Object:

Definition of an Object in OOP Context

In **Object-Oriented Programming (OOP)**, an **object** is an **entity** that has:

1. **State** → The data (attributes/properties) it holds.
2. **Behavior** → The actions (methods/functions) it can perform.
3. **Identity** → A unique reference that differentiates it from other objects (id, memory address).

Ex: Consider a **real-world example**: A car.

- **State:** Color, speed, brand, model, fuel level, etc.
- **Behavior:** Start, stop, accelerate, brake, etc.
- **Identity:** Each car object is uniquely identified by its number plate, even if two cars have the same attributes.

In programming aspect:

An object is an instance of a class.

What Defines an Object's Identity in OOP?

An **object's identity** is what **uniquely distinguishes it from other objects**, even if they have the same state and behavior.

The identity of an object is **inherent to the object itself** and is not solely determined by its attributes.

Identity in Java (Memory Address Perspective)

In Java, when you create an object, it gets a **unique memory address** in the **heap**.

Even if two objects have **the same state (attribute values) and behavior (methods)**, they are **different objects** because they reside at **different memory locations**.

Class

Conceptually:
A logical entity that represents a group of objects having similar properties and behaviour.

Programming aspect:

A class is a user-defined data type that serves as a blueprint for creating objects, encapsulating data (variables) and behavior (functions) within a single unit.

Method in Java

A **method** in Java is a block of code that performs a specific task. It enhances code reusability and modularity by allowing a piece of logic to be defined once and executed multiple times.

Key Features of Methods:

- **Code Reusability** – Methods prevent code duplication.
- **Modularity** – Breaks complex programs into smaller, manageable parts.
- **Encapsulation** – Can be used to hide internal implementation details.
- **Parameterization** – Accepts arguments to work dynamically.
- **Return Type** – Can return a value or be `void` if no value is returned.

Constructor:

A **constructor** in Java is a special type of method used to initialize objects. It is called when an instance of a class is created. The constructor's primary purpose is to assign initial values to the instance variables of the class.

Key Features of a Constructor:

1. **Same Name as Class** – The constructor must have the same name as the class.
2. **No Return Type** – Unlike methods, constructors do not have a return type (not even `void`).
3. Called Automatically – It is invoked automatically when an object of the class is created.

Types of Constructors in Java

1. Default Constructor (No-Argument Constructor)
2. Parameterized Constructor
3. Copy Constructor

Method vs Constructor in Java

Feature	**Method**	**Constructor**
Purpose	Performs a specific task	Initializes an object

Name	Can have any name	Must have the same name as the class
Return Type	Must have a return type (including `void`)	No return type (not even `void`)
Invocation	Called explicitly using an object	Called automatically when an object is created
Overloading	Supports method overloading	Supports constructor overloading
Inheritance	Can be inherited and overridden	Cannot be inherited but can be called using `super()`
Explicit Call	Called using `objectName.methodName()`	Called using `new ClassName()`

Example: Student Class in Java

```
// Defining a class that represents a group of students with common properties and behavior

class Student {
    // Properties (State)
    String name;
    int age;
    String grade;

    // Constructor to initialize student details
    Student(String name, int age, String grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
    // Behavior (Methods)
    void study() {
        System.out.println(name + " is studying.");
    }

    void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age + ", Grade: " + grade);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating multiple Student objects (instances of the Student class)
        Student student1 = new Student("Rahul", 20, "A");
        Student student2 = new Student("Priya", 19, "B");

        // Calling methods on the objects
        student1.displayInfo();
        student1.study();

        student2.displayInfo();
        student2.study();
    }
}
```

Explanation:

- The `Student` class defines a **group of students** with common properties: `name`, `age`, and `grade`.
- It has behaviors like `study()` and `displayInfo()`, which all students share.
- Multiple student **objects** (`student1`, `student2`) are created, following the **same structure** but having unique identities.

Exercise Programs:

1. Write a program to find the area and perimeter of a given circle.
    a. Using procedural approach
    b. Using oops approach

The four main Object-Oriented Programming (OOP) principles:

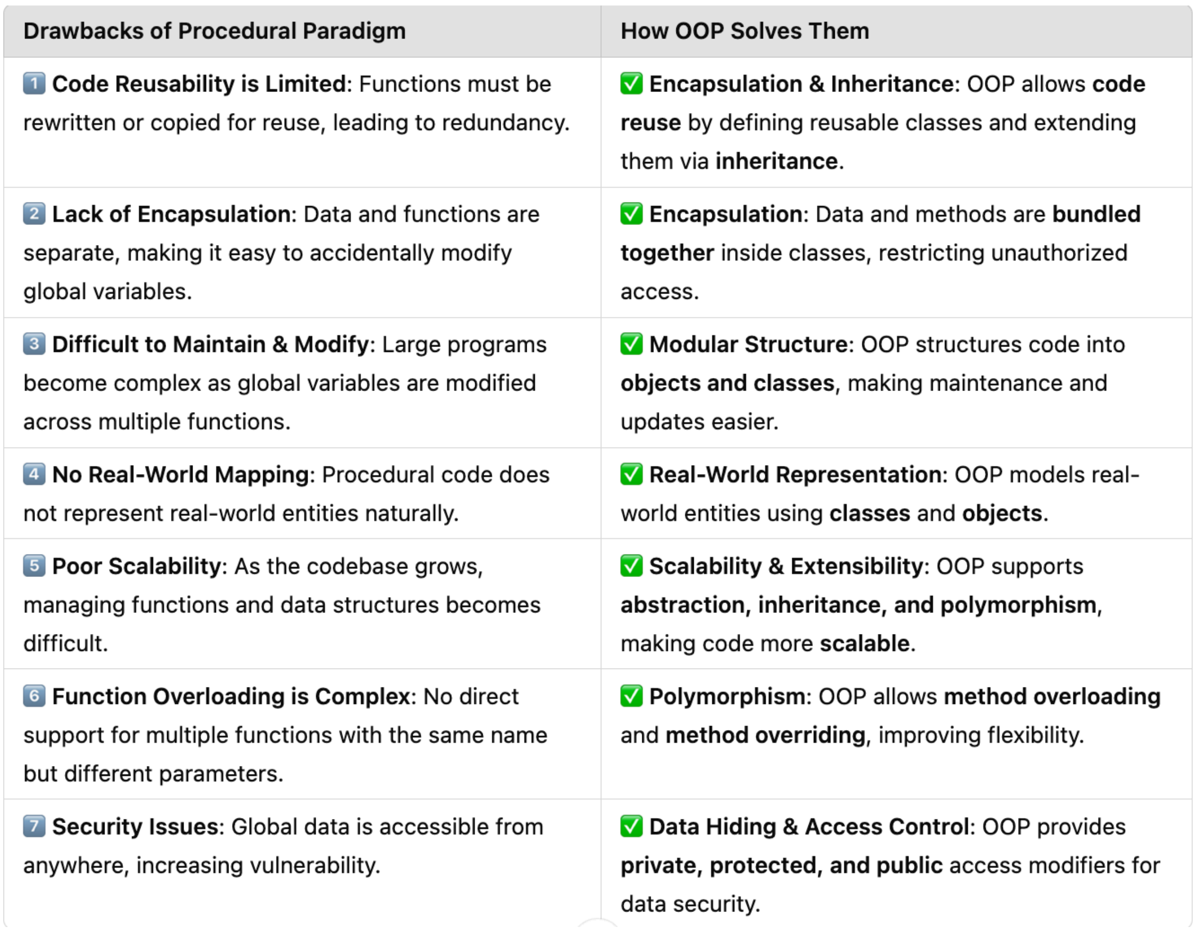
1. **Encapsulation** – Wrapping data (variables) and methods (functions) into a single unit (class) and restricting direct access to some details using access modifiers (`private`, `protected`, `public`).
   - Example: Using getters and setters to control access to private fields.

2. **Abstraction** – Hiding implementation details and exposing only the essential features using abstract classes and interfaces.
   - Example: A `Vehicle` interface with a method `move()`, implemented differently in `Car` and `Bike` classes.

3. **Inheritance** – Acquiring properties and behaviors from an existing class (parent/superclass) into a new class (child/subclass).
   - Example: A `Dog` class inheriting from an `Animal` class.

4. **Polymorphism** – Ability to take multiple forms, allowing methods with the same name to behave differently based on the object.
   - Example: Method overloading (same method name but different parameters) and method overriding (subclass providing a specific implementation of a parent class method).

Drawbacks of the Procedural Paradigm & How OOP Solves Them:



Drawbacks of Procedural Paradigm	**How OOP Solves Them**
1️⃣ **Code Reusability is Limited**: Functions must be rewritten or copied for reuse, leading to redundancy.	✅ **Encapsulation & Inheritance**: OOP allows **code reuse** by defining reusable classes and extending them via **inheritance**.
2️⃣ **Lack of Encapsulation**: Data and functions are separate, making it easy to accidentally modify global variables.	✅ **Encapsulation**: Data and methods are **bundled together** inside classes, restricting unauthorized access.
3️⃣ **Difficult to Maintain & Modify**: Large programs become complex as global variables are modified across multiple functions.	✅ **Modular Structure**: OOP structures code into **objects and classes**, making maintenance and updates easier.
4️⃣ **No Real-World Mapping**: Procedural code does not represent real-world entities naturally.	✅ **Real-World Representation**: OOP models real-world entities using **classes** and **objects**.
5️⃣ **Poor Scalability**: As the codebase grows, managing functions and data structures becomes difficult.	✅ **Scalability & Extensibility**: OOP supports **abstraction, inheritance, and polymorphism**, making code more **scalable**.
6️⃣ **Function Overloading is Complex**: No direct support for multiple functions with the same name but different parameters.	✅ **Polymorphism**: OOP allows **method overloading** and **method overriding**, improving flexibility.
7️⃣ **Security Issues**: Global data is accessible from anywhere, increasing vulnerability.	✅ **Data Hiding & Access Control**: OOP provides **private, protected, and public** access modifiers for data security.

Instance and Class Members in Java:

In Java, **members** refer to variables and methods that belong to a **class**. These members can be categorized into **instance members** and **class (static) members** based on how they are associated with the class and objects.

Instance Members (Object-Level Members)

- Instance members belong to **individual objects** of a class.
- Each object gets its **own copy** of instance variables.
- They are accessed using the **object reference**.
- Declared **without** the `static` keyword.

