**Java Features :**

Java is a **high-level, object-oriented, platform-independent** programming language. Its key features include **security, portability, robustness, and multithreading**. The JVM enables Java code to run on any machine, making it ‘Write Once, Run Anywhere.’ Java also has automatic garbage collection, a rich standard library, and built-in security mechanisms, which make it ideal for enterprise and cross-platform applications.

**What is OOPS ?**

Object-Oriented Programming System (OOPS) is a programming paradigm based on the concept of objects. It defines a set of principles used to describe and manage the state and behaviour of objects.

**Class (blueprint):**

A class in programming is a blueprint or template for creating objects. it defines the structure and behaviour that the objects created from the class will have.

A class specifies the attributes(properties) and methods (functions) that its objects will contain and use.

**Object (instance):**

It is an instance of the class.

It represents a specific "real-world" entity created based on the class blueprint.

**Pillars of OOP / OOPS concepts:**

1. **Encapsulation :**

Binding the data and functions into single entity also known as class

Use private to protect sensitive data.

provide getter/setter (accessor/ mutator methods) to access the private attributes.

Controls access using access specifiers

* **Private :** Accessible only within the class.
* **Default :** within a class and package
* **Protected :** Accessible within class, package and subclasses.
* **Public :** Accessible from anywhere.

E.g.

* Each backend service (e.g., LightOptimizationService, SmokeDetectorService) encapsulates its own **logic, data, and responsibilities**.
* Data models (e.g., SensorData, User, Notification) have **private fields with getters/setters**, ensuring data hiding and controlled access.

1. **Inheritance :**

Acquiring properties and behaviour from a parent (base) class to another class.

Code reusability

**Types:**

**Single Inheritance :** One base, one derived class.

**Multiple Inheritance** : Derived class inherits from multiple base classes. It is not allowed in java, but can be achieved using Interface

**Multilevel Inheritance** : Chain of inheritance.

**Hierarchical Inheritance** : One base, multiple derived classes.

**Hybrid Inheritance** : Combination of multiple inheritance types.

child class object can access parent class methods by extending that class and using super keyword to access variables, invoke methods, and calling constructor.

Invoking the parent class parameterized constructor

super(custId, custName);

1. **Polymorphism :**

Simply it means many forms

Polymorphism means the ability of an object to take many forms.

In programming, it allows one method, class, or interface to work in different ways depending on the context — like using the same method name for different tasks.

1. **Static Polymorphism (Compile time) (Overloading) :**

Achieved using **method overloading and operator overloading**

**For Overloading:**

1. **Method name** must be the **same**.
2. **Parameters** must be **different** (different type, number, or order).
3. **Return type** can be **same** or **different** — but return type **alone** is **not enough** to overload.
4. **Access modifiers** (public, private, etc.) can be **anything**.
5. **Static methods** **can** **be overloaded**.

E.g. Constructor overloading

same name, different attributes, within class

1. **Dynamic Polymorphism (Rum time) (Overriding) :**

If we are not satisfied with the parent class method implementation then we can override the parent class method in the child class.

**Method name, parameters, and return type** must be **exactly same**.

**Return type** can be **covariant** (child type).

* Example: Parent method returns Animal, child method can return Dog.

**Access level** cannot be more restrictive: Increase the scope of access modifier

* If parent method is public, child method must be public.

**Only inherited methods** can be overridden.

* (Private methods, static methods, constructors **cannot be overridden**.)

**Static methods** cannot be overridden (they are hidden, not overridden — called **method hiding**).

When you **declare a static method** with the same signature in a subclass, it **does not override** the superclass method — it **hides** it. This is known as **method hiding** and not overriding.

**Final methods** cannot be overridden.

**Override annotation (@Override)** is **optional**, but good practice.

Method should not have weaker access specifier

**4. Abstraction :**

Hides implementation details and shows only functionality.

