleeps 2s, back to RUNNABLE → RUNNING

			synchronized (res1) { // Karna locks res1 first
				// If Arjuna later tries to lock res1 (already held) → Arjuna gets BLOCKED
				System.out.println("Karna acquired Bramhastra");

				Thread.sleep(3000); // TIMED_WAITING

				synchronized (res2) { // Tries to lock res2 (if already held → BLOCKED)
					System.out.println("Karna acquired Sarpastra");

					Thread.sleep(3000); // TIMED_WAITING

					synchronized (res3) { // Tries to acquire res3
						// If Arjuna already holds res3 → DEADLOCK
						System.out.println("Karna acquired Pashupatastra");
					} // res3 released
				} // res2 released
			} // res1 released

			// When run() completes → TERMINATED
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}

public class ThreadLifeCycleDeadLock {

	public static void main(String[] args) {

		// Thread Life cycle :
```

```
		/*
		 * NEW           → When thread object is created using new Thread()
		 * RUNNABLE      → After start() is called; ready to run, waiting for CPU
		 * RUNNING       → When JVM picks the thread and run() starts executing
		 * TIMED_WAITING → Thread is sleeping or waiting with timeout (e.g., sleep)
		 * BLOCKED       → Waiting to acquire a lock (monitor) already held by another thread
		 * WAITING       → Waiting indefinitely (e.g., wait() or join() without timeout)
		 * TERMINATED    → After run() completes or thread is killed
		 * DEADLOCK      → Special case of BLOCKED: Two or more threads are waiting for each other's locks forever
		 * DEADLOCK - Its state where 2/more Threads enter Blocked state infinitely/forever due to interdependency of releasing lock
		 * LIVELOCK -
		 */

		//Creating 2 threads for each warrior
		Warriors w1 = new Warriors(); // NEW: Thread created but not yet started
		Warriors w2 = new Warriors(); // NEW: Another thread created

		w1.setName("Arjuna");
		w2.setName("Karna");

		w1.start(); // RUNNABLE: ready for execution
		w2.start(); // RUNNABLE: ready for execution

		// JVM Thread Scheduler may:
		// 1. Allow Arjuna to lock res3, then Karna locks res1 → Both will BLOCK waiting on each other's resource
		// 2. DEADLOCK occurs if:
		//    Arjuna holds res3 and waits for res2 → which Karna holds after res1 → and Karna tries to acquire res3
		//    → Both threads are BLOCKED forever: classic DEADLOCK
	}

}
```

==========
5. THREAD METHODS
==========

Method Summary:
- start() – starts a new thread.
- run() – thread logic (called by JVM).
- sleep(ms) – pause thread for given time.
- yield() – pause and allow others to run.
- join() – wait for another thread to finish.
- interrupt() – interrupt a sleeping/waiting thread.
- isAlive() – checks if thread is still running.
- setName(), getName() – thread name.
- setPriority(), getPriority() – priority (1-10).
- currentThread() – returns current thread reference. - its a static method

```
package multithreading.injava;

class Alpha extends Thread{
	@Override
	public void run() {
		Thread t = currentThread(); //can be called directly method name like this as we are inheriting this method by extending to Thread.
		System.out.println(t); //When called through start method it prints abt current thread - Thread[#21,Thread-0,5,main] -And this is Thread-0
	}
}

class Beta extends Thread{
	@Override
	public void run() {
		Thread t = currentThread(); //can be called directly method name like this as we are
```

```
inheriting this method by extending to Thread.
		System.out.println(t); //Thread-1 : Thread[#22,Thread-1,5,main]
	}
}

class Gamma extends Thread{
	@Override
	public void run() {
		System.out.println(currentThread());
		setName("S");
		System.out.println(currentThread());
	}
}

public class MultiThreadingInbuiltMethods extends Thread{ //Can class with main method extend Thread - Yes

	public static void main(String[] args) {

		//1.static currentMethod() in Thread class: which will give information about the thread that is currently executing
		Thread t = Thread.currentThread();
		System.out.println(t);
		//O/p : Thread[#1,main,5,main]
		//1st term- Thread Id, 2nd term - Name of thread, 3rd term - Priority(By Default java gives 5 to every thread which is average of Priorities in scale(1 to 10), 4th term - Thread group

		//We can get each terms separately also:
		System.out.println(t.getName()); //Name of thread only
		System.out.println(t.getPriority()); //prints priority only
		System.out.println(t.getThreadGroup()); //thread group and max priority

		//We can change the thread name , priority as below
		t.setName("v");
		System.out.println(t.getName());
		t.setPriority(4);
		System.out.println(t.getPriority());

		//Knowing current thread by calling this method in run() method and start it in main method - will give that threads details
//		Alpha a = new Alpha();
//		a.start();

		Alpha a = new Alpha();
		Beta b = new Beta();
		a.start();
		b.start();

		//Changing name of thread inside run method
		Gamma g = new Gamma();
		g.start();

	}
}
----------------------


package multithreading.injava;

class Alp extends Thread{
	@Override
	public void run() {
		try{
			System.out.println(getName()+" started to executing");
			sleep(3000);
			System.out.println(getName()+" is executing");
			sleep(3000);
			System.out.println(getName()+" completed executing");
			sleep(3000);
		}
		catch(Exception e) {
			e.printStackTrace();
```