Ex:

```
class Student {
    String name;  // Instance variable

    int age;      // Instance variable

    void display() { // Instance method

        System.out.println("Name: " + name + ", Age: " + age);

    }

    public static void main(String[] args) {

        Student s1 = new Student();  // Object 1

        s1.name = "Rahul";

        s1.age = 20;

        Student s2 = new Student();  // Object 2

        s2.name = "Priya";

        s2.age = 22;

        s1.display();  // Output: Name: Rahul, Age: 20

        s2.display();  // Output: Name: Priya, Age: 22

    }

}
```

Class Members (Static Members)

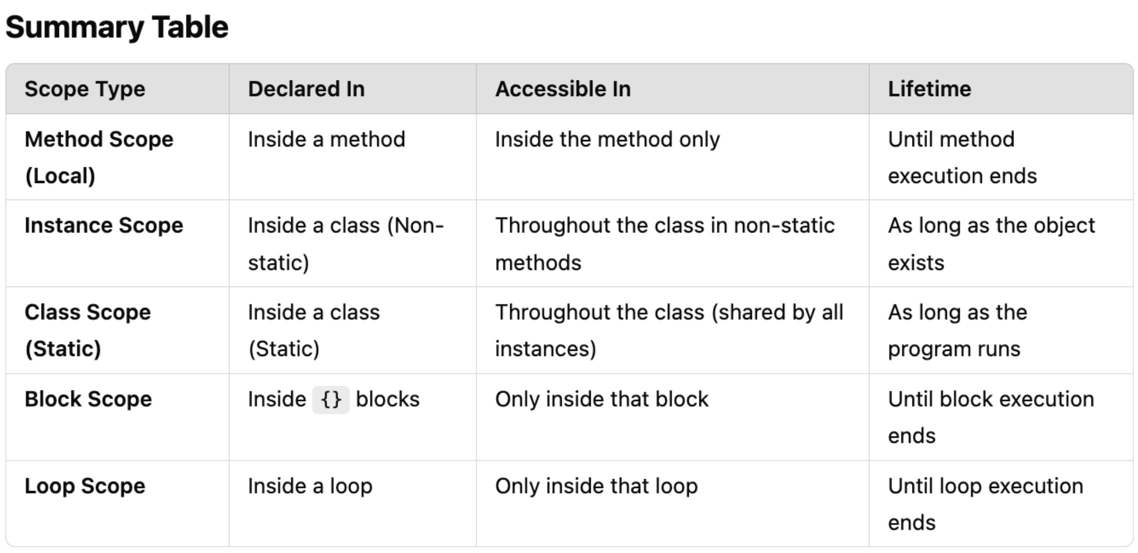
- Class members are **shared** among all objects.
- Declared using the `static` keyword.
- Memory for class members is allocated **once** at class loading time.
- Accessed using the **class name** or an object reference.

```
class Student {
    String name;   // Instance variable
    int age;       // Instance variable
    static String college = "ABC College"; // Static variable

    static void changeCollege() { // Static method
        college = "XYZ College";
    }
    void display() {
        System.out.println("Name: " + name + ", Age: " + age + ",
College: " + college);
    }
    public static void main(String[] args) {
        Student.changeCollege(); // Calling static method
        Student s1 = new Student();
        s1.name = "Rahul";
        s1.age = 20;
        Student s2 = new Student();
        s2.name = "Priya";
        s2.age = 22;
        s1.display();  // Output: Name: Rahul, Age: 20, College: XYZ
College
        s2.display();  // Output: Name: Priya, Age: 22, College: XYZ
College
    }
}
```

Note: instance methods can refer static members but not vice-versa

Scope of a variable and its life times in java:

The scope of a variable is the section/portion of the program where it is visible and can be accessed.



Summary Table

Scope Type	Declared In	Accessible In	Lifetime
Method Scope (Local)	Inside a method	Inside the method only	Until method execution ends
Instance Scope	Inside a class (Non-static)	Throughout the class in non-static methods	As long as the object exists
Class Scope (Static)	Inside a class (Static)	Throughout the class (shared by all instances)	As long as the program runs
Block Scope	Inside `{}` blocks	Only inside that block	Until block execution ends
Loop Scope	Inside a loop	Only inside that loop	Until loop execution ends

Method Scope (Local Scope)

- Variables declared inside a method are local to that method.
- They cannot be accessed outside the method.

- They exist only while the method is executing.

```
class Example {
   void display() {
      int num = 10; // Local variable
      System.out.println(num);
   }

   public static void main(String[] args) {
      Example obj = new Example();
      obj.display();
      // System.out.println(num); // ❌ Error: num is not accessible here
   }
}
```

Instance Scope (Object Scope)

- Instance variables belong to an object.
- They are declared inside a class but outside methods.
- They are accessible within non-static methods of the same class.
- They exist as long as the object exists.

```
class Student {
   String name; // Instance variable

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

   void display() {
      System.out.println("Student Name: " + name);
   }

   public static void main(String[] args) {
      Student s1 = new Student();
      s1.setName("Rahul");
      s1.display(); // Output: Student Name: Rahul
   }
}
```

Class Scope (Static Scope)

- Static variables belong to the class, not an instance.

- They are shared among all instances of the class.
- They are declared using the `static` keyword.

```
class Counter {
    static int count = 0; // Static variable

    Counter() {
        count++; // Increments for every object
    }

    void displayCount() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.displayCount(); // Output: Count: 2
    }
}
```

Block Scope

- Variables declared inside `{}` (block) are limited to that block.
- Mainly used in `if`, `for`, `while`, and `switch` statements.

```
class Example {
    public static void main(String[] args) {
        if (true) {
            int x = 20; // Block scope
            System.out.println(x);
        }
        // System.out.println(x); // ❌ Error: x is not accessible here
    }
}
```

Loop Scope

- Variables declared inside loops are accessible only within the loop.
- Useful for temporary counters or accumulators.

```
class LoopScope {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.print(i + " "); // Output: 1 2 3 4 5
        }
        // System.out.println(i); // ❌ Error: i is not accessible here
    }
}
```

Access Modifier Vs Specifier:

Java Official Term: "Access Modifiers"

officially, Java has only "Access Modifiers"—the term **"Access Specifier" is unofficial and not defined in the Java Language Specification (JLS).**

"Access Specifier" Is an Unofficial Term

- Many books and tutorials use the term **"Access Specifier"**, but **the Java Language Specification (JLS) does not mention it**.
- Some people **differentiate** between "Access Modifiers" and "Access Specifiers" by saying:
    - **Access Modifiers** → Control scope (e.g., `public`, `private`, `protected`).
    - **Access Specifiers** → A broader category, including **other modifiers like `static`, `final`, `synchronized`, etc.**

However, **this distinction is unofficial** and **not used in the JLS**.

The Java Language Specification (JLS) defines **four access control levels**, which are referred to as **access modifiers** in official documentation:

1. **`public`** – Accessible from anywhere.
2. **`protected`** – Accessible within the same package and subclasses.
3. **(default / package-private)** – Accessible only within the same package.
4. **`private`** – Accessible only within the same class.

Getters and Setters in Java

Definition

- **Getters and setters** are **methods** used to **access (get) and modify (set) private fields** of a class.
- They follow the **principle of encapsulation**, ensuring **controlled access** to class attributes.

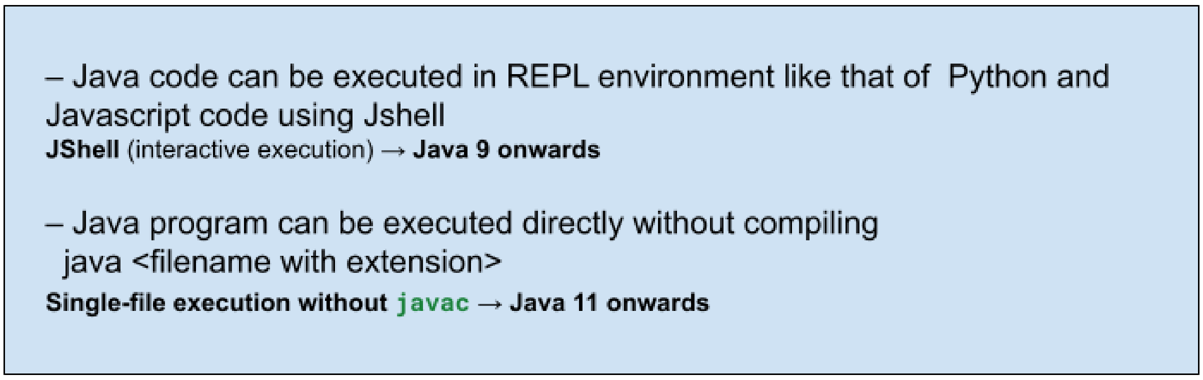
Why Use Getters and Setters?

1. **Encapsulation** → Keeps data hidden (`private`) and controls modification.
2. **Validation** → Allows adding logic to check values before setting them.
3. **Read-Only or Write-Only Fields** → Can allow only getting or setting if needed.
4. **Flexibility** → Can modify the internal logic without affecting users of the class.

```
class Student {
    private String name;  // Private field (Encapsulation)
    private int age;

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method with validation
    public void setAge(int age) {
        if (age > 0) { // Ensuring valid age
            this.age = age;
        } else {
            System.out.println("Age must be positive!");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Student student = new Student();

        // Setting values
        student.setName("Rahul");
        student.setAge(20);

        // Getting values
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());
    }
}
```

Few points to note with java new features:



– Java code can be executed in REPL environment like that of Python and Javascript code using Jshell
JShell (interactive execution) → **Java 9 onwards**

– Java program can be executed directly without compiling
  java <filename with extension>
Single-file execution without `javac` → Java 11 onwards

Memory management system in Java

Managed by the Java Virtual Machine (JVM).

Memory management system is divided into several key areas, each serving a specific purpose.

JVM memory structure in terms of the **heap**, **stack**, **method area**(**metaspace)**, and **runtime constant pool**, along with how different types of data are stored:

JVM Specification vs. Implementation Details

- **The JVM Specification gives a high-level, conceptual view** (e.g., Method Area, Runtime Constant Pool).

- **JVM Implementations (HotSpot, OpenJ9, etc.) may introduce additional optimizations** (e.g., Metaspace, SCP).

Thus, while the specification does not define "String Constant Pool" or "Metaspace" as official terms, these are widely used to describe implementation details in specific JVM versions.

Note: The String Pool is a subset of the Runtime Constant Pool.

- The String Pool specifically manages `String` literals to optimize memory usage by reusing identical `String` objects.
- The Runtime Constant Pool, on the other hand, contains a broader range of constants beyond just `String` literals.

