**OOPS-JAVA**

Following Links for the Reference:

<https://github.com/kunal-kushwaha/DSA-Bootcamp-Java/blob/main/assignments/14-oop.md>

<https://github.com/kunal-kushwaha/DSA-Bootcamp-Java/tree/main/lectures/17-oop/notes>

Day-1 Topics

1. Why we need a OOPs
2. When use the OOPs
3. What are Advantages of OOPs
4. Why Java Is Best for OOPs and Why to Choose Java except all other like C++, Python and more.
5. What is class and real time example
6. What is Object and real time example
7. Properties of the Object and Class
8. Memory Management[Heap and Stack] in Class, Object and Variables and more others.
9. What are Instance Variables
10. What is dynamic Allocation in Java / OOPs
11. Why java was a Static type language
12. What difference between Compile time and runtime
13. What is Constructor and By default Constructor in Java
14. Why only Pass by Value was Possible in Java
15. Explain the “new” Keyword in Java
16. What is Wrapper Class in Java
17. What is Final Keyword in Java
18. What is Garbage Collection in Java

**Why we need a OOPs**

We need OOPs to write clean, modular, and scalable code. It allows us to model real-world entities as objects, promoting code reusability through inheritance, security through encapsulation, flexibility via polymorphism, and simplicity using abstraction. These principles help us manage complexity, especially in large applications.

Additionally, OOPs supports design patterns and architectural best practices like SOLID principles, making our codebase more maintainable and testable. That’s why it's foundational in frameworks like Spring and commonly used in enterprise-level system design.

**✅ Why We Need OOPs**

* **Real-world modelling**: Represents entities like User, Car, Product as classes and objects.
* **Encapsulation**: Hides internal data and exposes only necessary methods (e.g., getBalance()).
* **Inheritance**: Reuses code across related classes (e.g., Car inherits from Vehicle).
* **Polymorphism**: Allows different behaviors using the same interface (e.g., draw() for Circle, Rectangle).
* **Abstraction**: Hides complex logic behind simple interfaces (e.g., DatabaseService.connect()).
* **Modularity**: Breaks code into separate components, making it easier to maintain and test.
* **Scalability**: Makes large systems easier to extend with new features.
* **Supports Design Patterns**: Enables use of Factory, Singleton, Strategy, and other patterns.
* **Essential for Architecture**: Backbone of frameworks like Spring, used in layered and clean architectures.
* **Improves Maintainability**: Changes in one part of the system have minimal impact on others.

**When use the OOPs**

I use OOPs when I'm building large-scale or modular applications where maintainability, scalability, and reusability are critical. For example, when designing layered architectures like MVC or working with frameworks like Spring Boot, OOP helps structure code into entities, services, and controllers using classes and interfaces.

It’s also ideal when the application involves real-world modelling—like users, orders, or payments—because OOP allows me to represent them as objects. Additionally, when I need to implement design patterns or follow SOLID principles, OOP is the go-to approach for writing clean, extensible code.

**What are Advantages of OOPs**

**Advantages of OOPs:**

* **Modularity:** Code is organized into classes and objects, making it easier to manage and debug.
* **Reusability:** Inheritance allows reuse of existing code, reducing redundancy and development time.
* **Encapsulation:** Data hiding protects object integrity by restricting direct access to internal states.
* **Flexibility:** Polymorphism lets the same interface work with different underlying forms, enabling dynamic behavior.
* **Maintainability:** Changes in one part of the system have minimal impact on others due to well-defined interfaces.
* **Scalability:** Easier to extend the system by adding new classes without affecting existing code.
* **Improved Productivity:** Clear structure helps teams collaborate and speeds up development.
* **Supports Design Patterns:** Facilitates implementing proven solutions for common problems, improving code quality.

**Why Java Is Best for OOPs and Why to Choose Java except all other like C++, Python and more.**

**Why is Java best for OOPs and why choose it over others like C++ or Python?**

Java is designed with OOP principles at its core, making it a **pure object-oriented language** where everything (except primitive types) is an object. Unlike C++, which supports both procedural and object-oriented styles, Java enforces a consistent OOP approach, which helps maintain cleaner and more maintainable code in large projects. Additionally, Java has **automatic memory management via garbage collection**, reducing common errors like memory leaks or pointer issues often seen in C++.

Compared to Python, which is dynamically typed and allows multiple paradigms, Java’s **strong static typing and compile-time checks** catch many errors early, improving reliability for enterprise-grade applications. Java’s rich ecosystem, platform independence (via JVM), vast libraries, robust frameworks (like Spring), and strong community support make it the preferred choice for scalable, secure, and maintainable OOP-based software development in industry.