```
		}
	}
}

class Bet extends Thread{
	@Override
	public void run() {
		try{
			System.out.println(getName()+" is executing");
			sleep(3000);
			System.out.println(getName()+" is executing");
			sleep(3000);
			System.out.println(getName()+" completed executing");
			sleep(3000);
		}
		catch(Exception e) {
			e.printStackTrace();
		}
	}
}



public class MultiThreadingInbuiltMethods2 extends Thread { //Can class with main method extend
Thread - Yes

	public static void main(String[] args) throws Exception{

		//join(), isAlive()

//		System.out.println("Main thread started executing");
//		Alp a = new Alp();
//		Bet b =new Bet();
//		a.start();
//		b.start();
//		System.out.println("Main thread completed executing");


		//O/p: Main thread first starts and completes execution, later thread 0 and 1 will
complete.
		/*Main thread started executing
		Main thread completed executing
		Thread-1 is executing
		Thread-0 started to executing
		Thread-1 is executing
		Thread-0 is executing
		Thread-1 completed executing
		Thread-0 completed executing
		*/

		//-----------------------------------------------------------------

		// To make main thread start first and complete last, while threads 0 and 1 (a and b)
execute in between
		System.out.println("Main thread started executing"); // Main thread begins

		Alp a = new Alp(); // Creates object 'a' of thread class Alp → Thread-0 stack created in
stack segment
		Bet b = new Bet(); // Creates object 'b' of thread class Bet → Thread-1 stack created in
stack segment

		a.start(); // Starts Thread-0 → Its own run() method executes on a separate thread
		a.join();  // Main thread waits for Thread-0 to finish before continuing

		System.out.println(a.isAlive()); //false //checks whether the thread is executed or has
already executed

		b.start(); // Starts Thread-1 → Executes its run() independently
		b.join();  // Main thread waits for Thread-1 to finish before continuing

		System.out.println(b.isAlive()); //false //checks whether the thread is executed or has
already executed
```

```
            System.out.println("Main thread completed executing"); // Main thread finishes after both
a and b complete

            //o/p
            /*
             * Main thread started executing
Thread-0 started to executing
Thread-0 is executing
Thread-0 completed executing
Thread-1 is executing
Thread-1 is executing
Thread-1 completed executing
Main thread completed executing
             */

            //-------------------------------------------------------------------------

            //2 ways to create small stack
            //1. By creating objects o those classes that are extending to Thread class
            //2. Creating Thread class Objects and passing references of classes to constructor



      }
}
```

==========
6. THREAD PRIORITY
==========

Priority Levels:
- MIN_PRIORITY = 1
- NORM_PRIORITY = 5 (default)
- MAX_PRIORITY = 10

Note:
- Scheduler may or may not respect priority.
- Thread with higher priority might get preference, but not guaranteed.

==========
7. THREAD SCHEDULER
==========

- Thread scheduler is part of JVM/OS that decides which thread to run.
- Uses time-slicing (round-robin) or priority-based.
- JVM does not guarantee order of thread execution.

=> Multi Threading using single/one run method:

```
package multithreading.injava;

import java.util.Scanner;

class SingleRunMethod extends Thread{

      @Override
      public void run() {

            String name = getName();

            if(name.equals("ADD")) {
                  adding();
            }else if(name.equals("CHARS")) {
                  printchars();
            }else {
                  printNums();
```

```
		}
	}

	void adding() {
		//1st task
			System.out.println("Adding started");
			Scanner sc = new Scanner(System.in);
//			System.out.println(sc.nextInt()/sc.nextInt()); //When we give denominator as 0 , there will be execution, only this thread will be abruptly terminated and other threads will continue executing as threads are independent.
			//Exception in thread "Thread-0" java.lang.ArithmeticException: / by zero - So we can use try catch

			try {
				System.out.println(sc.nextInt()+sc.nextInt());
			}
			catch(Exception e) {
//				e.printStackTrace();
				System.out.println(e.getMessage());
			}
			System.out.println("Adding completed");
		}

	void printchars() {
		//2nd task - chars
		System.out.println("Printing characters");
		for(int i=65; i<=69; i++) {
			System.out.println((char)i);

			try {
				Thread.sleep(2000);
			}catch(Exception e) {
				e.printStackTrace();
			}
		}
		System.out.println("Printing characters completed");
}

	void printNums() {
		//3rd task - num
		System.out.println("Printing numbers");
		for(int i=1; i<=10; i++) {
			System.out.println(i);

			try {
				Thread.sleep(2000);
			}catch(Exception e) {
				e.printStackTrace();
			}
		}
		System.out.println("Completed");
	}
}

public class MultiThreadingWithSingleRunMethod {

	public static void main(String[] args) {

//		Multi-Threading With Single Run Method

		//Creating 3 threads for each method
		SingleRunMethod s1 = new SingleRunMethod();
		s1.setName("ADD");

		SingleRunMethod s2 = new SingleRunMethod();
		s2.setName("CHARS");