JVM Memory Areas

Memory Area	Shared or Per Thread?	Purpose
Method Area	Shared	Stores class metadata, static variables, and runtime constant pool.
Heap	Shared	Stores objects and arrays (instance variables).
JVM Stack	Per Thread	Stores method call frames (local variables, operand stack, etc.).
Native Method Stack	Per Thread	Supports native method execution (non-Java code).
PC Register	Per Thread	Tracks the address of the currently executing JVM instruction.

What Goes into Heap?

The **Heap** is the primary memory area for storing objects and their associated data. It is managed by the **Garbage Collector (GC)**

Category	Stored in Heap?	Additional Info

Object Instances	✅ Yes	All objects created with `new`
Instance Variables	✅ Yes	Stored inside the object
Static Variables	❌ No	Stored in Metaspace, but referenced objects go to Heap
Objects Referenced by Static Variables	✅ Yes	The reference is in Metaspace, but the object is in Heap
Arrays & Collections	✅ Yes	Lists, Maps, Sets, etc.
Strings (`new String()`)	✅ Yes	Unlike literals, stored in heap

What Goes into Stack Memory ?

The **Stack Memory** is a part of **JVM Memory** that is used for **method execution and local variable storage**. Each thread in Java gets its **own stack**, making it **thread-safe** and **automatically managed** (Last-In, First-Out - LIFO).

Category	Stored in Stack?	Additional Info
Method Call Frames	✅ Yes	Created for each method call, removed after execution
Local Variables	✅ Yes	Includes primitives & object references
Method Parameters	✅ Yes	Stored inside the method's stack frame
Return Values	✅ Yes	Stored temporarily before assignment
Static Variables	❌ No	Stored in **Metaspace**
Objects	❌ No	Stored in **Heap**, but their references are in Stack

What Goes into Method Area (Metaspace)?

JVM stores various pieces of **class metadata**, which are essential for executing Java programs

Category	Stored in Metaspace?	Additional Info

Class Metadata	✅ Yes	Class name, hierarchy, modifiers
Static Variables	✅ Yes (Reference)	Primitive values inside Metaspace, objects in Heap
Instance Variables	❌ No	Stored in Heap with objects
Method Definitions	✅ Yes	Includes signatures, return types
Method Bytecode	✅ Yes	Used by JIT Compiler
Constant Pool	✅ Yes	String literals, method/field references
Static Blocks	✅ Yes	Initialization bytecode

What Goes into Runtime Constant Pool (RCP)?

The **Runtime Constant Pool (RCP)** is part of **Metaspace** (earlier, it was in the **PermGen** before Java 8). It is a **per-class structure** that stores **constants and symbolic references** needed during execution.

Note:

- The String Constant Pool is a subset of the Runtime Constant Pool.
- The String Constant Pool specifically manages `String` literals to optimize memory usage by reusing identical `String` objects.
- The Runtime Constant Pool, on the other hand, contains a broader range of constants beyond just `String` literals.

Stored in RCP

Type	Example	Stored in RCP?	Why?
String Literals	`"Hello"`	✅ **Yes**	String literals are stored in the **String Constant Pool (SCP)**, which is inside RCP.
Final Constants (`static final`)	`static final int A = 10;`	✅ **Yes**	`A` is replaced with `10` at compile time, and `10` is stored in RCP.

Final Strings (`static final`)	`static final String S = "World";`	✅ **Yes**	`"World"` is in **String Constant Pool (SCP)**, and `S` is replaced with `"World"`.
Class References	`Example.class`	✅ **Yes**	Class metadata references are stored in RCP.
Method References	`someMethod()`	✅ **Yes**	Method references (symbolic links) are stored in RCP.

NOT Stored in RCP

Type	Example	Stored in RCP?	Where Stored Instead?
Instance Variables	`int a = 10;`	❌ **No**	**Heap (inside object)**
Local Variables	`int x = 20;`	❌ **No**	**Stack (inside method frame)**
Object Instances	`new Example();`	❌ **No**	**Heap**
Static Variables (Non-Final)	`static int b = 30;`	❌ **No**	**Metaspace (Class metadata area)**

Summary of Data Storage:

- **Instance Variables**: Stored in the heap within their respective objects.
- **Objects**: Stored in the heap.
- **Local Variables**: Stored in the stack.
- **Static Data**: Stored in metaspace.
- **Final instance Data**: stored in the heap
- **Final Static Data**: Stored in RCP (Runtime Constant Pool).
- **Class Information**: Stored in metaspace.

Key Points:

- **Heap**: For objects and instance variables; managed by GC.
- **Stack**: For local variables and method frames; thread-specific.
- **Method Area(Metaspace)**: For class metadata, method metadata, static variables, and byte code
- **Runtime Constant Pool**: For class constants, String Constant Pool, and symbolic references(class reference, method reference) part of metaspace.

Example:

```
public class MemoryExample {
  // Static variable (stored in metaspace)
  private static int staticVar = 10;

  // Final static variable (stored in metaspace)
  private static final String FINAL_STATIC_VAR = "Hello, World!";

  // Instance variable (stored in the heap)
  private int instanceVar;

  public MemoryExample(int instanceVar) {
    this.instanceVar = instanceVar;
  }

  public void printValues() {
    // Local variable (stored in the stack)
    int localVar = 20;
    System.out.println("Static Variable: " + staticVar);
    System.out.println("Final Static Variable: " + FINAL_STATIC_VAR);
    System.out.println("Instance Variable: " + instanceVar);
    System.out.println("Local Variable: " + localVar);
  }

  public static void main(String[] args) {
    // Object creation (stored in the heap)
    MemoryExample obj = new MemoryExample(30);

    // Method call (creates a stack frame)
    obj.printValues();
  }
}
```

Exercises

1. Few common Concepts
    a) Memory management for the Circle application
    b) Read only and write only attributes
    c) Pojo vs Bean

    POJO (Plain Old Java Object)

    A **POJO** is a simple Java class with **no restrictions**.

    Characteristics:

    i) **No specific rules** (can have any variables, methods, and constructors).
    ii) **No need for getter/setter methods** (can use public fields).
    iii) **Can have public, private, or protected fields.**
    iv) **Can use any constructor** (default or parameterized).
    v) **Does NOT require serialization**.

    A **JavaBean** follows specific conventions to ensure **reusability and encapsulation**.

    Characteristics:

    - **Must have private fields** (encapsulation).
    - **Must have public getters and setters**.
    - **Must have a no-argument constructor**.
    - **Must be serializable (implements `Serializable`)**.

    d) Copy Constructor
    e) What is System, out, print in System.out.print(), how it is working
    f) Static block code that gets executed before the main method starts
    g) Singleton Pattern

- Demonstrate using Circle example

h) toString(), hashCode(), equals()

These three methods are **inherited from `java.lang.Object`** and play a key role in handling objects efficiently.

What is `hashCode()` in Java?

- A **hash code** is a numeric value (an `int`) that represents an object
- **helps in faster lookups and comparisons** in hash-based collections like `HashMap`, `HashSet`, and `HashTable`.

Key Points About hashCode()

- **It returns an integer value** (can be positive, negative, or zero).
- **It is not unique** → Different objects can have the same hash code (**hash collision**).
- **If `a.equals(b) == true`, then `a.hashCode() == b.hashCode()`** (to ensure correct behavior in collections).

What is a Hash Collision?

A **hash collision** occurs when **two different objects have the same `hashCode()`** value, even though they are not equal according to `equals()`.

2. Problem Statement to demonstrate **Encapsulation**:

   Develop a **Bank Application in Java** to demonstrate **Encapsulation** while maintaining a bank account. The application should allow users to create a bank account, check their balance, deposit money, and withdraw money while ensuring data security using encapsulation principles.

Phase 1: Requirements Analysis & Design

1. **Define the requirements**:
   - The application should have a `BankAccount` class with private attributes for account details.
   - Provide methods to deposit, withdraw, and check the balance.
   - Ensure proper validation for deposits and withdrawals (e.g., no negative deposits, insufficient funds).
2. **Identify core functionalities**:
   - Account creation

    - Encapsulation of account details (e.g., balance should not be directly modified)
    - Secure access to balance via getter method
    - Deposit and withdrawal functionalities

Phase 2: Implementation

3. **Create the `BankAccount` class**:
    - Implement private fields: `accountNumber`, `accountHolderName`, `balance`
    - Provide public getter method for balance (`getBalance()`)
    - Implement public methods `deposit(double amount)` and `withdraw(double amount)`
    - Ensure deposit and withdrawal methods validate input values
4. **Create a `BankApplication` class** (Main class to interact with `BankAccount`)
    - Provide a menu-driven interface using a loop
    - Allow users to:
        - Create a new account
        - Check account balance
        - Deposit money
        - Withdraw money

Inheritance:

Inheritance in Java

Definition:

Inheritance is an **Object-Oriented Programming (OOP) concept** that allows one class (**child/subclass**) to inherit the properties and behavior of another class (**parent/superclass**). This promotes **code reusability** and establishes a natural **hierarchical relationship** between classes.

Key Features:

Code Reusability – The child class reuses the functionality of the parent class.
Method Overriding – Allows a subclass to provide a specific implementation of a method from the parent class.
Extensibility – Enables the creation of specialized classes based on existing ones.

Reduces Redundancy – Avoids code duplication by defining common logic in a base class.

Types of Inheritance in Java:

Single Inheritance – One class inherits from another.
Multilevel Inheritance – A class inherits from another class, which itself is inherited from another.
Hierarchical Inheritance – Multiple classes inherit from the same parent class.
Multiple Inheritance (via Interfaces) – Java does not support multiple inheritance with classes but allows it with **interfaces**.

Ex:

```
// Parent class
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
   }
}

// Child class inheriting from Animal
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks!");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.eat();  // Inherited method
      dog.bark(); // Own method
   }
}
```

Real-World Analogies for Is-A and Has-A Relationships

1. Is-A Relationship (Inheritance) Example: Animal & Dog

- **Analogy:**
  A **Dog** is a type of **Animal**.
    - All dogs are animals, but not all animals are dogs.

    - A **Dog** inherits characteristics from an Animal (like breathing, eating).

2. Has-A Relationship (Composition) Example: Car & Engine

- **Analogy:**
  A **Car** has an **Engine**.
    - A car is not an engine, but it contains one.
    - The engine can exist independently and can be replaced in a car.

Composition vs. Aggregation in Has-A Relationships

Both **composition** and **aggregation** are forms of the **has-a** relationship in Object-Oriented Programming (OOP). However, they differ in how strongly objects are related.