Achieved through abstract classes (having at least one pure virtual function)

It provides the modularity, and a proper structure which is easy to understand

Abstraction can be achieved using:

**1. Abstract classes (using the abstract keyword)**

a. Contains at least one abstract method (a method without a body).

b. Cannot be instantiated; you can’t create objects of an abstract class because object need to be fully implemented and abstract class is incomplete.

c. Can have both abstract (unimplemented) and concrete (implemented) methods.

d. use extends keyword to implement the method in child class

e. a class can be made abstract even if it does not contains any abstract method

**2. Interfaces (using the interface keyword)**

a. Defines a contract for what it can do without specifying how it does it.

b. It contains all the abstract methods, and can also contain default and static methods after Java 8

b. Methods in interfaces are implicitly abstract and public (prior to Java 8) and attributes/ variables are by default public, static and final.

c. A class implements an interface using the implements keyword.

e.g.

* Interfaces like NotificationService, SensorProcessor, or AlertHandler define **what a service does**, not **how it does it**.
* Clients (like the controller or gateway) interact through abstract layers (e.g., service interfaces), not with implementation details.

**Constructor:**

1. it is a special method used in OOP that is automatically called when an object is created from a class.

2. Used to initialise for object when it is created.

3. it is automatically invoked

4. **The constructor’s name should be the same as the class name.**

5. there are three type of constructors - default, parameterized and parameterless.

6. if a class have only parameterised constructor, then you should invoke that otherwise there will be compilation error

7. if there's no parameterised constructor then default constructor will get invoked automatically

**The default constructor invokes super() and sets all instance variables to a default value such as 0, null.**

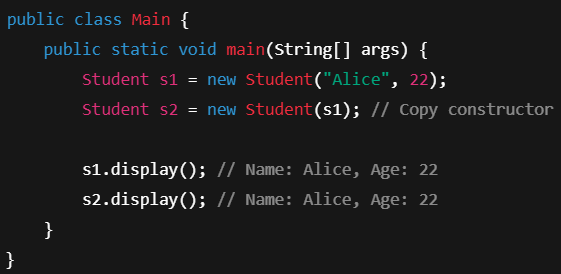
**If we do not define a constructor for a class, the compiler will generate one for us**

**If we want to call parent class constructor, we must call it in the constructor's first line.**

**Copy Constructor :**

A Copy constructor is a constructor in java that **creates a new object by copying the values of existing object** of same class.

* Java doesn’t provide default copy constructor; we need to create it manually.
* It accepts one parameter which is object of same class.
* Used when we need duplicate object with same values



**Copy Constructor vs clone()**

**Copy constructor** can be defined according to the requirement, no need to handle cloneNotSupportedException.

**clone()** is a method of **object class**, we **cannot override** the implementation of this method as it is more complex, we **must implement the Cloneable** **Interface as .clone() is protected** and cannot be called directly to handle the exception.

**Cloneable** is a marker interface, only used to indicate JVM that this class allows cloning.

**Abstract class :**

1. Abstract keyword signifies that something is not complete.

2. An abstract method is a method without any definition.

3. Abstract class cannot be instantiated (object cannot be created)

4. If a class contains at least one abstract method, the class should be abstract.

5. It enforces **inheritance & overriding**.

6. Object of child class can access non abstract method of parent (abstract) class.

7. A class extending a class must provide body to abstract method otherwise that class needs to be declared as abstract.

Customer regularCustomer = new RegularCustomer();

if the parent reference is abstract. The object must be of child.

**Interface :**

1. methods in an interface are implicitly public & abstract (methods can be default / static)

2. data fields are implicitly public, static & final.

3. we cannot create an object of an interface & constructor inside Interface.

4. An interface/class can extend more than one interface (Multiple inheritance)

5. Implements keyword is used

6. The class implementing an interface must implement all the specified methods otherwise, they should be made abstract.

**Super keyword :**

used to invoke parent class constructor in child class

used for invoking parent class method from a child class method

used to access parent class instance variable in child class in case theres a variable in the child class with the same name.

**This keyword :**

refer to instance variable when there’s ambiguity with local variable.

call another method of the same class explicitly

call another constructor of same class.

Static variable / methods are accessed using classname

E.g. Customer.counter: (counter is a variable name)

**Final keyword :**

final keyword is used where the value remains constant and never change

used with classes, variable & methods

**Classes** - cannot be extended

**Methods** - cannot be overridden

**Variales** - values remain unchanged

**Static Block :**

A Static block is used for static initialization in java.

It runs once when the class is loaded into memory, before the construction or main method

**Object class : Parent of all classes**

**.equals() :**

1. .equals() method is present inside the Object class.
2. It is used to compare values of two objects.
3. By default, it checks if the reference/memory address of two objects are equal or not.
4. If we want to compare the values/content of two objects then we need to override this method.