		SingleRunMethod s3 = new SingleRunMethod();
		s3.setName("NUMS");

		//To make one run() method to sit in all three threads small stack also
		s1.start();
		s2.start();
```

```
            s3.start();

            //As usual Thread scheduler will tell JVM to execute other threads when current thread
has delay
      }

}
```

==========
8. SYNCHRONIZATION (Thread Safety)
==========

🔸 What is Synchronization?

Is the locking system in Java where we can lock the shared resources/methods with help of synchronized keyword to ensure only one thread can access instructions/executed at any point of time.

Synchronization is the process of controlling access to **shared resources** in a **multi-threaded environment**, to avoid **data inconsistency** and **race conditions**.

In Java, **multiple threads may try to read/write a shared variable at the same time**, which can lead to unpredictable results. Synchronization ensures that **only one thread accesses the critical section** of code at a time.

🔸 Why is Synchronization Needed?

❌ Problem Without Synchronization:
- Threads may overwrite each other's data.
- Leads to **inconsistent**, **corrupt**, or **unexpected results**.

✅ Solution:
- Synchronization ensures **thread safety** – only one thread can access a critical section at any given time.

- String Buffer is synchronized(methods), single thread can access the methods of this class at any point of time(Thread-safe) (No Race condition, so slow),(No dead lock
- and String builder is non synchronized(not Thread safe) where multiple threads can access the methods of this class at any point of time. (Race condition, so fast), (Chances of Dead lock

---
🔸 Critical Section

- A **critical section** is a block of code that **modifies shared resources**.
- It must be accessed by **only one thread at a time** to avoid inconsistency.
- Synchronization protects this critical section.

Semaphore: They are the instructions that are being executed by single thread at any point of time.
Monitors : Monitors are instructions where in only one thread is executing all of them at any point of time.

- **Use Monitor** when you want:
  - Only one thread at a time (mutual exclusion)
  - Thread communication (wait/notify)
  - Simple and built-in mechanism
- **Use Semaphore** when you want:
  - To allow a **limited number of threads**
  - A more flexible, manual control of permits
  - Advanced synchronization behavior (e.g., bounded access)

==========
🔹 1. SYNCHRONIZED METHOD (Object-level lock)
==========

📌 Syntax:
synchronized returnType methodName() {

```
    // critical section
}
```

📌 How it works:
- Locks the **current object (`this`)**.
- Only one thread can execute this method on the same object at a time.

📌 Example:
```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++; // critical section
    }
}
```

Note:
- If two threads are working on **different objects**, they won't block each other.

```
package multithreading.injava;

class Bathroom{
	synchronized void bathroom() { //synchronized keyword make only one thread to be executed/access these methods instructions at a time
		try {
			System.out.println(Thread.currentThread().getName()+" has entered the bathroom");
			Thread.sleep(2000);
			System.out.println(Thread.currentThread().getName()+" is using the bathroom");
			Thread.sleep(2000);
			System.out.println(Thread.currentThread().getName()+" has left the bathroom");
		}catch(Exception e) {
			e.printStackTrace();
		}
	}
}

class Boy extends Thread{

	Bathroom ba;
	public Boy(Bathroom ba) { //This will receive the Bathroom b reference /address and now ba instance variable in Boy class will refer to Bathroom Object
		this.ba = ba;
	}

	@Override
	public void run() {
		ba.bathroom();
	}
}

class Girl extends Thread{ //Now Girl class has access to bathroom and can access its method in run to make thread work

	Bathroom ba;
	public Girl(Bathroom ba) {
		this.ba = ba;
	}

	@Override
	public void run() {
		ba.bathroom();
	}
}

class Others extends Thread{

	Bathroom ba;
	public Others(Bathroom ba) {
		this.ba = ba;
```

```
	}

	@Override
	public void run() {
		ba.bathroom();
	}
}

public class SynchronizationThreadSafety  {

	public static void main(String[] args) throws Exception{

		// SYNCHRONIZATION (Thread Safety): Is the locking system in Java where we can lock the shared resources/methods with help of synchronized keyword to ensure only one thread can access instructions/executed at any point of time.

		Bathroom b = new Bathroom(); //We must pass this reference to below objects to make those threads to go to this Bathroom object

		Boy bo = new Boy(b);
		Girl g = new Girl(b);
		Others o = new Others(b);

		bo.setName("Boy");
		g.setName("Girl");
		o.setName("Others");

		//This is same as sequential
//		bo.start();
//		bo.join();
//		g.start();
//		g.join();
//		o.start();
//		o.join();

		//So we can use synchronized keyword to Bathroom methods.
	}

}
```

==========

🔹 2. SYNCHRONIZED BLOCK (Object-level lock)

==========

📌 Syntax:

```
synchronized (lockObject) {
   // critical section
}
```

📌 Why use it?

- Synchronize only the **critical part**, not the whole method.
- Increases performance.

📌 Example:

```
class Counter {
   int count = 0;
   Object lock = new Object();

   void increment() {
      synchronized (lock) {
         count++;
      }
   }
}
```

Note:

- Can use `this` as lock or a separate object.
- More efficient than synchronizing entire method.