1. Composition (Strong Has-A Relationship)

- **Definition**:
    - It represents a **strong** relationship where the **contained object (part) cannot exist independently** of the container (whole).
    - If the **whole** object is destroyed, the **part** object is also destroyed.
- **Real-World Example: Car and Engine**
    - An **Engine** is a **part of** a **Car**.
    - If the car is destroyed, the engine is useless (it cannot function on its own).

2. Aggregation (Weak Has-A Relationship)

- **Definition**:
    - It represents a **weak** relationship where the **contained object (part) can exist independently** of the container (whole).
    - If the **whole** is destroyed, the **part** still exists.
- **Real-World Example: Student and College**
    - A **Student** belongs to a **College**, but the **Student can exist even if the College closes**.
    - The **College** contains **multiple Students**, but each Student has their own existence.

Super keyword:

The `super` keyword in Java is used to **refer to the parent class (superclass)** from a subclass. It helps in accessing the parent's methods, constructors, and variables.

In newer versions of Java, `super` **as a method or variable reference** can be used anywhere in the subclass. However, when calling a **superclass constructor**, `super()` must still be the **first statement** inside a constructor.

1. **`super` for Methods and Variables**
   - Can be used **anywhere** inside a method in a subclass.

```
class Parent {
    void display() { System.out.println("Parent method"); }
}
class Child extends Parent {
    void show() {
        System.out.println("Child method");
        super.display(); // `super` used as a next statement (not the first)
    }
}
```

2. **`super` for Constructor Calls**
   - `super();` **must be the first statement** if calling the parent class constructor.

```
class Parent {
    Parent() { System.out.println("Parent constructor"); }
}
class Child extends Parent {
    Child() {
        // Some code here would cause an error
before super()

        super();  // Must be the first statement

        System.out.println("Child constructor");

    }

}
```

Note:

- **For methods and variables** → `super` can be used anywhere in the subclass.
- **For constructors** → `super();` **must** be the first statement.

Exercise :

Problem Statement

Develop a **Java Application** to demonstrate **Inheritance** by creating a simple **Banking System**. The application should have a base class representing a generic bank account and derived classes for specific account types (e.g., **SavingsAccount** and **CurrentAccount**). Each account type should inherit common properties and methods while having its own unique behavior.

Task List:

Phase 1: Understanding & Design

1. **Define requirements**:
   - Implement a **base class (`BankAccount`)** with common properties and methods.
   - Create two subclasses:
     - **SavingsAccount** (withdrawal restricted to balance).
     - **CurrentAccount** (withdrawal allowed beyond balance within an overdraft limit).
   - Demonstrate **runtime polymorphism** using method overriding.
2. **Identify core functionalities**:
   - Account creation.
   - Deposit and withdrawal.
   - Different withdrawal behaviors based on account type.
   - Display account details.

Phase 2: Implementation

3. **Create `BankAccount` (Base Class)**:
   - Define `accountNumber`, `accountHolderName`, `balance`.
   - Implement common methods:
     - `deposit(double amount)`.
     - `withdraw(double amount)`. *(To be overridden in subclasses)*.
     - `displayAccountDetails()`.
4. **Create `SavingsAccount` (Subclass)**:
   - Inherits from `BankAccount`.
   - **Overrides** `withdraw(double amount)` to prevent overdrafts (withdrawals limited to balance).
5. **Create `CurrentAccount` (Subclass)**:
   - Inherits from `BankAccount`.
   - **Overrides** `withdraw(double amount)` to allow overdraft up to a limit.

Which Members Are Inherited from Parent to Child in Java? 🚀

In Java, when a **class (child/subclass) extends another class (parent/superclass)**, certain members (fields, methods) are **inherited**, while others are **not**. Let's break it down.

Members That Are Inherited

Member Type	Inherited ?	Notes
Public Methods	✅ Yes	Directly accessible in child class.
Protected Methods	✅ Yes	Accessible within the package and by subclasses.
Public Fields	✅ Yes	Directly accessible.

Protected Fields	✅ Yes	Directly accessible in the child class.
Default (Package-Private) Methods & Fields	✅ Yes	**Only if child is in the same package.**

📌 **Example of Inherited Members:**

```java
class Parent {
    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30;  // Package-private (default)

    public void publicMethod() {
        System.out.println("Public method in Parent");
    }

    protected void protectedMethod() {
        System.out.println("Protected method in Parent");
    }

    void defaultMethod() {  // Package-private
        System.out.println("Default method in Parent");
    }
}

class Child extends Parent {
    void display() {
        System.out.println(publicVar);    // ✅ Accessible
        System.out.println(protectedVar); // ✅ Accessible
        // System.out.println(defaultVar); // ❌ Not accessible if
in a different package

        publicMethod();    // ✅ Accessible
        protectedMethod(); // ✅ Accessible
        // defaultMethod(); // ❌ Not accessible if in a different
package
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}
```

Key Takeaways

1. Public & Protected members are inherited.
2. Default members are inherited only within the same package.
3. Private members are NOT inherited.
4. Constructors are NOT inherited.
5. Static members belong to the class and are NOT inherited.
6. Final methods are not overridden, but final fields are inherited as constants.

Polymorphism

Polymorphism in Java refers to the ability of an object to take multiple forms, allowing the same method to behave differently based on the actual object type at runtime.

Types of Polymorphism in Java:

1. **Compile-time Polymorphism (Method Overloading)**
    - Achieved by defining multiple methods with the same name but different parameters (method signature).

Example:

```java
class MathUtils {

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

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

}
```

2. **Runtime Polymorphism (Method Overriding)**
    - Achieved when a subclass provides a specific implementation of a method already defined in the parent class.

Example:

```
class Animal {

    void sound() { System.out.println("Animal makes a
sound"); }

}

class Dog extends Animal {

    void sound() { System.out.println("Dog barks"); }

}

public class Test {

    public static void main(String[] args) {

        Animal myPet = new Dog(); // Upcasting

        myPet.sound(); // Calls Dog's sound() method
(runtime binding)

    }

}
```

Key Points:

- **Method Overloading** (Same method name, different parameters) → Compile-time binding.
- **Method Overriding** (Same method signature, different behavior in subclass) → Runtime binding.
  - **Achieved via inheritance and interfaces.**
  - **Upcasting enables dynamic method dispatch.**

Abstraction

Abstraction is the process of **hiding implementation details** and **showing only essential features** to the user. It helps in reducing complexity and increasing code maintainability.

How Abstraction is Achieved in Java?

1. **Using Abstract Classes (`abstract` keyword)**
    - An abstract class can have **both abstract (unimplemented) and concrete (implemented) methods**.
    - It **cannot be instantiated** but can be subclassed.

Example:

```
abstract class Vehicle {

    abstract void start(); // Abstract method (no implementation)

}

class Car extends Vehicle {

    void start() { System.out.println("Car starts with a key"); }

}
```

2. **Using Interfaces (`interface` keyword)**
    - An interface **only contains method declarations** (until Java 8, which introduced default methods).
    - A class must **implement** the interface and provide method definitions.

Example:

```
interface Animal {

    void makeSound(); // Abstract method

}

class Dog implements Animal {

    public void makeSound() { System.out.println("Dog barks"); }

}
```

Key Points

- **Hides implementation details** and provides only necessary functionalities.
- **Abstract classes** can have both abstract and concrete methods, while **interfaces** contain only abstract methods (before Java 8).
- In the newer versions interfaces can have default, private and static methods

- **Encourages loose coupling** in software design.

Exercise :

Objective:

Design a system that models different geometric shapes (`Circle`, `Rectangle`, `Triangle`) using only interfaces. Each shape should implement a method to calculate its area, demonstrating abstraction and polymorphism in Java.

Task List: Geometric Shape Area Calculator (Using Interfaces in Java)

Step 1: Define the Interface (Abstraction)

Create an interface `Shape` with a method `calculateArea()`.

Step 2: Implement Shape Interface in Different Classes

Create a `Circle` class that implements `Shape`.

- Add a `radius` variable.
- Implement `calculateArea()` using the formula: $\pi \times r^2$.

Create a `Rectangle` class that implements `Shape`.

- Add `length` and `width` variables.
- Implement `calculateArea()` using the formula: length × width.

Create a `Triangle` class that implements `Shape`.

- Add `base` and `height` variables.
- Implement `calculateArea()` using the formula: 0.5 × base × height.

Step 3: Demonstrate Polymorphism: In the `main` method, create an array or list of `Shape` objects containing different shape instances (`Circle`, `Rectangle`, `Triangle`).

Interface with Static, Private, and Default Methods in Java

Starting from **Java 8**, interfaces can have **default and static methods**.
From **Java 9**, interfaces can also have **private methods**.

Feature	Introduced in	Purpose
Default Methods	Java 8	Provides default behavior, can be overridden.
Static Methods	Java 8	Belongs to the interface, cannot be overridden.
Private Methods	Java 9	Helps avoid duplication, used only inside the interface.

When to Use These Features?

✔ **Default Methods:** When you want to add new methods to an interface **without breaking existing implementations**.
✔ **Static Methods:** When you need **utility methods** that are related to the interface but don't belong to instances.
✔ **Private Methods:** When you need **common logic** to be used only inside **default/static methods**.

Would you like a real-world example using these concepts? 🚀

Example: Default Method in an Interface

```
interface Vehicle {
    void start(); // Abstract method
    // Default method with implementation
    default void fuelStatus() {
        System.out.println("Fuel status: Full");
    }
}
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car starting...");
    }
    // Overriding default method (optional)
    @Override
    public void fuelStatus() {
        System.out.println("Fuel status: Half");
    }
}
public class DefaultMethodExample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();
        myCar.fuelStatus(); // Calls overridden method in Car
    }
}
```

Points to discuss:

1) How do we achieve encapsulation
2) How do we achieve inheritance and its types in java (using extends, implements)
3) Order of calling constructors in a child class when it is inherited from a parent class
4) Usage of this, super, final, static keywords and their locations while using..

Keyword	**Usage**
`this`	Refers to current object
`this()`	Calls another constructor in same class
`super`	Refers to parent class
`super()`	Calls parent constructor
`final` **variable**	Prevents reassignment (constant)
`final` **method**	Prevents method overriding
`final` **class**	Prevents inheritance
`static` **variable**	Shared across all instances
`static` **method**	Called without object
`static` **block**	Executes before `main()`

5) Types of polymorphism(How it is achieved- at compile time , at runtime)

6) Types of relations between objects

Aggregation	"Has-A" (Weak)	One object contains another, but both can exist independently	`Library` and `Books`
Composition	"Has-A" (Strong)	One object contains another and controls its lifecycle	`Car` and `Engine`
Inheritance	"Is-A"	One class inherits from another	`Dog` and `Animal`

7) Can we overload the parent class methods in child class
8) Can we upcast and downcast any parent and child instance (upcasting: child to parent - implicit, downcasting: parent to child - explicit with appropriate check if the parent variable hold child object)
9) How do we achieve abstraction (abstract, interface)
10) Default access modifiers in interfaces(for abstract methods, variables)
11) Can we upgrade or downgrade the accessibility in extended or implemented classes

Enum in Java (Enumeration)

An **enum** in Java is a special data type used to define a collection of **constant values**. It improves **readability, type safety**, and **code organization** when working with fixed sets of related values.