**What is class and real time example**

**What is a Class?**  
A **class** is a blueprint or template in OOP that defines the structure and behavior (data members and methods) of objects. It encapsulates related properties (attributes) and functions (methods) into a single unit, allowing us to create multiple objects with the same characteristics.

**Real-time Example:**  
Consider a Car class. It can have attributes like colour, model, and engineType, and methods like start(), accelerate(), and brake(). Each specific car (like your red Honda or a blue Toyota) is an **object** created from this Car class, having its own values for the attributes but sharing the same behavior defined by the class.

**What is Object and real time example**

**What is an Object?**  
An **object** is an instance of a class. It represents a specific entity with defined values for the properties (attributes) and can perform actions using the class methods. Objects hold actual data and interact with each other in an OOP system.

**Real-time Example:**  
Using the earlier Car class example, your own car — say, a red 2020 Honda Civic — is an **object** of the Car class. It has specific attribute values like colour = red, model = Civic, and year = 2020, and you can invoke methods like start() or accelerate() on this particular car object.

**Properties of the Object and Class**

**Properties of a Class:**

* **Blueprint/Template**: Defines the structure (attributes) and behavior (methods) common to all objects of that type.
* **No Memory Allocation**: A class itself doesn’t hold data; it just describes what objects will have.
* **Can Have Members**: Contains fields (variables), methods (functions), constructors, and nested classes.
* **Supports Access Modifiers**: Controls visibility with keywords like private, public, protected.
* **Can Inherit**: Supports inheritance to create new classes based on existing ones.

**Properties of an Object:**

* **Instance of a Class**: A concrete entity created using a class blueprint.
* **Has State**: Contains actual values for attributes defined in the class.
* **Has Behavior**: Can invoke methods defined in the class to perform actions.
* **Occupies Memory**: Stores data in memory unique to that particular object.
* **Identity**: Each object has a unique identity, even if their state (attribute values) are the same.

**Memory Management[Heap and Stack] in Class, Object and Variables and more others.**

**Memory Management: Heap vs Stack**

**1. Stack Memory:**

* Stores **local variables** and **method call information** (like parameters, return addresses).
* Memory is allocated and deallocated automatically in a **LIFO (Last-In, First-Out)** manner.
* Variables stored here have **short-lived scope**—only live during method execution.
* Example: Primitive local variables like int x = 10; inside a method are stored on the stack.

**2. Heap Memory:**

* Stores **objects** and **class instances** created with new.
* Memory allocation is **dynamic** and managed by the **Garbage Collector** in Java.
* Objects remain in heap as long as references to them exist.
* Example: When you create an object Car myCar = new Car();, the Car object is stored in heap, while the reference myCar is stored in stack.

**How This Relates to Classes, Objects, and Variables:**

* **Class** itself doesn’t occupy memory until an object is created; class **methods and static variables** are stored in a special area called the **Method Area (or Metaspace in newer JVMs)**.
* **Objects** are stored on the **heap** with all their instance variables.
* **Reference variables** (like myCar) live on the **stack** and point to objects in the heap.
* **Primitive variables** inside objects live within the heap as part of the object’s data.
* **Local primitive variables** inside methods live on the stack.

**Summary Example:**

**public void drive() {**

**int speed = 60; // stored on stack**

**Car myCar = new Car(); // reference 'myCar' on stack, 'Car' object on heap**

**myCar.accelerate(speed); // method call info on stack**

**}**

* speed is a primitive local variable on the stack.
* myCar reference is on the stack pointing to the Car object on the heap.
* The Car object's attributes are stored inside the heap object.

**What are Instance Variables**

**What are Instance Variables?**

Instance variables are **non-static variables declared inside a class but outside any method**. Each object created from the class gets its **own copy** of these variables, so their values can differ between objects. They represent the **state or attributes** of an object.

**Example:**  
In a Car class, variables like colour, model, and speed are instance variables. Each Car object can have a different colour or speed, reflecting its individual state.

**What is dynamic Allocation in Java / OOPs**

**What is Dynamic Allocation in Java/OOPs?**

Dynamic allocation refers to **allocating memory at runtime** (rather than compile-time), allowing the program to create objects as needed during execution. In Java, this happens when you use the new keyword to create an object. The memory for the object is allocated on the **heap**, enabling flexible and efficient use of memory.