```java
package multithreading.injava;

class Bathroom1{
	//Now to make only 5 and 6 to be synchronized we can use synchronized keyword for a block also
	void bathroom() { //synchronized keyword make only one thread to be executed/access these methods instructions at a time
		try {
			System.out.println(Thread.currentThread().getName()+" is executing 1");
			Thread.sleep(2000);
			System.out.println(Thread.currentThread().getName()+" is executing 2");
			Thread.sleep(2000);
			System.out.println(Thread.currentThread().getName()+" is executing 3");
			Thread.sleep(2000);
			System.out.println(Thread.currentThread().getName()+" is executing 4");
			Thread.sleep(2000);

			//So here only 5 and 6 will be synchronized
			//Why synchronize block: Synchronize only the **critical part**, not the whole method.
			synchronized(this){ //So this keyword will make only current object thats executing to access this statements and lock until this block is executed.
				System.out.println(Thread.currentThread().getName()+" is executing 5");
				Thread.sleep(2000);
				System.out.println(Thread.currentThread().getName()+" is executing 6");
			}
		}catch(Exception e) {
			e.printStackTrace();
		}
	}
}

class Boy1 extends Thread{

	Bathroom1 ba;
	public Boy1(Bathroom1 ba) { //This will receive the Bathroom b reference /address and now ba instance variable in Boy class will refer to Bathroom Object
		this.ba = ba;
	}

	@Override
	public void run() {
		ba.bathroom();
	}
}

class Girl1 extends Thread{ //Now Girl class has access to bathroom and can access its method in run to make thread work

	Bathroom1 ba;
	public Girl1(Bathroom1 ba) {
		this.ba = ba;
	}

	@Override
	public void run() {
		ba.bathroom();
	}
}

class Others1 extends Thread{

	Bathroom1 ba;
	public Others1(Bathroom1 ba) {
		this.ba = ba;
	}

	@Override
	public void run() {
		ba.bathroom();
	}
}
```

```
public class SynchronizationBlockThreadSafety2  {

	public static void main(String[] args) throws Exception{

		// SYNCHRONIZATION (Thread Safety): Is the locking system in Java where we can lock the shared resources/methods with help of synchronized keyword to ensure only one thread can access instructions/executed at any point of time.

		Bathroom1 b = new Bathroom1(); //We must pass this reference to below objects to make those threads to go to this Bathroom object

		Boy1 bo = new Boy1(b);
		Girl1 g = new Girl1(b);
		Others1 o = new Others1(b);

		bo.setName("Boy");
		g.setName("Girl");
		o.setName("Others");

		//This is same as sequential
//		bo.start();
//		bo.join();
//		g.start();
//		g.join();
//		o.start();
//		o.join();

		//So we can use synchronized keyword to Bathroom methods.
		bo.start();
		g.start();
		o.start();
		/*
		 * Girl is executing 1
Others is executing 1
Boy is executing 1
Others is executing 2
Girl is executing 2
Boy is executing 2
Boy is executing 3
Girl is executing 3
Others is executing 3
Others is executing 4
Boy is executing 4
Girl is executing 4
Others is executing 5
Others is executing 6
Girl is executing 5
Girl is executing 6
Boy is executing 5
Boy is executing 6

		 * */
	}

}
```

==========

🔹 3. STATIC SYNCHRONIZED METHOD (Class-level lock)

==========

📌 Syntax:

```
synchronized static void methodName() {
   // critical section
}
```

📌 How it works:

- Locks the **Class object**, not an instance.
- Useful when **shared data is static**.

📌 Example:
```
class Counter {
   static int count = 0;