In Java, an `enum` can also contain **fields, constructors, and methods**, making it more powerful than simple `final static` constants.

Enums can have:

- **Instance variables** (fields)
- **Constructors** (for initialization)
- **Methods** (to define behavior)

Example: Enum with Methods

```java
// Defining an Enum with Fields, Constructor, and Method
enum Color {
    RED("#FF0000"), GREEN("#00FF00"), BLUE("#0000FF");
    private String hexCode; // Field (Instance Variable)
    // Constructor (called automatically when Enum constants are
created)
    Color(String hexCode) {
        this.hexCode = hexCode;
    }
    // Getter Method
    public String getHexCode() {
        return hexCode;
    }
}
public class EnumExample {
    public static void main(String[] args) {
        // Accessing Enum Constant and Its Method
        System.out.println("Color: " + Color.RED);
        System.out.println("Hex Code: " + Color.RED.getHexCode());
    }
}
```

How It Works Behind the Scenes?

Step-by-Step Breakdown:

1. `Color.RED` is an **instance** of `Color`.
2. When `Color.RED` is created, its constructor (`Color(String hexCode)`) is called with `#FF0000`.
3. The `hexCode` is stored in the `hexCode` field of `RED`.
4. When `getHexCode()` is called, it returns the stored hex code.

🔹 **Key Points:**

- Enums are implicitly **final and cannot be extended**.
- Enum constructors are **private** by default (no `public` or `protected`).
- Enum constants behave like **objects** and can store **data** and **behaviors**.

Exercise :

1. Use a switch expression to return the color name with code for a given color constant in an enum

Record in java

- a `record` is a special type of **class** that is designed to be **immutable** and serves as a **data carrier**.
- It eliminates boilerplate code like **constructors, getters, `toString()`, `equals()`, and `hashCode()`**, making Java code cleaner and more concise.
- Introduced in **Java 14**, finalised in **Java 16**

Syntax of a `record`

A record is declared using the `record` keyword:

```java
record Person(String name, int age) {}
```

Equivalent java code without record:

```java
class Person {
    private final String name;
    private final int age;

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

    public String name() { return name; }
    public int age() { return age; }

    @Override public String toString() {
     return "Person[name=" + name + ", age=" + age + "]";
    }
    @Override public boolean equals(Object o) {
      /* Standard equality check */
    }
    @Override public int hashCode() {
      /* Standard hash calculation */
    }
}
```

This automatically provides:

- A **constructor** (`Person(String name, int age)`)
- **Getters** (`name()` and `age()`)
- **`toString()`** (`"Person[name=John, age=25]"`)
- **`equals()` and `hashCode()`**

Benefits of Using `record` Here

- **Eliminates Boilerplate Code**: No need to manually define constructors, getters, `toString()`, `equals()`, and `hashCode()`
- **Encapsulation + Immutability**: Fields are `final`, preventing modification after creation.

Can we extend record with other record or a class:

- No, records in Java cannot extend other records (or any other class) because they are implicitly declared as `final`.
- But, records in Java can implement interfaces

Exercise :

1. Create student record and create two students and check if they are same or not

2. Design a Java program where you create a `Shape` interface that declares a method `calculateArea()`. Implement this interface using **records** for `Circle` and `Rectangle`.
    - The `Circle` record should have a `radius` field and calculate the area as **$\pi \times r^2$**.
    - The `Rectangle` record should have `length` and `breadth` fields and calculate the area as **length × breadth**.
    - Write a `main` method to create instances of both `Circle` and `Rectangle`, and print their areas.

When to Use `record`?

- When you need **simple data-holder classes**
- When you want **immutability**
- When you work with **DTOs, JSON models, database entities (read-only)**
- When used with **pattern matching and sealed classes**

Sealed classes

- Sealed classes in Java provide more control over inheritance.
- They allow a class or interface to specify which other classes or interfaces can extend or implement them.
- Introduced in **Java 15** as a preview feature and finalized in **Java 17**

Key Features of Sealed Classes

1. **Controlled Inheritance:** Only specified classes can extend a sealed class.
2. **Enhances Maintainability:** Prevents unintended subclassing.
3. **Supports Pattern Matching:** Useful in switch expressions with exhaustive cases.

Ex:

```
sealed class Animal permits Dog, Cat {}  // Only Dog and Cat can extend Animal

final class Dog extends Animal {}  // Cannot be extended further
non-sealed class Cat extends Animal {} // Can be extended by any class
```

Allowed Subclass Types

- **`final`** → Cannot have further subclasses.
- **`sealed`** → Must also define permitted subclasses.
- **`non-sealed`** → Allows unrestricted subclassing.

Exercise:

1. Problem Statement:

   Create a **sealed class `Person`** that represents a general person. The `Person` class should **only permit** `Student` and `Employee` to extend it.

   - **`Student`** should be a **`final`** class with fields `name` and `grade`.
   - **`Employee`** should be a **`non-sealed`** class so that it can be **extended without restrictions**.
   - Create **two subclasses of `Employee`**:
     - **`FullTimeEmployee`** (final) → Has fields `name` and `salary`.
     - **`ContractEmployee`** (final) → Has fields `name` and `hourlyRate`.

2. Problem Statement:

   Create a **sealed interface `Shape`** that defines a method `calculateArea()`. The `Shape` interface should **only allow** `Circle` and `Rectangle` to implement it.

   - Implement `Circle` as a **record** with a `radius` field, where the area is **$\pi \times r^2$**.
   - Implement `Rectangle` as a **record** with `length` and `breadth` fields, where the area is **length × breadth**.
   - In the `main` method, create instances of `Circle` and `Rectangle`, and print their areas.

Pattern Matching:

Pattern Matching in Java simplifies type checks (`instanceof`) and `switch` statements by eliminating **explicit casting** and reducing boilerplate code. It enhances readability, safety, and maintainability.

Key Features of Pattern Matching

Pattern Matching for `instanceof` (Java 16)

Before Java 16, explicit casting was required after checking `instanceof`.

Now, **type binding** happens directly in the `instanceof` check.

🚀 **Example: Before Java 16**

```
if (obj instanceof String) {

    String str = (String) obj;  // Explicit casting needed

    System.out.println(str.length());

}
```

Java 16+ (Pattern Matching)

```
if (obj instanceof String str) {  // No need for explicit casting!

    System.out.println(str.length());

}
```

✔ **Eliminates explicit casting**
✔ **Reduces code repetition**

Pattern Matching in `switch` (Java 17+)

Now, `switch` supports **object types**, making it easier to handle **sealed class hierarchies**.

🚀 Example: Using `switch` with Pattern Matching

```
sealed interface Shape permits Circle, Rectangle {}
record Circle(double radius) implements Shape {}
record Rectangle(double length, double width) implements Shape {}
void printShape(Shape shape) {
    switch (shape) {
        case Circle c -> System.out.println("Circle area: " + (Math.PI *
c.radius() * c.radius()));
        case Rectangle r -> System.out.println("Rectangle area: " +
(r.length() * r.width()));
        default -> System.out.println("Unknown shape");
    }
}
```

✔ **No need for `instanceof` + casting**

✔ **Ensures all cases are covered (exhaustive checks with `sealed` classes)**

Exercise :

Shape Area Calculation using Sealed Interface and Pattern Matching

Design a **sealed interface Shape** with a method `area()` that allows only `Circle` and `Square` as implementations.

Write a Java program that:

- Uses a **`switch` expression** and **pattern matching** to evaluate and return the area based on the shape type.
- Uses **records** for `Circle` and `Square` for an immutable and concise implementation.

Guarded pattern:

- Guarded patterns in Java **extend pattern matching** by introducing **additional conditions** using the `when` keyword.
- This makes pattern matching **more selective** and **expressive** compared to regular pattern matching.

```
 String result = switch(Integer.valueOf(100)){
          case Integer i1 when i1 % 2 == 0 -> "even";
          case Integer i1 when i1 % 2 != 0 -> "odd";
          default -> "invalid";
      };
```

Exercise :

1. Implement the grade for marks program using the guarded pattern.

Points to discuss :

1) Yield statement in switch expression
2) Guarded patten and grade marks problem using guarded pattern

Exception Handling :

An **exception** in Java is an **unexpected event** that occurs **during program execution**, disrupting the normal flow of instructions.

Exceptions typically occur due to:

- **Invalid user input** (e.g., dividing by zero)
- **File not found**
- **Network failure**
- **Database connection errors**

Java provides a robust **exception handling mechanism** using `try`, `catch`, `finally`, `throw` and `throws` to handle these issues gracefully.

Keywords in Java Exception Handling

Java provides **five keywords** for handling exceptions:

1. **`try`** → Defines a block where exceptions might occur.
2. **`catch`** → Handles exceptions thrown in the `try` block.
3. **`throw`** → Used to explicitly throw an exception.
4. **`throws`** → Declares exceptions that a method might throw.
5. **`finally`** → Executes code **regardless** of exception occurrence.