This contrasts with static allocation where memory size and lifetime are fixed at compile-time. Dynamic allocation is essential in OOP to create multiple objects from a class, each with its own state, supporting concepts like polymorphism and runtime behavior changes.

**Why java was a Static type language**

**Why is Java a Static-Typed Language?**

Java is statically typed because **variable types are checked at compile-time**, meaning every variable and expression type is known before the program runs. This helps catch many errors early, such as type mismatches, which improves code reliability and safety. Static typing also enables better performance optimizations by the compiler.

Moreover, static typing enforces clear contracts in the code, making it easier to maintain and understand, especially in large-scale enterprise applications. While it requires more upfront declaration compared to dynamically typed languages like Python, the trade-off leads to fewer runtime errors and more predictable behavior.

**What difference between Compile time and runtime**

**Difference Between Compile-Time and Runtime:**

* **Compile-Time:**  
  This is the phase when the source code is translated into bytecode (in Java) or machine code (in other languages). During compile-time, the compiler checks for **syntax errors, type checking, and code structure**. If errors are found, the program won’t compile. Compile-time ensures the code is structurally correct before execution.
* **Runtime:**  
  This is when the program is actually executed by the JVM or CPU. Runtime involves **dynamic activities** like object creation, method calls, memory allocation, and handling user input or external data. Errors occurring at runtime (like dividing by zero or null pointer exceptions) are called **runtime errors** because they happen while the program is running.

**What is Constructor and By default Constructor in Java**

**What is a Constructor?**  
A **constructor** is a special method in a class that is called automatically when an object is created. Its main purpose is to **initialize the object’s state** by assigning initial values to instance variables. Constructors have the same name as the class and do not have a return type, not even void.

**What is a Default Constructor?**  
If no constructor is explicitly defined in a class, Java provides a **default constructor** automatically. This default constructor is **no-argument** and initializes the object with default values (e.g., 0 for numeric types, null for objects). However, if you define any constructor yourself, the default constructor is not created by the compiler.

**Why only Pass by Value was Possible in Java**

**Why Only Pass-by-Value Is Possible in Java?**

Java strictly uses **pass-by-value** for method arguments, meaning it passes a copy of the variable’s value to the method. For **primitive types**, this is straightforward—the actual value is copied. For **objects**, the *reference* (memory address) to the object is copied, not the object itself. This means the method receives a copy of the reference, allowing it to modify the object’s contents but not the original reference variable outside the method.

Java doesn’t support pass-by-reference (where a method could modify the caller’s variable directly) because it was designed for simplicity and to avoid unintended side effects, making the language safer and easier to debug.

**Explain the “new” Keyword in Java**

**What is the new Keyword in Java?**

The new keyword is used to **dynamically allocate memory on the heap** for a new object at runtime. When you use new, it creates an instance of a class, allocates memory for it, and calls the constructor to initialize the object. It returns a reference to the newly created object.

Without new, objects cannot be created in Java; you only have references, not actual instances. This keyword is fundamental for implementing OOP concepts like encapsulation and polymorphism by allowing multiple objects with their own state.

**What is Wrapper Class in Java**

**What is a Wrapper Class in Java?**

A **wrapper class** in Java is a class that **encapsulates (wraps) a primitive data type** into an object. This allows primitives like int, char, or Boolean to be treated as objects, which is necessary when working with Java Collections (like ArrayList) or APIs that require objects.

Java provides wrapper classes for all primitive types, such as Integer for int, Double for double, and Character for char. Wrapper classes also provide useful utility methods for converting, comparing, and manipulating the primitive values.

**What is Final Keyword in Java**

**What is the final Keyword in Java?**

The final keyword is used to declare **constants** or **immutability** in Java. It can be applied to variables, methods, and classes:

* **Final Variable:** Its value, once assigned, **cannot be changed** (constant).
* **Final Method:** Cannot be **overridden** by subclasses.
* **Final Class:** Cannot be **inherited** (no subclass can extend it).

Using final helps enforce security, thread-safety, and design constraints by preventing modification or extension where it’s not desired.

**What is Garbage Collection in Java**

**What is Garbage Collection in Java?**

Garbage Collection (GC) is an automatic memory management process in Java where the **JVM identifies and removes objects that are no longer reachable or needed** by the program. This helps free up heap memory and prevents memory leaks without requiring explicit deallocation by the programmer.

GC runs in the background and tracks object references. When an object becomes unreachable (no live references), it becomes eligible for garbage collection. This automatic cleanup improves application performance and simplifies programming by abstracting manual memory management.

Day-2 Topics