   synchronized static void increment() {
      count++;
   }
}
```

Note:
- Only one thread can execute **any static synchronized method** of the class at a time.

==========

🔹 4. CLASS-LEVEL LOCK USING synchronized(ClassName.class)

==========

📌 Example:
```
class MyClass {
   static void method() {
      synchronized (MyClass.class) {
         // critical section
      }
   }
}
```

Note:
- Useful when you need to lock class-wide operations.

==========

🔹 5. MEMORY VIEW – HOW SYNCHRONIZATION WORKS INTERNALLY

==========

When a thread enters a synchronized block/method:
✅ It:
- **Acquires a lock** (monitor) on the object or class.
- **Flushes its working memory** (local cache).
- **Reads fresh values** from main memory.
- Performs operations.
- **Writes back updated values** to main memory.
- **Releases the lock** once done.

❌ Without synchronization:
- Threads may read **stale/cached values**.
- No guarantee of visibility of changes to other threads.

==========

🔹 6. COMMON ISSUES (If Not Synchronized Properly)

==========

❌ Race condition → Two threads modify data at the same time → inconsistent results
❌ Dirty read → One thread reads partial result of another
❌ Thread interference → Threads corrupt each other's results
❌ Deadlock → Two threads wait on each other indefinitely
❌ Livelock → Threads keep changing state without progress
❌ Starvation → Low priority thread never gets access

==========

🔹 7. BEST PRACTICES FOR SYNCHRONIZATION

==========

✅ Keep synchronized blocks as **short** as possible
✅ Avoid **nested synchronized blocks** (can lead to deadlock)

✅ Avoid **blocking operations** (e.g., I/O) inside synchronized blocks
✅ Prefer **synchronized blocks** over methods for fine control
✅ For multiple threads, consider using:
  - `ReentrantLock` (explicit lock with features)
  - `AtomicInteger`, `AtomicReference`, etc.
  - `ConcurrentHashMap`, `CopyOnWriteArrayList` (for collections)

==========
🔹 8. ALTERNATIVES TO synchronized
==========

1. **ReentrantLock** (java.util.concurrent.locks)
  - More flexible than synchronized
  - Allows tryLock(), interruptible locking

2. **Atomic Variables** (java.util.concurrent.atomic)
  - Lock-free, thread-safe variable manipulation
  - e.g., AtomicInteger, AtomicBoolean

3. **Concurrent Collections**
  - Designed for high-concurrency environments
  - Avoid manual synchronization
  - Examples: ConcurrentHashMap, BlockingQueue

==========
🔹 9. SYNCHRONIZATION VS VOLATILE
==========

Feature	synchronized	volatile
Purpose	Mutual exclusion & visibility	Only visibility (no locking)
Locks used?	Yes	No
Performance	Slower	Faster
Atomicity	Yes	No
Use case	Protect critical section	Shared flags or status only

🟦 Synchronized Method:
```
synchronized void method() {
  // only one thread at a time
}
```

🟦 Synchronized Block:
```
synchronized(obj) {
  // critical section
}
```

🟦 Static Synchronization:
```
synchronized static void method() {
  // lock on class object
}
```

🟦 ReentrantLock (Advanced):
```
Lock lock = new ReentrantLock();
lock.lock();
try {
  // critical section
} finally {
  lock.unlock();
}
```

==========

9. VOLATILE KEYWORD
==========

- Ensures visibility of variable changes across threads.
- Prevents caching of variable in thread-local memory.

Example:
volatile boolean flag = true;

Note:
Does not guarantee atomicity. Use `AtomicInteger` for atomic updates.

==========
10. INTER-THREAD COMMUNICATION
==========

- Achieved using wait(), notify(), notifyAll()
- Used to coordinate threads

Example:
synchronized(obj) {
  obj.wait(); // releases lock and waits
  obj.notify(); // wakes up one thread
}

Must be called within synchronized block.

==========
11. DAEMON THREAD - Helper thread
==========

- Background thread that dies when all user threads finish.
- Use for logging, cleanup, GC, etc.

Example:
Thread t = new Thread(() -> {/* task */});
t.setDaemon(true);
t.start();

Note: Must call setDaemon(true) before start().

```java
package multithreading.injava;

class Virat extends Thread{
	@Override
	public void run() {

		BatCoach b = new BatCoach();
		BowlCoach bo = new BowlCoach();
		b.setName("Ba");
		bo.setName("Bo");

		//This setDaemon(true) method makes the threads daemon when the argument is true and if
its said false it will no longer be daemon thread
		b.setDaemon(true);
		bo.setDaemon(true);

		//To check whether the thread is daemon thread or not
		System.out.println(b.isDaemon()); //true
		System.out.println(bo.isDaemon()); //true if they are daemon threads

		b.start();
		bo.start();

		try {
			System.out.println("Enters ground");
			Thread.sleep(2000);
```

```
                System.out.println("Do warmup");
                Thread.sleep(2000);
                System.out.println("Fielding");
                Thread.sleep(2000);
                System.out.println("bowling");
                Thread.sleep(2000);
                System.out.println("Batting");
                System.out.println("Leave the ground");
            }catch(Exception e) {
                e.printStackTrace();
            }
        }
}

class BatCoach extends Thread{
        @Override
        public void run() {
            for(; ;) {
                try {
                    System.out.println("Batting coach in the ground");
                    Thread.sleep(2000);
                }catch(Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        }
}

class BowlCoach extends Thread{
        @Override
        public void run() {
            for(; ;) {
                try {
                    System.out.println("Bowling coach in the ground");
                    Thread.sleep(2000);
                }catch(Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        }
}

public class DaemonThread {

        public static void main(String[] args) {

//          Daemon thread - Helper Thread for primary thread.. Eg: auto save in word which will do
its work until wprd app is open

            //Identify daemon and primary thread, we want Thread-1 and 2 to execute only till primary
thread i.e thread-0 is executing

            /* Rule :
             * 1. Identify daemon threads
             * 2. Primary thread should be the one to create the daemon threads - Create Objects of
daemon threads and start them inside run method of primary thread as above
             * 3. Primary thread should use setDaemon() which takes boolean value as argument(set to
true) and create Thread
             * 4. Must call setDaemon(true) before start()
             * So now the daemon threads execute until primary thread is executing
             * */

            Virat v = new Virat();
            v.setName("V");
            v.start();



        }

}
```

==========
12. THREAD GROUP
==========

- Allows managing multiple threads together.