Ex:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; //  ArithmeticException (Divide by zero)
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("Execution completed.");
        }
    }
}
```

```
Throwable
├── Error (JVM Errors)
│   ├── OutOfMemoryError
│   ├── StackOverflowError
│   └── VirtualMachineError
│
└── Exception (Recoverable)
    ├── Checked Exception
    │   ├── IOException
    │   ├── SQLException
    │   └── ClassNotFoundException
    │
    └── RuntimeException (Unchecked)
        ├── NullPointerException
        ├── ArithmeticException
        ├── IndexOutOfBoundsException
        ├── IllegalArgumentException
        └── ClassCastException
```

Exception Hierarchy in Java

Java's **exception hierarchy** is structured under the `Throwable` class, which has two main branches:

1. **`Exception`** → Recoverable errors (checked & unchecked)
2. **`Error`** → Critical system failures (non-recoverable)

1. `Throwable` (Root Class)

- The **base class** for all exceptions & errors.
- Contains methods like `printStackTrace()`, `getMessage()`.

---

2. `Error` (Critical, Non-Recoverable)

- Raised by JVM, indicating **serious problems** that **shouldn't be caught**.
- Examples:
  - `OutOfMemoryError` → When JVM runs out of heap space.
  - `StackOverflowError` → Infinite recursion causing stack overflow.

3. Exception (Recoverable Errors)

- Can be **handled using try-catch**.
- Divided into **Checked** and **Unchecked** exceptions.

Checked Exceptions

- Must be **declared in the method signature** or handled using `try-catch`.
- Examples:
  - `IOException` → Issues in file handling.
  - `SQLException` → Errors in database operations.

Unchecked Exceptions

- **Extends `RuntimeException`**, doesn't require explicit handling.
- Examples:
  - `NullPointerException` → Accessing an object reference that is `null`.
  - `ArithmeticException` → Dividing by zero.

Key Takeaways

✔ `Error` → Critical, unrecoverable, JVM-related issues.
✔ `Exception` → Recoverable, should be handled.
✔ `Checked Exceptions` → Must be caught or declared.
✔ `Unchecked Exceptions` → Programmer mistakes (should be avoided).

Try with resources :

`try-with-resources` is a feature introduced in **Java 7** that automatically closes resources (like files, database connections, etc.) after use, preventing **resource leaks**.

- **Auto-closes resources** that implement `AutoCloseable` (e.g., files, DB connections).
- **No need for `finally` block** to close resources.
- **Prevents resource leaks** and makes code cleaner.
- **Multiple resources** can be used in a single `try` block.
- Catch block is also optional after try

User-Defined Exceptions in Java

A **user-defined exception** (custom exception) is a class that extends `Exception` (checked) or `RuntimeException` (unchecked).

Steps to Create a User-Defined Exception

1. Create a class extending `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).
2. Provide constructors to pass error messages.
3. Use `throw` to raise the exception.
4. Handle it using `try-catch` or declare it using `throws`.

Exercise :

A bank maintains customer accounts where each account has a balance. When a customer attempts to withdraw money, the system must check whether the account has **sufficient balance**. If the withdrawal amount exceeds the available balance, the system should throw a **user-defined exception** named `LowBalanceException`.

Requirements:

1. **Create a `BankAccount` class** with the following:
    - A private field `balance` to store the current balance.
    - A method `deposit(double amount)` to add money to the account.
    - A method `withdraw(double amount)` that:
        - **Checks if the balance is sufficient** before withdrawal.
        - **Throws `LowBalanceException`** if the withdrawal amount is greater than the balance.
2. **Define a custom exception `LowBalanceException`** that extends `Exception`.
3. **Create a `BankApp` class** to:
    - Instantiate a `BankAccount` object.
    - Try withdrawing an amount greater than the balance to trigger the exception.
    - Handle the exception gracefully.

Inner Classes :

In Java, **inner classes** are classes defined within another class. They help logically group classes, enhance encapsulation, and improve code readability.

Java provides four types of inner classes:

Member Inner Class

- A non-static inner class defined inside another class.
- Can access all members (including private) of the outer class.
- Requires an instance of the outer class to be instantiated.

```
class Outer {
    class Inner {
        String getMessage() {
            return "Inside Member Inner Class";
```

```java
        }
    }
}

public class MemberInnerClassTest {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        System.out.println(inner.getMessage()); // Expected output:
Inside Member Inner Class
    }
}
```

Static Nested Class

- A static class inside another class.
- Cannot access non-static members of the outer class directly.
- Instantiated without an object of the outer class.

```java
class Outer {
    static class Inner {
        String getMessage() {
            return "Inside Static Nested Class";
        }
    }
}

public class StaticNestedClassTest {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        System.out.println(inner.getMessage()); // Expected output:
Inside Static Nested Class
    }
}
```

Local Inner Class

- Defined inside a method or block.
- Can access final or effectively final variables of the enclosing method.

```java
class Outer {
    String getLocalInnerMessage() {
        class Inner {
            String getMessage() {
                return "Inside Local Inner Class";
            }
        }
        Inner inner = new Inner();
        return inner.getMessage();
    }
}

public class LocalInnerClassTest {
    public static void main(String[] args) {
        Outer outer = new Outer();
        System.out.println(outer.getLocalInnerMessage()); // Expected
output: Inside Local Inner Class
    }
}
```

Anonymous Inner Class

- A class without a name that is instantiated and defined in a single expression.
- Used when a class is needed only once, often with interfaces or abstract classes.

```
abstract class Greeting {
    abstract void sayHello();
}
public class Test {
    public static void main(String[] args) {
        Greeting g = new Greeting() {
            void sayHello() { System.out.println("Hello from Anonymous
Inner Class"); }
        };
        g.sayHello();
    }
}
```

Multithreading :

Thread:

A thread is the smallest unit of execution within a process.

What is Multithreading?

Multithreading is a programming technique where multiple threads execute independently within a single process.

Multithreading enables a program to perform multiple tasks concurrently, improving efficiency and responsiveness.

Key Advantages of Multithreading

1. **Better CPU Utilization**
    - Threads can run in parallel on multi-core processors, making efficient use of CPU resources.
2. **Faster Execution**
    - Tasks that can be executed concurrently (like file downloads, UI updates, and calculations) can be completed faster.
3. **Improved Responsiveness**
    - In GUI-based applications, multithreading helps keep the user interface responsive while performing background tasks.
4. **Concurrent Processing**
    - Useful in server applications where multiple client requests are handled simultaneously (e.g., web servers, database servers).
5. **Efficient Resource Sharing**
    - Threads within the same process share memory and resources, reducing the overhead of creating multiple processes.

Difference Between a Process and a Thread

Feature	Process	Thread
Definition	A process is an independent program in execution with its own memory and resources.	A thread is a lightweight unit of execution within a process. Multiple threads share the same process resources.
Memory Usage	Each process has its own memory space (separate heap and stack).	Threads of the same process share the same memory (heap) but have separate stacks.
Inter-Communication	Processes require **Inter-Process Communication (IPC)** (e.g., pipes, sockets, shared memory) to communicate.	Threads can directly communicate with each other since they share memory.
Creation Overhead	Creating a process is **expensive** (involves allocating separate memory, OS overhead).	Creating a thread is **lightweight** (shares process memory, less overhead).
Execution	Processes run independently and may not share data.	Threads of the same process can run in parallel and share data.

Switching Cost	Context switching between processes is **slower** (requires switching memory spaces).	Context switching between threads is **faster** (shares memory).
Isolation	One process crashing does **not** affect others.	One thread crashing can **affect** the entire process.
Use Case	Used for running separate applications (e.g., browser, text editor).	Used for parallel execution inside a single application (e.g., browser tabs, handling multiple requests in a server).

Differences: Sequential, Concurrent, Parallel, and Asynchronous Execution

Execution Type	**Description**	**Example Scenario**
Sequential Execution	Tasks run one after another, completing one before starting the next.	Reading a file line by line and processing each line before moving to the next.
Concurrent Execution	Multiple tasks start but may not finish simultaneously; they interleave execution.	Handling multiple client requests in a web server where each request gets CPU time in slices.
Parallel Execution	Multiple tasks execute at the same time on different CPU cores.	Sorting two large arrays in parallel on a multi-core processor.
Asynchronous Execution	A task starts and runs in the background without blocking the main program.	Fetching data from an API while continuing with other operations.

Task vs Thread : Real-World Analogy

Imagine a **restaurant kitchen**:

- **CPU (Chef 👨‍🍳)** → Does the actual cooking (execution) - worker.
- **Thread (Supervisor/Head Chef 👨‍💼)** → Assigns orders (tasks) to the chef (CPU) - thread.

- **Tasks (Orders 📜)** → Individual dishes that need to be cooked.

  Threads themselves don't execute tasks—they just **manage** the execution.

  The CPU is the real worker that actually runs the instructions.

Why Do We Need Threads?

Threads **exist to manage execution efficiently**.

The CPU alone cannot decide what to execute next; it needs **instructions** and **scheduling**—this is where threads come in.

In Java, multithreading is achieved using:

- **Extending the `Thread` class**
- **Implementing the `Runnable` interface**
- **Using `ExecutorService` (Thread Pools)**
- **Using `CompletableFuture` for asynchronous programming** (Java 8+)
- **Using Virtual Threads (Java 21+)** for lightweight concurrency

Steps in developing a multithreaded application in Java :

1. Identify the task to be executed as a separate thread
2. Implement the task as a run method either by extending the Thread class or implementing the Runnable interface
3. Create a thread object
   a. If using `Thread`, instantiate it directly.
   b. If using `Runnable`, pass it to a `Thread` instance.
4. Start the thread using the `.start()` method.

Simple Hello World' multi threaded application:

```
class MyThread implements Runnable {
   public void run() {
      System.out.println("Hello Word!");
   }

