🚀 Java Core

1. **Explain OOPS Concepts with examples?**

Encapsulation

- Definition:
  Encapsulation is the process of binding data (variables) and methods (functions) together into a single unit (class) and restricting direct access to some of the object's components.

Selenium Example:
The WebDriver API is a great example. Selenium provides us with methods like click() and sendKeys() but hides the internal implementation from the user.
Inheritance

- Definition:
  Inheritance is the mechanism by which one class (child/subclass) acquires the properties and behaviors of another class (parent/superclass).

Selenium Example:
The BaseTest class in Selenium frameworks is typically extended by all test classes to inherit common setup, teardown, and utility methods.
Polymorphism

- Definition:
  Polymorphism means many forms. It allows a single interface to represent different types of actions.

Selenium Example:
The List of WebElements (List<WebElement>) in Selenium is polymorphic because the same action (like click()) can be performed on different elements using the same method signature.
Abstraction

- Definition:
  Abstraction is the process of hiding unnecessary implementation details and showing only the essential features to the user.

Selenium Example:

In Selenium frameworks, Page Object Model (POM) is a great example. Each Page class exposes only relevant actions to the test class while hiding complex locators and element interactions.

2. **What is encapsulation, and how does it improve security?**

Encapsulation is the process of binding data (variables) and methods (functions) into a single unit (class) and restricting direct access to some of the object's components. This is typically achieved by making variables private and providing controlled access through public getter and setter methods.

Examples: Getter and Setters.

3. **What is abstraction and how does it help in software design?**

Abstraction is the process of hiding the internal implementation details and showing only the essential features to the user. It simplifies software design by focusing on what an object does rather than how it does it.

4. **Can we achieve abstraction without using abstract classes?**

Yes, in Java, we can achieve abstraction without using abstract classes by using interfaces.
Interfaces provide a way to define a contract (method declarations) without specifying how those methods should be implemented.
When a class implements an interface, it provides the concrete behavior for the methods, achieving abstraction by exposing only essential details to the user and hiding the implementation.

```
interface Animal {
   void makeSound();  // abstract method
}
class Dog implements Animal {
   public void makeSound() {
      System.out.println("Bark");
   }
}
```

5. **What is inheritance and how does it promote code reusability?**

Inheritance is an **OOP principle** where a subclass can acquire the properties and behaviors (methods and variables) of a parent class using the extends keyword.
It promotes **code reusability** because we can write common logic once in the parent class and reuse it in multiple subclasses without rewriting it.
In Selenium, we often create a BaseTest class where we:

- Declare WebDriver setup in the @BeforeMethod.

  Close the browser in the @AfterMethod.

6. **What is the diamond problem in inheritance?**

The diamond problem in inheritance occurs when a class inherits from two parent classes that share a common superclass, creating ambiguity about which method to inherit.
Java avoids this problem by not supporting multiple inheritance with classes. It allows multiple inheritance via interfaces, where conflicts can be resolved explicitly.

7. **Difference between compile-time and runtime polymorphism.**

Feature	Compile-Time Polymorphism	Run-Time Polymorphism
Also Known As	Method Overloading	Method Overriding
Decision Timing	Resolved at Compile-Time	Resolved at Run-Time
Method Signature	Same method name, different parameters	Same method signature
Relationship	Can exist in the same class	Requires inheritance
Example	add(int a, int b)add(double a, double b)	WebDriver driver = new ChromeDriver();Driver-specific get() method runs

8. **Can we override static methods?**

No, **we cannot override static methods** in Java. This is because **static methods belong to the class**, not to instances of the class. Overriding is a concept that applies to **instance methods** — where the method that gets executed is determined at runtime based on the object's actual class.
However, if a subclass defines a static method with the same signature as a static method in the parent class, it's called **method hiding**, not overriding.

9. **Can we override a private method in Java?**

No, private methods cannot be overridden in Java because they are not inherited by child classes.

10. **Can we override a method that throws an exception with a method that does not?**

Yes, it is allowed.
In Java, when overriding a method:

- The overriding method in the child class can throw fewer exceptions, narrower (more specific) exceptions, or no exception at all.

  It cannot throw broader or new checked exceptions that are not declared in the parent method.

11. **What is dynamic method dispatch?**

Dynamic Method Dispatch is the process by which a call to an overridden method is resolved at runtime, rather than at compile time.
It is a key mechanism that enables runtime polymorphism in Java.
When a parent class reference variable is used to refer to a child class object, Java determines which version of the overridden method to execute at runtime based on the actual object type.

12. Can we create an object of an abstract class?

No, we cannot create an object of an abstract class in Java.
The reason is that an **abstract class is incomplete by design** — it may contain one or more **abstract methods** (methods without implementation), which must be implemented by a concrete subclass.
Since the abstract class cannot provide full behavior on its own, Java does **not allow direct instantiation** of it.