```
ThreadGroup group = new ThreadGroup("MyGroup");
Thread t1 = new Thread(group, new MyRunnable());
```

Useful for organizing threads hierarchically.

==========
13. THREAD POOL (Executor Framework)
==========

- Reuses fixed number of threads.
- Improves performance by avoiding thread creation overhead.

Types:
- newFixedThreadPool(n)
- newCachedThreadPool()
- newSingleThreadExecutor()
- newScheduledThreadPool()

Example:

```
ExecutorService ex = Executors.newFixedThreadPool(5);
ex.execute(new MyRunnable());
ex.shutdown();
```

==========
14. JAVA CONCURRENCY PACKAGE (java.util.concurrent)
==========

Main Components:
- ExecutorService: Manages thread pool
- Callable<V>: Task with return
- Future<V>: Result of Callable
- ScheduledExecutorService: Schedule tasks
- CountDownLatch: Wait for n threads to finish
- CyclicBarrier: Wait until all threads reach a point
- Semaphore: Limit concurrent access
- ReentrantLock: Advanced locking
- ReadWriteLock: Separate read/write locks
- BlockingQueue: Producer-consumer support

Atomic Variables:
- AtomicInteger, AtomicBoolean, AtomicReference
→ Thread-safe, lock-free updates.

==========
15. CONCURRENT COLLECTIONS
==========

Thread-safe versions of standard collections:
- ConcurrentHashMap
- CopyOnWriteArrayList
- ConcurrentLinkedQueue
- LinkedBlockingQueue

Note: Avoid Vector, Hashtable in modern code. Use java.util.concurrent instead.

==========
16. CALLABLE VS RUNNABLE
==========

Feature	Runnable	Callable
Return Value	No	Yes
Exception	No checked	Can throw checked
Method	run()	call()
Used With	Thread	ExecutorService

==========
17. COMMON THREAD EXCEPTIONS
==========

- IllegalThreadStateException: Calling start() twice
- InterruptedException: Thread interrupted while waiting/sleeping
- RejectedExecutionException: Executor can't accept new task
- ConcurrentModificationException: Accessing collection while modifying without sync

==========
18. THREAD LOCAL STORAGE
==========

- ThreadLocal<T> → Stores variable local to each thread.

```
ThreadLocal<Integer> threadId = ThreadLocal.withInitial(() -> 0);
threadId.set(100);  // visible only to current thread
```

Useful for user session, transactions, logs.

==========
19. FORK/JOIN FRAMEWORK (Java 7+)
==========

- Splits large task into subtasks, runs in parallel.

Example:

```
class MyTask extends RecursiveTask<Integer> {
   protected Integer compute() {
      if (task is small) return result;
      else {
         MyTask t1 = new MyTask(...);
         MyTask t2 = new MyTask(...);
         t1.fork();
         int result2 = t2.compute();
         int result1 = t1.join();
         return result1 + result2;
      }
   }
}

ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new MyTask());
```

==========
20. BEST PRACTICES FOR MULTITHREADING
==========

✅ Use ExecutorService instead of manually creating threads.
✅ Minimize shared mutable state.
✅ Prefer immutable and stateless objects.
✅ Use high-level concurrency APIs.
✅ Handle InterruptedException properly.
✅ Always shutdown executor services.
✅ Avoid busy-wait loops; use wait/notify or blocking queues.

✅ Profile and debug carefully — multithreading issues are hard to reproduce.

==========

==========
🔹 SEMAPHORE & MONITOR
==========
🟦 SEMAPHORE
✅ Definition:
A semaphore is a synchronization aid that controls access to a shared resource through the use of a counter.

It allows a limited number of threads to access a resource at the same time.

Introduced in Java via java.util.concurrent.Semaphore.

✅ Working:
Semaphore has a permit counter.

A thread must acquire a permit before accessing the resource.

When done, it releases the permit.

✅ Types:
Binary Semaphore (mutex) – Only 1 permit → acts like a lock.

Counting Semaphore – Allows multiple permits.

✅ Constructor:
java
Copy
Edit
Semaphore s = new Semaphore(int permits);
✅ Common Methods:
acquire() – Acquires a permit (waits if not available).

release() – Releases a permit (increments counter).

tryAcquire() – Acquires permit if available, else returns false.

✅ Example:
java
Copy
Edit
Semaphore sem = new Semaphore(2); // max 2 threads allowed at once

sem.acquire(); // get permit (blocks if 0)
try {
  // critical section
} finally {
  sem.release(); // release permit
}
✅ Use Case:
Controlling access to a fixed-size pool (DB connections, printers).

Rate limiting (e.g., allow 5 API calls/sec).

🟩 MONITOR
✅ Definition:
A monitor is a logical entity (or mechanism) that combines:

Mutual exclusion (only one thread in a block at a time),

Inter-thread communication (wait/notify).

In Java, every object is associated with a monitor.

✅ How it works:
When a thread enters a synchronized method/block, it acquires the object's monitor (i.e., lock).

Other threads trying to enter synchronized blocks on the same object must wait until the monitor is released.

✅ Synchronized Code = Monitor-based:
java
Copy
Edit
```
synchronized(obj) {
  // this block is monitored - only one thread can access it
}
```
✅ Inter-thread Communication (with Monitor):
java
Copy
Edit
```
synchronized(obj) {
  obj.wait();     // releases monitor & waits
  obj.notify();   // wakes up one waiting thread
  obj.notifyAll(); // wakes up all waiting threads
}
```
✅ Key Points:
Monitors are implicitly used with synchronized.

Every object in Java has a monitor lock.

Used for both mutual exclusion and coordination.

✅ Use Case:
Coordinating producer-consumer threads.

Ensuring one-at-a-time access to shared code/resources.

🔁 DIFFERENCE: Semaphore vs Monitor

Feature	Semaphore	Monitor (synchronized)
Purpose	Limit access to shared resource (N threads)	Mutual exclusion + communication
Control	Allows N threads at once (permits)	Allows only 1 thread (lock)
Communication	No built-in wait/notify	Supports wait(), notify(), notifyAll()
Explicit or Implicit	Explicit object (java.util.concurrent)	Implicit (via synchronized)
Java support	java.util.concurrent.Semaphore	Built into the language (all Objects)
Use Case	Resource pool, throttling	Safe shared access, coordination

==========

THREADS – COMPLETE INTERVIEW Q&A

1. Theory Questions with Answers

What is a thread in Java?

- A thread is a lightweight subprocess, the smallest unit of execution in Java. Multiple threads can run concurrently within a single process.

Difference between Process and Thread?

- Process: Independent execution unit with its own memory.
- Thread: Shares process memory, lightweight, faster to create and switch.

Ways to create a thread in Java?

- Extend Thread class and override run()
- Implement Runnable interface and pass to Thread constructor
- Implement Callable with FutureTask for return values.

Thread lifecycle states?

- NEW → RUNNABLE → RUNNING → WAITING / TIMED_WAITING / BLOCKED → TERMINATED.

Difference between start() and run()?
- start() → Creates a new thread and calls run() asynchronously.
- run() → Executes in the current thread, no new thread created.

Difference between sleep() and wait()?
- sleep(ms) → From Thread class, pauses execution for specified time, doesn't release lock.
- wait(ms) → From Object class, releases lock and waits until notified or timeout.

Explain wait(), wait(long timeout), wait(long timeout, int nanos)
- wait() → Waits indefinitely until notified.
- wait(timeout) → Waits for specified milliseconds or until notified.
- wait(timeout, nanos) → Waits for specified time + nanoseconds.

Explain notify() vs notifyAll()
- notify() → Wakes up one random waiting thread.
- notifyAll() → Wakes up all waiting threads.

Why wait(), notify() are in Object class, not Thread class?
- Because they act on the monitor lock of an object, and every object in Java can be used as a lock.

What is thread priority?
- Integer value from 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY), default is 5 (NORM_PRIORITY).

What is daemon thread?
- A thread that runs in background to support user threads, JVM exits when only daemon threads remain.

What is thread safety?
- When multiple threads access shared resources without data inconsistency.

How to achieve thread safety?
- Synchronization, volatile, Atomic classes, Locks, ThreadLocal.

What is join() method?
- Makes calling thread wait until the target thread completes.

Difference between synchronized method and synchronized block?
- Method → Locks whole method.
- Block → Locks only a specific section.

What is yield() method?
- Suggests scheduler to give other threads a chance to execute (not guaranteed).

What is deadlock?
- Two or more threads waiting on each other's locks indefinitely.

How to avoid deadlock?
- Lock ordering, timeout with tryLock(), minimal lock scope.

What is starvation?
- A thread waits indefinitely because other threads monopolize resources.

What is race condition?
- Incorrect result due to unsynchronized access to shared data.

2. Coding Questions with Answers

Q1: Create and start a thread using both ways.

java
CopyEdit
```
// Method 1: Extend Thread
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread via Thread class");
   }
}
new MyThread().start();
// Method 2: Implement Runnable
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Thread via Runnable interface");
   }
}
new Thread(new MyRunnable()).start();
```

Q2: Demonstrate wait() and notify().

java
CopyEdit

```
class Shared {
   synchronized void print() {
      try {
         wait(); // releases lock
         System.out.println("Resumed after notify");
      } catch (InterruptedException e) {}
   }
   synchronized void resumeThread() {
      notify();
   }
}
```

Q3: Deadlock example.

