**BASIC Part**

**Class-** class helps us to generate objects.

**Example for children** : class is book and object are pages in a book

**New –**  it sends request to the class to create object

- once object is created it will get objects address and stores that into reference variable.

**Garbage Collector** – GC is used to delete the unwanted objects from RAM which is developed by developer.

**Non Static** – Non static are where static keyword is not used eg. Int x=10;

* This are directly stored into object memory
* If we want to access the variable then create object
* A a1=new A();
* a1.x
* it should be inside class but not in method like global variable.
* Without creating object we cannot access non static
* It is also called instance variable

**Static** – Static keyword used to declare static.

Eg. Static int x=10;

* These are stored into common memory of class
* If we want to access the variable then
* A.x
* it should be inside class but not in method like global variable.

**Stack memory-**

* main method or any method will be store in stack memory
* once code flow stops then method will removed from stack memory and object address will also removed from heap memory
* code will store in stack

**Heap memory-**

- main method’s object will be stored in heap memory

- Once object is created in heap memory it will remain - After disconnection with the stack memory then the object which is already present it will eligible for GC

- object will store in heap

1. **JDK** – Java Development Kit

It’s the full package you install to develop Java applications.

Contains:

* JRE (Java Runtime Environment)
* Development tools (like javac – the compiler, debugger, etc.)
* Java source code libraries

👨‍💻 Use case: You’re a developer writing and compiling code → You need JDK.

2. **JRE** – Java Runtime Environment

JRE is what’s needed to run Java applications. It includes:

* JVM (Java Virtual Machine)
* Core Java class libraries
* Runtime resources

👨‍🏫 Use case: You just want to run an app (not write or compile) → You need JRE

3. **JVM** – Java Virtual Machine

This is the engine. It runs the compiled .class files (bytecode).

It’s platform-dependent but behaves the same across systems.

Responsibilities:

* Converts bytecode to machine code (Just-In-Time Compiler)
* Memory management (heap/stack)
* Garbage collection
* Security checks
* Multithreading support

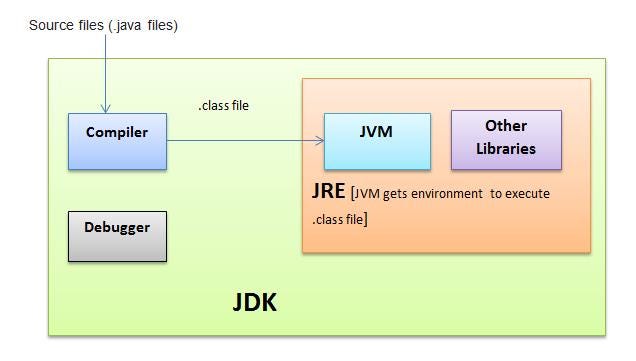
👨‍💼 JVM is part of the JRE, and it is the lowest level runtime.

**📝** Quick Analogy:

* JDK = A kitchen (includes the stove, fridge, chef tools, and ingredients) → you cook
* JRE = Stove + ingredients only → you can heat food, not cook from scratch
* JVM = The fire that actually cooks the food 🔥

📌 Interview Tip: Be able to explain how code flows:

1. Write .java file → JDK (javac) compiles to bytecode → .class file
2. JRE gives you JVM to execute .class
3. JVM uses class loader → memory management → bytecode verification → runs the program



**Abstract keyword**

**🧠 What is abstract in Java?**

In Java, the abstract keyword is used to define:

1. Abstract classes
2. Abstract methods

It’s all about defining something without fully implementing it — a blueprint or contract for subclasses.

**Common Interview Questions**

🔹 Q1: **Can an abstract class have a constructor?**

✅ Yes. Abstract classes can have constructors, and they’re called when a subclass is instantiated.

🔹 Q2**: Can an abstract class have static methods**?

✅ Yes. Abstract classes can have static methods (they must be fully implemented though).

🔹 **Q3: Can we declare an abstract method in a non-abstract class?**

❌ No. If a class has even one abstract method, the class must be declared abstract.

**🔹 Q4: Can an abstract class implement an interface**?

✅ Yes. And it doesn't have to implement all methods — the class can remain abstract and let its subclass implement them.

| **Feature** | **final** | **static** | **abstract** |
| --- | --- | --- | --- |
| Definition | Prevents modification | Belongs to the class, not instance | Incomplete definition, to be implemented by subclasses |
| Applied to | Variables, Methods, Classes | Variables, Methods, Blocks, Classes | Classes, Methods |
| Variable | Value cannot be changed (constant) | Shared across all objects (one copy) | ❌ Not allowed |
| Method | Cannot be overridden | Can be called without object | Declared but not implemented |
| Class | Cannot be extended (no subclassing) | ✅ Allowed (e.g., static nested class) | Cannot be instantiated |
| Inheritance | Prevents inheritance | Not related to inheritance | Enforces inheritance & overriding |
| Purpose | Restriction | Shared behavior/data | Design contract / blueprint |

**🎯 Individual Explanation**

🔵 **final**

* final variable → constant (can be assigned only once)
* final method → cannot be overridden
* final class → cannot be extended

🧪 Example:

final class Animal { } // class Dog extends Animal ❌ Error: Animal is final

final int x = 10; // x = 20; ❌ Error

🔵 **static**

* Belongs to class, not object
* Used for utility methods and shared variables

🧪 Example:

class MyClass { static int count = 0;

static void show() {

System.out.println("Static method");

}

}

MyClass.show(); // no object needed

🔵 **abstract**

* Used for creating abstract classes and methods (no body)
* Meant to be inherited and implemented

🧪 Example:

abstract class Shape { abstract void draw(); // no body }

class Circle extends Shape { void draw() { System.out.println("Drawing circle"); } }

**🧠 Quick Trick for Interviews:**

* Use final when you want to lock things down
* Use static when you want to share or call without objects
* Use abstract when you want to enforce a common structure or contract

**⚔️ Real-Life Analogy:**

Imagine you’re designing a company structure:

* **abstract** **class** = abstract "Employee" → every employee must define how they work (draw() / work())
* **static** = company-wide policies like holiday count → same for everyone
* **final** = permanent values like employee ID → cannot be changed

| **Feature** | **Interface (Java 8+)** | **Abstract Class** |
| --- | --- | --- |
| Keyword | interface | abstract |
| Inheritance Type | Implements | Extends |
| Multiple Inheritance | ✅ Allowed | ❌ Not directly (only single inheritance) |
| Methods | All public abstract (default, static allowed in Java 8+) | Can have abstract + concrete methods |
| Variables | public static final by default | Can have any type (private, protected...) |
| Constructors | ❌ Not allowed | ✅ Allowed |
| Access Modifiers | Only public (for methods) | All access levels allowed |
| Use Case | Contract: “What a class must do” | Partial implementation: “How and what to do” |

**Both Interface and Abstract Class help in achieving abstraction — hiding implementation details.**

**🛻 Project: Second-Hand Car Portal System**

You have different types of users interacting with your platform:  
→ Buyer, Seller, Admin, Dealer, Mechanic, etc.

Also, you have different vehicle types like Car, SUV, Truck, etc.

Let’s look at two practical examples:

—

🧩 **Example 1: User Abstraction**

We can create an abstract class User:

abstract class User { String name; String email;

void login() {

System.out.println("User logged in with email: " + email);

}

abstract void performRole(); // each user role behaves differently

}

👨 Buyer class:

class Buyer extends User { void performRole() { System.out.println("Buyer can search and view car listings."); } }

👨‍🔧 Dealer class:

class Dealer extends User { void performRole() { System.out.println("Dealer can post new car listings."); } }