13. Difference between abstract class and interface.

1. a class can only extend **one abstract class**.
   Java **supports multiple inheritance** via interfaces.
2. Can have **both abstract and non-abstract (concrete) methods**.
   From Java 8 onward, it can have **default and static methods**, but **no regular instance methods**.
3. Can have **instance variables** and maintain **state**.
   — no instance variables allowed.
4. Can have **constructors**, which are useful for initializing common code.
   Cannot have constructors — because interfaces **cannot be instantiated** or hold state.

14. What is a functional interface?

A **functional interface** in Java is an interface that contains **exactly one abstract method**. It may have **any number of default or static methods**, but only **one abstract method** defines its "functionality."

15. Lambda expressions and usage.

A **lambda expression** is an **anonymous function** — a shorthand way to implement a **functional interface**.

```
(parameter1, parameter2, ...) -> { // method body }
```

16. How to prevent a method from being overridden.

To **prevent a method from being overridden** in Java, you simply use the **final keyword**.

17. **What is the difference between class and object?**

A **class** in Java is a **blueprint or template** for creating objects. It defines the structure (fields) and behavior (methods) that the objects created from it will have.
An **object** is a **real instance** of a class — it occupies memory and can be used to call methods or access variables defined in the class.

B. Constructors & Keywords

1. **What are constructors in Java? Types?**

Constructors in Java are special methods that are used to initialize objects. They are automatically called when an object of a class is created.
A constructor:

- Must have the same name as the class.
- **Does not** have a return type (not even void).
- Can be **overloaded** (multiple constructors with different parameters).

Types of Constructors:

1. **Default Constructor** – Takes no arguments.
2. **Parameterized Constructor** – Takes parameters to initialize an object with specific values.

2. **Why use constructors?**

Constructors in Java are used to initialize objects when they are created. Every time an object is instantiated using the new keyword, a constructor is automatically called.
If you do not define any constructor in a class, Java provides a **default constructor** that initializes the object with default values. You can also define your own constructors (called **parameterized constructors**) to initialize an object with specific values at the time of creation.

3. **Explain constructor chaining.**

Constructor chaining is the process of calling one constructor from another **within the same class** or **from the parent class** using the this() or super() keywords respectively.
It helps in **avoiding code duplication** and maintaining a **cleaner and more organized constructor structure**.

4. **Can a constructor be final,static in Java?**

No, a constructor cannot be declared as final in Java. The purpose of the final keyword is to prevent method overriding, but constructors are **not inherited**, so there's no sense in marking them as final.
Similarly, constructors **cannot be static** because they are used to create instances of a class, while static methods and blocks belong to the class itself, not instances.
However, a constructor **can be private**, which is useful in design patterns like **Singleton**, where object creation is restricted.

5. **What is this keyword?**

In Java, the this keyword is a reference variable that refers to the **current object** of the class. It is commonly used to:

1. **Differentiate instance variables from local variables** when they share the same name.
2. **Call another constructor** in the same class (this()).

6. **What is super keyword?**

The super keyword in Java is used to refer to **members (variables, methods, and constructors)** of the **immediate parent class**.
It helps in:

1. **Accessing parent class fields** that are hidden by subclass fields.
2. **Invoking the parent class methods** that are overridden.
3. **Calling the parent class constructor** from the subclass constructor.

7. **Difference between final, finally, and finalize.**

The final, finally, and finalize keywords in Java serve different purposes. The final keyword is used to restrict modification: a final variable cannot be reassigned, a final method cannot be overridden, and a final class cannot be extended. The finally block is used in exception handling and ensures that a block of code is executed regardless of whether an exception is thrown or not—commonly used for resource cleanup. The finalize() method was part of the Object class and was invoked by the garbage collector before an object was removed from memory, allowing for cleanup operations, but it has been deprecated since Java 9 and removed in later versions, making it obsolete for modern Java development.

8. **Difference between final and static.**

The final keyword is used to **restrict modification**. It can be applied to variables (which means their value cannot be changed), methods (which means they cannot be overridden), and classes (which means they cannot be extended). On the other hand, the static keyword indicates that a member (variable or method) belongs to the **class rather than the instance**. That means static members are shared among all objects of the class. A key difference is that final enforces immutability or restriction, while static provides **shared access** across instances. Also, static methods cannot be overridden in the true polymorphic sense—they are hidden if re-declared in a subclass.

9. **What happens if we create a static method?**

When you create a static method in Java, it belongs to the class rather than any instance of the class, meaning it can be called without creating an object. Static methods are often used for utility or helper functions, like those found in the Math or Collections classes. Because they are tied to the class, static methods cannot directly access instance variables or instance methods—they can only interact with other static members. Additionally, static methods cannot be overridden in the same way as instance methods; instead, if a subclass defines a static method with the same signature, it hides the parent class method rather than overriding it. This is known as method hiding. Overall, static methods are useful when behavior is not dependent on object state and should be shared across all instances.