```
java
CopyEdit
class A { synchronized void methodA(B b) { b.last(); } synchronized void last() {} }
class B { synchronized void methodB(A a) { a.last(); } synchronized void last() {} }
```

3. MCQs with Answers

wait() belongs to which class?
a) Thread
b) Object
Ans: b) Object
sleep() releases lock?
a) Yes
b) No
Ans: b) No
Default thread priority in Java?
a) 1
b) 5
c) 10
Ans: b) 5
Which method ends a thread?
a) stop()
b) run() return
Ans: b) run() return
Which is not a valid thread state?
a) RUNNABLE
b) WAITING
c) SUSPENDED
Ans: c) SUSPENDED (Deprecated)

4. Common Advanced Questions with Short Answers

- **Why use Callable over Runnable?** → Callable returns a result and can throw checked exceptions.
- **What is volatile keyword?** → Ensures visibility of changes across threads.
- **What is ThreadLocal?** → Provides thread-confined variables.
- **How to stop a thread safely?** → Use a boolean flag checked in run().
- **What is Executor Framework?** → High-level API to manage thread pools (ExecutorService).

Java Wrapper Classes

1. Introduction

- **Wrapper classes** in Java are **object representations** of primitive data types.
- They “wrap” a primitive value into an **object** so it can be treated like an object.
- Located in **java.lang** package.

2. Why Wrapper Classes Are Needed

Collections API requirement – Collections (e.g., ArrayList, HashMap) can store only objects, not primitives.
Utility methods – Wrapper classes have many utility methods (e.g., parsing strings to numbers).
Type conversion – Converting between different numeric types.
Synchronization – Objects can be synchronized (primitives cannot).
Serialization – Only objects can be serialized.

3. Primitive vs Wrapper Mapping

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

4. Autoboxing and Unboxing

Autoboxing: Automatic conversion from primitive → wrapper object. Which happens in Collections
Unboxing: Automatic conversion from wrapper object → primitive.

java
CopyEdit

```
// Autoboxing
int num = 10;
Integer obj = num; // primitive to object
// Unboxing
Integer obj2 = 20;
int val = obj2; // object to primitive
```

5. Creating Wrapper Objects

```
Integer i1 = Integer.valueOf(25); // Using valueOf() – preferred
Integer i2 = new Integer(25); // Deprecated since Java 9
Integer i3 = 25; // Autoboxing
```

6. Important Methods in Wrapper Classes

Common methods:

- valueOf(String s) → returns wrapper object from string.
- parseInt(String s) / parseDouble(String s) → primitive from string.
- toString() → converts to String.
- compareTo() → compares two wrapper objects.
- equals(Object o) → compares content.
- MAX_VALUE, MIN_VALUE → constants for limits.

Example:

java
CopyEdit

```
int num = Integer.parseInt("123"); // String → primitive
Integer obj = Integer.valueOf("456"); // String → object
System.out.println(Integer.MAX_VALUE); // 2147483647
```

7. Memory & Performance Notes

- Wrapper objects are **immutable** (value cannot change after creation).
- Java caches frequently used wrapper objects (Integer from -128 to 127) → called **Integer Cache**.
- Excessive autoboxing/unboxing in loops can hurt performance.

8. Differences – Primitive vs Wrapper

Primitive	Wrapper Object
Faster	Slower (extra memory, object overhead)
Stored in stack	Stored in heap
No null allowed	Can be null
No methods	Has utility methods

9. Common Pitfalls

NullPointerException on unboxing a null wrapper:

```
java
CopyEdit
Integer obj = null;
int x = obj; // Throws NPE
```

Equality confusion:

```
java
CopyEdit
Integer a = 100, b = 100;
System.out.println(a == b); // true (cached)
Integer c = 200, d = 200;
System.out.println(c == d); // false (outside cache range)
```

10. Real-World Uses

- **Storing primitive values in Collections**:

```
java
CopyEdit
List<Integer> list = new ArrayList<>();
list.add(10); // Autoboxing
```

- **Parsing user input**:

```
java
CopyEdit
Scanner sc = new Scanner(System.in);
int age = Integer.parseInt(sc.nextLine());
```

11. Interview Questions

A. Theory Questions

What are wrapper classes in Java? Why are they needed?
Explain autoboxing and unboxing with examples.
Why are wrapper classes immutable?
What is Integer caching? What is its range?
Can we extend wrapper classes?
Difference between parseInt() and valueOf()?

B. Coding Questions

Q1. Convert a String to int without using parseInt():

```
java
CopyEdit
String s = "123";
int num = Integer.valueOf(s); // or new Integer(s)
```

System.out.println(num);
Q2. Demonstrate autoboxing, unboxing, and equality issues:

java
CopyEdit
Integer x = 127, y = 127;
System.out.println(x == y); // true
Integer p = 128, q = 128;
System.out.println(p == q); // false
System.out.println(p.equals(q)); // true
Q3. Write code to check for Integer cache range:

java
CopyEdit
for (int i = -130; i <= 130; i++) {
  Integer a = i, b = i;
  if (a != b) {
    System.out.println("Not cached: " + i);
  }
}

C. MCQs with Answers

Q1. Which statement is true about wrapper classes?
a) They are mutable
b) They can store null
c) They store primitive values directly on stack
d) All are true
Answer: b) They can store null

Q2. Output?

java
CopyEdit
Integer a = 100, b = 100;
System.out.println(a == b);
Integer c = 200, d = 200;
System.out.println(c == d);
Answer:

arduino
CopyEdit
true
false
(First is cached, second is outside cache range)

Q3. What will happen?

java
CopyEdit
Integer obj = null;
int x = obj;
a) 0 will be assigned
b) NullPointerException
c) Compilation error
Answer: b) NullPointerException

Q4. Which method returns a primitive value from String in Integer class?
a) valueOf()
b) parseInt()
c) getInt()
d) toPrimitive()
Answer: b) parseInt()

Q5. Which primitive does Character wrapper represent?
a) String
b) char
c) byte
Answer: b) char