**.toString() :**

1. the toString() method is a built-in method in the Object class.
2. It’s used to return a string representation of an object.
3. by default it returns class name @ hexadecimal from of hashcode
4. if you want to show something meaningful, like the object’s data then you must override the method

**.hashCode() :**

1. Return type is integer, it returns a 32 bit number which is unique to all objects.
2. **There are far more possible combinations of object data than possible hash codes (which are 32-bit integers).**
3. Since Java’s hashCode() method returns an int, there are only **2³² (about 4.29 billion)** unique possible hash codes.
4. But the number of possible combinations of data inside objects can be **much larger** than that — especially for objects holding long strings, numbers, or multiple fields.

It is used in HashMap to calculate the bucket index.

**Record class :**

A **record** in Java is a **special type of class** introduced in **Java 14 (preview)** and made **stable in Java 16**. It is used to create **immutable classes** with **less boilerplate code**.

**Syntax** - public record Person(String name, int age) { }

* No need to create class constructor, getters, toString(), equals(), and hashCode()
* Cannot have **instance fields** other than the ones declared in the header.
* Cannot be **abstract**, **extend another class**, or **change field values** after creation.
* We can use multiple constructors, create static method also inside record.

Use records **only when**:

* You just need to **carry data** (like a DTO or value object)
* The data is **immutable**

**JIT Compiler :**

**JIT** stands for **Just-In-Time** compiler.  
It’s a **part of the runtime environment** that **compiles code while the program is running**.

A smart system that waits until your program is about to run something, then compiles it **just in time** for execution, instead of compiling everything ahead of time.

Java Code (.java)

↓

Compiler (javac)

↓

Bytecode (.class files)

↓

Java Virtual Machine (JVM)

↳ Interpreter ←— runs code line-by-line

↳ JIT Compiler ←— compiles "hot" code to native machine code

In Java, the JIT (Just-In-Time) compiler is a part of the JVM (Java Virtual Machine). Its main job is to improve the performance of Java applications by compiling bytecode into native machine code at runtime, just before execution.

**When Java code is compiled using javac, it gets converted to bytecode (.class files). This bytecode is platform-independent and runs on the JVM. Initially, the JVM interprets the bytecode line-by-line, which is slower. But when certain methods or code blocks are executed frequently (called 'hot code'), the JIT compiler kicks in. It compiles those parts into optimized machine code, so they run much faster.**

The JVM used by default in Java is HotSpot, and it includes two JIT compilers: C1 for quick startup and C2 for better long-term performance.

Advantages :

* **Faster runtime performance** (for long-running apps)
* **Dynamic optimization** (can optimize based on actual usage)
* **Smaller initial binaries** (since code is compiled at runtime)

Disadvantages :

* **Slower startup time** (since it compiles during execution)
* Higher **memory usage**

**AOT is removed in Java 16.**

**Question :**

**What is Run-time Polymorphism?**

**What is Abstraction?**

**What is the difference between Method Overloading and Method Overriding?**

**What is the Interface? Why we use default in it?**

**Object without any reference will they exist in application**

**difference between default and protected**

**What version of java you are using**

Currently I am using Java 17 version it is Long Time Support (LTS) version which provides stability and modern language features like sealed classes, pattern matching in instanceOf, float point and switch case enhancement. Also, better garbage collection performance. **For production systems, we typically prefer using an LTS version for long-term maintainability."**

**features of java**

Can we create an object of interface?

Can interface have constructors?

**What is method overriding with covariant return types?**

**What is Enum?**

Enum stands for **Enumerator**.

An enum is a special **data type** in java lets you define your own **set of constraints**.

**What is Enum overloading?**

**enum overloading** usually refers to overloading **constructors** or **methods** **inside an enum** to give enum constants **more flexibility or custom behavior**.

**Difference between static block and constructor**

**Type casting**

**Widening Casting (Automatic / Implicit)**

* Small type → Big type
* Happens **automatically** (no risk of data loss).

**Narrowing Casting (Manual / Explicit)**

* Big type → Small type
* You must **explicitly** tell Java.
* There **may be data loss**.