   public static void main(String[] args) {
      Thread t1 = new Thread(new MyThread());
      t1.start(); // Starts a new thread of execution
   }
}
```

Tasks:

1) Create two threads one thread printing hello and other thread printing world
2) Print numbers from 1 to 100 in different threads

Thread Lifecycle in Java

In Java, a thread goes through several states from creation to termination. The lifecycle consists of **five main states**, as defined in `Thread.State`:

Thread Lifecycle States

State	Description	Example
NEW	Thread is created but not started yet.	`Thread t = new Thread();`
RUNNABLE	Thread is ready to run but waiting for CPU time.	`t.start();`
RUNNING	Thread is actively executing.	The thread is executing its `run()` method.
BLOCKED	Thread is waiting to acquire a lock (due to `synchronized`).	Waiting for a synchronized resource.
WAITING	Thread is waiting indefinitely for another thread to notify it.	`wait();` without timeout.
TIMED_WAITING	Thread is waiting for a fixed amount of time.	`Thread.sleep(1000);`
TERMINATED	Thread has completed execution or was stopped.	Thread finishes `run()`.

Executor Framework:

Executor Framework in Java

The **Executor Framework** in Java (introduced in Java 5) provides a high-level API for managing and controlling a pool of threads efficiently. It decouples task submission from the execution process, making it easier to manage concurrent tasks without manually creating and managing threads.

Importance of Executor Framework

1. **Thread Pooling** – Reduces overhead by reusing threads instead of creating new ones.
2. **Task Management** – Provides better control over task execution (e.g., scheduling, pausing, and shutting down).
3. **Scalability** – Helps in handling multiple tasks efficiently using limited resources.
4. **Predefined Implementations** – Provides various implementations like `FixedThreadPool`, `CachedThreadPool`, and `ScheduledThreadPool`.

Why Do We Need to Shut Down the Executor at the End?

When using `ExecutorService`, it's important to **shut it down properly** after task execution is complete. If we don't call `shutdown()`, the program may keep running indefinitely because the executor **maintains active worker threads**, preventing the JVM from exiting.

Exercise Questions:

3) Print prime numbers between 1 to 100 using executor service
4) Repeat the above using virtual threads

Virtual Threads:

Virtual Threads in Java (Introduced in Java 19 & Stable in Java 21)

Virtual threads are a lightweight alternative to traditional **platform threads** in Java. They allow running a massive number of concurrent tasks efficiently without consuming too many OS threads.

Key Features of Virtual Threads

1. **Lightweight** – Unlike platform threads, virtual threads do not map directly to OS threads; instead, they are managed by the JVM.
2. **Efficient Concurrency** – Millions of virtual threads can run simultaneously without exhausting system resources.
3. **No Need for Thread Pooling** – Since virtual threads are cheap, thread pooling is unnecessary.

4. **Seamless Debugging** – Works with existing debugging tools.
5. **Perfect for IO-Bound Tasks** – Optimized for tasks that involve waiting (e.g., database queries, HTTP calls).

Platform Threads

A platform thread in Java is a traditional thread that is directly mapped to an OS thread.

- Created using `new Thread()`, `Executors.newFixedThreadPool()`, etc.
- Each platform thread directly consumes an OS resource (one thread = one OS thread).
- If it blocks, the OS thread is also blocked.
- Example: CPU-intensive tasks, single-threaded operations.

What is a Carrier Thread

Carrier Threads (A Subset of Platform Threads)

- **Carrier threads are platform threads** that the JVM **internally** uses to execute **virtual threads**.
- The JVM **creates a small number of carrier threads** (usually matching the number of CPU cores).
- They are **not dedicated** to specific virtual threads—**they keep switching** between different virtual threads.
- **Virtual threads are not bound** to a single carrier thread.

Platform Vs Carrier Vs Virtual threads:

- Platform threads = OS threads.
- Carrier threads = Platform threads used by the JVM to run virtual threads.
- Virtual threads = Lightweight, not tied to OS threads, run on carrier threads.

Key Differences: Virtual vs. Platform Threads

Feature	Platform Threads	Virtual Threads
Managed By	OS Scheduler	JVM Scheduler
Resource Usage	Heavy	Lightweight

Concurrency Limit	Limited (based on CPU cores)	Millions of threads possible
Ideal For	CPU-bound tasks	I/O-bound tasks (Web APIs, DB calls)
Context Switching Cost	High	Low

When to Use Virtual Threads?

✅ Handling thousands/millions of short-lived tasks (e.g., HTTP requests, DB queries).
✅ Microservices & high-concurrency applications where threads mostly wait for I/O.
❌ Not ideal for CPU-intensive tasks (e.g., heavy computations, AI models).

Best Practice to run smoothly:

- Virtual threads are not daemon threads, but they don't block the main thread automatically.
- The main thread exits once it completes execution, which might cause virtual threads to terminate prematurely unless explicitly handled.
- Use `join()` or `ExecutorService` to manage virtual thread execution properly.

Before Virtual Threads (Java ≤ 18) – Platform Threads Only

- Java threads were **one-to-one mapped** to OS threads.
- Each Java thread required OS resources (memory, kernel structures).
- Creating thousands of threads was **expensive** due to high resource consumption.
- **Blocking operations (I/O, sleep, waiting for a lock) blocked the OS thread**, causing inefficiencies.

🔹 **Example of Platform Threads (Before Virtual Threads)**

```java
public class PlatformThreadExample {
    public static void main(String[] args) {
        Runnable task = () -> {
            System.out.println("Running on: " + Thread.currentThread());
        };

        for (int i = 5; i > 0; i--) {
            new Thread(task).start();
        }
    }
}
```

🛑 **Issues:**

- Each `new Thread(task).start();` creates a new OS thread.
- If we spawn thousands of such threads, performance degrades.
- **Thread context switching** is expensive because the OS has to manage scheduling.

🚀 After Virtual Threads (Java 19+) – Carrier Threads Manage Virtual Threads

- **Virtual threads** are **not tied to OS threads**.
- A **small pool of platform threads (carrier threads)** executes many virtual threads.
- **Virtual threads block without blocking OS threads**, improving efficiency.

🔹 **Example Using Virtual Threads (Java 19+)**

```
public class VirtualThreadExample {
    public static void main(String[] args) {
        Runnable task = () -> {
            System.out.println("Running on: " + Thread.currentThread());
        };

        for (int i = 5; i > 0; i--) {
            Thread.startVirtualThread(task);
        }
    }
}
```

⚡ Major Improvements with Virtual Threads

Feature	Before (Platform Threads)	After (Virtual Threads)
Thread creation	**Expensive**, 2MB stack memory per thread	**Cheap**, just a small Java object

Blocking calls (I/O, DB, sleep)	Blocks OS thread (inefficient)	Unmounts virtual thread, freeing carrier thread
Number of concurrent threads	**Limited** (OS thread limit)	**Millions** possible
Context switching	**OS-managed, expensive**	**JVM-managed, lightweight**

- **Before Java 19**, Java used **only platform threads** (each mapped to an OS thread).
- **After Java 19**, Java introduced **virtual threads**, which are scheduled on **carrier (platform) threads** for efficiency.
- Virtual threads significantly **reduce memory usage** and **eliminate thread blocking inefficiencies**.

2️⃣ Virtual Threads and How They Use Carrier Threads

- **Virtual threads** are lightweight and **not backed by a dedicated OS thread**.
- Instead, they **run on carrier threads**, which are pooled platform threads.
- A single carrier thread can **execute multiple virtual threads**, but **only one at a time**.
- If a virtual thread **blocks on I/O**, it is unmounted from the carrier thread, allowing the carrier to handle another virtual thread.

3️⃣ Mounting & Unmounting: How Virtual Threads Work on Carrier Threads

Virtual threads get **mounted** onto carrier threads for execution and **unmounted** when they hit blocking calls.

Daemon and Non-Daemon Threads in Java

Daemon Thread:

A **daemon thread** is a background thread that provides services to user threads. It does not prevent the JVM from exiting, and it terminates automatically when all non-daemon threads finish execution.

📌 **Example use cases:** Garbage Collection, Monitoring, Logging.

Non-Daemon (User) Thread:

A **non-daemon thread** (or **user thread**) is an essential thread that performs the core logic of an application. The JVM continues running until all non-daemon threads finish execution.

📌 **Example use cases:** Main application logic, Worker Threads, Request Handling.

Can a Virtual Thread Be a Daemon Thread?

🚨 No! Virtual Threads are always non-daemon threads.

- They do not stop when all platform threads exit.
- You must explicitly manage their lifecycle using `join()` or an `ExecutorService`.

Points to discuss:

- Multithreaded program to check and print if a number is prime or not from 1 to n
- Demonstrating Power of virtual threads
- Thread life cycle
    - A thread starts in the **NEW** state.
    - It moves to **RUNNABLE** when `start()` is called.
    - Running when actively executing.
    - It can enter **WAITING, TIMED_WAITING, or BLOCKED** during execution.
    - Finally, it reaches the **TERMINATED** state when execution is complete.
- Platform vs Carrier vs Virtual threads
    a. **Platform thread**: The Java wrapper for an Operating System (OS) thread that is scheduled by the thread scheduler of the OS.

       If the task is blocked, thread is also blocked - lowering the efficiency of CPU

    b. **Virtual Thread**: A lightweight abstraction of a task that can be bound to a platform thread and is scheduled by the Java virtual thread scheduler. Virtual thread is just an object.

       If the operation is blocked, the virtual thread is unmounted from the carrier thread and the carrier thread is used by another virtual thread.

c. **Carrier thread**: The platform thread on which a virtual thread is mounted(bounded). JVM manages a pool of carrier threads.

How the JVM Manages Carrier Threads

- JVM **dynamically creates and manages a pool of platform threads** to schedule virtual threads.
    - These platform threads act as **carrier threads** for executing virtual threads.
    - Virtual thread scheduling does **not** have a fixed number of carrier threads.
    - The JVM **creates more carrier threads if needed** but keeps their number relatively small.

- **Virtual threads are scheduled on available carrier threads.**
    - The JVM dynamically assigns virtual threads to these platform threads.

- **If a virtual thread blocks, the JVM unmounts it from the carrier thread.**
    - The **freed-up carrier thread** can now execute another virtual thread.

- Once the blocked virtual thread is ready, the JVM remounts it on a carrier thread. This ensures efficient reuse of platform threads.

Summary:

✔ **JVM creates platform threads** (by calling OS-level APIs).

✔ **OS fully manages the platform threads** (scheduling, execution, CPU usage).

✔ **JVM manages virtual threads** and dynamically assigns them to platform threads.

Text blocks

- Text Blocks in Java (Since Java 13, Preview in Java 14, Standard in Java 15)
- A **text block** is a **multi-line string literal** enclosed in triple double-quotes (`"""`), making it easier to write multi-line strings without escape sequences.
- **Preserves formatting** (newlines, indentation).

    - **No need for escape sequences** (`\n`, `\"`).

```
String json = """
    {
        "name": "John",
        "age": 25,
        "city": "New York"
    }
    """;
System.out.println(json);
```

Conditions to Validate

1. The **opening triple quotes (`"""`) must be on the same line as the assignment**.
2. The **content must start on the next line** (not on the same line as `"""`).
3. The **closing triple quotes (`"""`) can be on a new line** (or in the same line)

`var` in Java (Introduced in Java 10)

- **`var` is used for local variable type inference**, allowing the compiler to determine the type at compile time.
- It **cannot be used for fields, method parameters, or return types**.

  Void method(){

  var a = 123; // a will be inferred as int

  var name = “Rama” ; // name will be inferred as String

  }

- _ (underscore) : used like a placeholder, to fulfill the syntax.

Generics:

Generics in Java is a feature that allows the creation of **classes, interfaces, and methods with type parameters**, ensuring **type safety** and **code reusability** while enabling compile-time type checking.

Generics can be called **"parameterized classes"** because they allow defining classes with **type parameters**.

These type parameters act like placeholders for actual data types, making the class flexible and reusable.

Using generics for a **parameterized class:**

```
class Box<T> {
    private T value;
    public void setValue(T value) { this.value = value; }
    public T getValue() { return value; }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>();
        intBox.setValue(10);
        System.out.println(intBox.getValue()); // 10
    }
}
```

Generics in a method by declaring a type parameter **inside the method signature** instead of the class.

This allows the method to work with different data types while maintaining type safety.

```
class Util {
    // Generic method with type parameter T
    public static <T> void print(T data) {
        System.out.println(data);
    }
}

public class Main {
    public static void main(String[] args) {
        Util.print(10);         // Works with Integer
        Util.print("Hello");    // Works with String
        Util.print(3.14);       // Works with Double
    }
}
```

Note:

- `<T>` before the return type (`void`) declares `T` as a **type parameter** for the method.
- The method works with **any type** (`Integer`, `String`, `Double`, etc.).
- **No need to specify the type explicitly**; Java infers it from the arguments.

Exercise :

Problem Statement:

Design a Generic Box Class for Storing and Retrieving Objects

Develop a generic class `Box<T>` that can store and retrieve objects of any type. The class should include:

1. A field to hold an object of type `T`.
2. A method to set the object (`setA(T a)`).
3. A method to retrieve the stored object (`getA()`).

In the `GenericsDemo` class, demonstrate the usage of `Box<T>` with different data types:

- Store and retrieve a `String` value.
- Store and retrieve an instance of the `Person` record, which contains `name` and `age`.

Bounded Type Parameters in Generics (Class Level)

Definition: Bounded type parameters restrict the types that can be used as arguments for a generic class.

Syntax:

```
class GenericClass<T extends SomeClass> { ... }
```

Upper Bound (`extends`):

- The type parameter must be a subclass of a specified class or implement a specific interface.

- Example: `class Box<T extends Number> {}` → `T` can be `Integer`, `Double`, etc.

Wildcards in Generics (Java)

Should be applied only at method level, cannot be applied at class level

Wildcards (`?`) in Java **generics** provide flexibility when working with unknown or multiple types.

They allow a generic type to accept **different but related** types, making code more reusable.

Unbounded (`?`) → When type is unknown, accepts any type

```
void printList(List<?> list) {  // Works with List of any type
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

Can pass `List<Integer>`, `List<String>`, etc.

Upper Bounded (`? extends T`) → For **reading** values. Accepts `Type` or its **subtypes**.

- We know the parent type T, so we can assign child elements and can perform read operations.
  - Ex: Parent p = c; p is known type as T
- The parent cannot be added to unknown child type– write operation is not allowed

```
void sumNumbers(List<? extends Number> list) {
    double sum = 0;
    for (Number num : list) {
        sum += num.doubleValue();
    }
    System.out.println(sum);
}
```

Works with `List<Integer>`, `List<Double>`, etc., since they extend `Number`.

Lower Bounded (`? super T`) → For **writing** values.

- Accepts `Type` or its **supertypes**.
    - We know the child type T(Ex: Integer), So we cannot assign parent type(Ex: Object) to child type-

      Integer a = Object o: not possible

      read operation not possible. But we can add the child elements to its parent (?)whatever it may be..
    - Object.add (Integer)

```java
void addNumber(List<? super Integer> list) {
    list.add(10); // Allowed
}
```

Works with `List<Integer>`, `List<Number>`, or `List<Object>`.

Functional Programming:

Functional programming is a paradigm where functions are treated as first-class citizens.

What does "First-Class Citizens" mean?

In functional programming, **functions are treated like values**. This means you can:

- Assign functions to variables
- Pass functions as arguments to other functions
- Return functions from other functions

Key Features of Functional programming approach:

Higher-Order Functions

- Functions that take other functions as arguments or return functions
- Enhances code reusability and modularity

Declarative Approach

- Focuses on what to do, not how to do it
- Makes code cleaner, more readable, and maintainable

Pure Functions

- Always return the same output for the same input
- Have no side effects (do not modify external state)

Immutability

- Data remains unchanged after creation
- Instead of modifying, new values are returned

Function Composition

- Combining small functions to form more complex ones
- Encourages modular and maintainable code

Recursion (Instead of Loops)

- Uses self-calling functions instead of mutable loops
- Helps maintain immutability and functional purity

Lazy Evaluation

- Expressions are evaluated only when needed
- Improves performance by avoiding unnecessary calculations

Functional Interface:

- An interface exactly with one abstract method.
- Can have any default, private or static methods.

Lambda expression:

Implementation of a single abstract method present in the functional interface

an anonymous function

- No modifier
- No return type
- No method name
- Body and parameters to be separated by ->

Practice Examples:

1. Expression to print hello world
2. Expression that take two int values and prints sum
3. Expression that take two int values and returns sum

Various forms of simplified expressions

An expression with

- no input value and no return - print hello lambda
- one input value and no return - given a string display its length
- one input value and return result - given a string return in its upper case form
- Two are more input values and no return - given three int values display its sum
- Two are more input values and return result - given three input values, return the max value

Built In Functional Interfaces in java.util.function package:

Function : R apply(T a)

Predicate : boolean test(T a)

Supplier : R get()

Consumer : void accept(T a)

- **Function<T, R>**
  `R apply(T a)`
  - Takes an input of type `T` and returns a result of type `R`.
- **Predicate<T>**
  `boolean test(T a)`
  - Takes an input of type `T` and returns a `boolean`, typically used for filtering or conditions.
- **Supplier<R>**
  `R get()`
  - Takes no input and provides (supplies) a result of type `R`.
- **Consumer<T>**
  `void accept(T a)`
  - Takes an input of type `T` and performs some operation without returning a result.
- Already existing functional interfaces up to 1.7
  - Runnable
  - Callable
  - Comparable
  - Comparator
  - ActionListener

Streams

In Java, a stream is a sequence of elements from a data source that can be processed in a functional style.

Streams facilitate operations on collections of data, such as filtering, mapping, and reducing, allowing developers to write concise and expressive code.

Key features of streams in Java:

1. A stream is not a data structure, instead it takes input from the Collections, Arrays or I/O channels.
2. Streams don't modify the original data, they only provide the result as per the pipelined methods.
3. Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined.
4. Terminal operations mark the end of the stream and return the result.

Stream Pipeline

The three main aspects of a stream :
source, intermediate operations, and terminal operations.

Source → Intermediate Operations → Terminal Operation

- Streams always start with a **source** (Collection, Array, Stream.of, etc.).
- They pass through **intermediate operations** (transformations like `filter`, `map`, `sorted`).
- A **terminal operation** (like `collect`, `count`, `forEach`) triggers execution.

Creating Streams:

A. From Collections:

Collections in Java provide a `stream()` method to create a stream.

```
List<String> names = List.of("Alice", "Bob", "Charlie");

Stream<String> nameStream = names.stream();
```

B. From Arrays:

Use `Arrays.stream()` to create a stream from an array.

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

IntStream numberStream = Arrays.stream(numbers);
```

C. Using `Stream.of()`

You can create a stream from a set of elements directly.

```
Stream<String> fruitStream = Stream.of("Apple", "Banana", "Mango");
```

Intermediate Operations:

Operation	Description
filter(n -> n > 10)	Filters elements greater than 10
map(n -> n * 2)	Transforms each element by multiplying by 2
flatMap(List::stream)	Flattens nested lists into a single stream
distinct()	Removes duplicate elements
sorted()	Sorts elements in natural order
sorted(Comparator.reverseOrder())	Sorts elements in reverse order
peek(System.out::println)	Performs an action (debugging/logging)
limit(5)	Limits the stream to 5 elements
skip(3)	Skips the first 3 elements

Terminal Operations

Operation	Description
reduce(0, Integer::sum)	Reduces elements to their sum

forEach(System.out::println)	Prints each element
collect(Collectors.toList())	Collects elements into a list
count()	Counts the number of elements
min(Comparator.naturalOrder())	Finds the minimum element
max(Comparator.naturalOrder())	Finds the maximum element
anyMatch(n -> n > 10)	Checks if any element matches the condition
allMatch(n -> n > 10)	Checks if all elements match the condition
noneMatch(n -> n > 10)	Checks if no element matches the condition
findFirst().orElse(null)	Retrieves the first element
findAny().orElse(null)	Retrieves any element (useful in parallel streams)

mapTo Methods in Java Streams

In Java Streams, mapTo methods are specialized versions of map() used for primitive data types (int, long, double). They provide better performance by avoiding unnecessary boxing and unboxing.

Types of mapTo Methods

Method	Description	Return Type
mapToInt(ToIntFunction<T>)	Converts each element to an int value	IntStream
mapToLong(ToLongFunction<T>)	Converts each element to a long value	LongStream

mapToDouble(ToDoubleFunction<T>)	Converts each element to a double value	DoubleStream

Collect:
`collect()` transforms a stream into a collection or other structures.
It is widely used with `Collectors` like `toList()`, `toSet()`, `toMap()`, `joining()`, `groupingBy()`, etc.
It is powerful for grouping, partitioning, and reducing data efficiently.

IntStream Specialized Reducers

- `sum()`: Calculates the sum of the elements.
- `average()`: Calculates the average of the elements.
- `max()`: Finds the maximum element.
- `min()`: Finds the minimum element.

Exercise Questions:

1. Filter Even Numbers and Collect into a List
   - Given List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
   - Use filter() to get even numbers and collect() to store them in a list.
2. Square Each Number and Collect into a List
   - Given List<Integer> numbers = List.of(2, 4, 6, 8, 10);
   - Use map() to square each number and collect() to store results.
3. Find the Sum of All Odd Numbers
   - Given List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
   - Use filter() to get odd numbers, then reduce() to find the sum.
4. Convert List of Strings to Uppercase
   - Given List<String> words = List.of("java", "streams", "functional");
   - Use map() to convert each word to uppercase and collect() into a list.
5. Concatenate All Strings into a Single String
   - Given List<String> names = List.of("Alice", "Bob", "Charlie");
   - Use reduce() to concatenate all names into a single string.
6. Filter Students Who Scored More Than 75 and Collect into a List
   - Define record Student(String name, int age, double marks) {}

Given a list of students:

```
List<Student> students = List.of(

  new Student("Alice", 22, 80),

  new Student("Bob", 20, 65),

  new Student("Charlie", 23, 90)
```

);

    - Use filter() to get students with marks > 75, then collect() into a list.
7. Get List of Student Names Who Passed (Marks >= 40)
    - Use filter() to check for marks >= 40, then map(Student::name), and collect().
8. Find the Total Marks of All Students
    - Use map(Student::marks) and reduce(0.0, Double::sum).
9. Find the Highest Marks Scored
    - Use map(Student::marks) and reduce(Double::max).
10. Find the Average Marks of All Students
    - Use mapToDouble(Student::marks).average().orElse(0).

Roadmap to SDE roles