**How does toString() behave in default method?**

**How many ways to create a string?**

Java creates strings in two ways: using literals (string pool) and using the new keyword (heap memory).

Other ways include converting from arrays, objects, or using builders.

**Suppose I want to add a name to the final variable, how can I do that?**

final object reference (like a List, Map, etc.)

You can’t reassign the object, but you can modify the contents of the object.

final List<String> names = new ArrayList<>();

names.add("Alice"); // ✅ This is allowed

names.add("Bob"); // ✅ Also allowed

// names = new ArrayList<>(); ❌ Not allowed (can't reassign)

So here, you're adding a name to the object the final variable points to, not changing the variable itself.

**How to stop a class from getting inherited? other way apart from final?**

Use private Constructor, Use Enum instead Enum is a final class and cannot be extended

**What is Immutability in Java?**

Immutability means once an object is created, its state (data) cannot be changed.

In other words - No setters. No changes to internal fields after construction. Safe to share across threads — thread-safe by default.

Example:

All wrapper classes in Java (String, Integer, Double, etc.) are immutable.

**How Can We Create an Immutable Class?**

To make a class immutable:

* **Mark the class final**. So it can’t be subclassed and altered.
* Make all **fields private and final**. Prevents direct access and reassignment.
* **No setters**. Don’t allow updates after creation.
* Initialize fields via **constructor** only. Set values during object creation.
* If a field is a mutable object (like Date, List). Return a defensive copy in getter methods.

**What is immutable class?**

An immutable class is one whose state cannot be changed once created. A classic example is the String class in Java. I can also create my own immutable classes by declaring the **class final, using private final fields, initializing them via constructor, and not providing any setters.**

Immutable objects are **thread-safe, simple to debug, and are great for use cases like cache keys, multithreaded applications, and safe data transfer**. For example, I created an immutable Employee class to store employee ID and name where these values must never change after creation.

Immutable = unchangeable after construction

**E.g. Wrapper class, String class**

**Why Use Immutable Objects?**

* Thread-safe without synchronization
* Safe to cache
* Predictable behavior
* Easy to use in collections like Set and as keys in Map

**What is garbage collection in Java?**

* A garbage collection is a process where Java Virtual Machie (JVM) automatically identifies and removes object that are no longer needed or reachable in memory, freeing up the **heap space** and preventing leaks.
* Java uses an automatic garbage collector to manage memory, so developers don’t need to manually release memory like languages such as C, C++.
* Common algorithms **include Mark and Sweep, Generational Garbage Collection, and G1 (Garbage First).**
* The heap is divided **into Young Generation Old Generation and sometimes Metaspace** (for class metadate). **Short-lived objects** usually collected quickly from the **Young Generation** using techniques like **minor GC**, while **long-lived** objects go into the **Old Generation**, which is cleaned less frequently **(major GC).**
* Developers can monitor or tune garbage collection using JVM options or **tools like jvisualvm, jconsole, or profiling tools.**

**Mark and Sweep** is a two-step process to remove unused objects from memory:

1. **Mark Phase:**  
   The garbage collector starts from root references (like local variables) and marks all reachable (in-use) objects.
2. **Sweep Phase:**  
   It scans the heap and deletes all **unmarked (unused)** objects, freeing up memory.

**Metaspace**:

It is a memory area used by JVM to store metadata of class such as class name, methods, features, annotations etc. Replacement of PermGen. It is dynamically resizable according to the requirement.

**Finalize() :**

A finalize() method in Java is a special method **defined in the object class**. It was designed to allow an object to clean up resources (like closing files or releasing connections) before it is garbage collected.

* protected void finalize() throws Throwable
* can be overridden and is called by JVM before garbage collection only once, and not guaranteed to be called at all.
* **It is deprecated in Java 9 and removed in java 18, because:**
* It caused **unpredictable** behaviour.
* It could delay GC if objects are **resurrected** inside finalize().
* Better alternatives now exist like try-with-resource or using AutoCloseable interface.

**Ways to create an Object in Java**

1. Using new keyword ---

MyClass obj = new MyClass();

1. Using clone() method

MyClass obj2 = (MyClass) obj1.clone();

1. Using ObjectInputStream(Deserialization)
2. Using Reflection with Constructor
3. Using Class.forName() ---

Class cls = Class.forName(“MyClass”);

Object obj = cls.newInstance();