10. **Can we execute a Java program without the main() method?**

In modern versions of Java (Java 7 and above), **you cannot execute a standalone Java program without the main() method**. The JVM (Java Virtual Machine) looks for the main(String[] args) method as the entry point to start program execution.

11. **Can we have multiple main() methods in different classes?**

Yes, **we can have multiple main() methods in different classes** in Java.

12. **Can we declare the main method as private?**

No, **we cannot declare the main() method as private** in Java if we want the program to run normally.

C. Exception Handling

1. **Can a try block have multiple catch blocks?**

Yes, a try block in Java can have multiple catch blocks, and this is done to handle different types of exceptions separately. When an exception occurs, the JVM looks for the first matching catch block based on the type of exception thrown. Once a matching block is found, the remaining catch blocks are skipped.

2. **Can we have a try block without a catch block?**

Yes, in Java, you can have a try block without a catch block, as long as it is followed by a finally block. This is valid because the finally block is guaranteed to execute, regardless of whether an exception is thrown or not.

3. **Can we catch multiple exceptions in a single catch block?**

Yes, in Java, you can catch multiple exceptions in a single catch block using the multi-catch syntax (introduced in Java 7).

4. **What happens if we throw an exception in a finally block?**

If an exception is thrown in the finally block, it will override any exception that may have been thrown in the try or catch block. It can lead to loss of the original exception, which may make debugging difficult.

5. **Exception handling in Java and custom exceptions.**

Exception handling in Java is a mechanism that allows us to gracefully handle runtime errors, ensuring that the normal flow of the program is not interrupted. Java uses try-catch-finally blocks to catch and handle exceptions.

6. **Difference between checked and unchecked exceptions.**

Feature	Checked Exceptions	Unchecked Exceptions
Definition	Exceptions that are checked at **compile time**	Exceptions that occur at **runtime**
Inheritance	Subclasses of Exception (but **not** RuntimeException)	Subclasses of RuntimeException
Compiler Requirement	Must be **either handled** using try-catch or **declared** using throws keyword	No need to explicitly handle or declare them
Examples	IOException, SQLException, ParseException, ClassNotFoundException	NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, IllegalArgumentException
When to Use	For **recoverable conditions** – where the program can recover	For **programming errors** or **logic flaws** that should be fixed in code
Handling	Developer is forced to handle these	Developer **may choose** to handle them

7. **Difference between throw and throws.**

Feature	throw	throws
Purpose	Used to explicitly throw an exception	Declares what exceptions a method may throw
Usage	Inside a method or block	In the method declaration/signature
Syntax	throw new ExceptionType("message");	public void methodName() throws ExceptionType
Can throw	Only one exception at a time	Can declare multiple exceptions
Object type	Followed by an instance of Throwable	Followed by one or more exception classes
Example	throw new ArithmeticException("/ by zero");	public void readFile() throws IOException

throw new ExceptionType("Custom error message");

returnType methodName() throws ExceptionType1, ExceptionType2 {

D. Java Memory Management

1. **Stack and heap memory areas.**

In Java, stack memory stores method call frames, local variables, and parameters in a LIFO (last-in-first-out) manner. It is thread-specific, fast, and memory is automatically released when methods end.

Heap memory, on the other hand, is used to store objects and is shared across threads. Objects created with new are stored in the heap, and memory is managed by the Garbage Collector. Heap is larger and suitable for data needing a longer lifespan than a single method call.

2. **System.out.println() definition.**

System.out.println() is a Java statement used to print text or values to the console, followed by a newline.

- **System** is a built-in Java class from the java.lang package.
- **out** is a static member of System, representing the standard output stream (usually the console).
- **println()** is a method of the PrintStream class (the type of out) that prints the passed argument and then moves the cursor to a new line.

E. Java Core Syntax and Behavior

1. **Access Modifiers: public, private, protected, default.**

In Java, **access modifiers** define the visibility or accessibility of classes, methods, and variables. There are four main types:

1. **public**:
   Accessible from **anywhere** in the program — within the same class, other classes, other packages.
   Example: A public method in a class can be called from any other class.
2. **private**:
   Accessible **only within the same class**.
   Example: A private variable cannot be accessed from outside its class, even in subclasses.
3. **protected**:
   Accessible **within the same package** and also in **subclasses (even in different packages)**.
   Example: Useful for inheritance when you want to allow limited access to derived classes.
4. **Default (no modifier)**:
   When no access modifier is specified, it is **package-private** by default — accessible **only within the same package**.
   Example: A method with no modifier cannot be accessed outside its package.

Modifier	Same Class	Same Package	Subclass (diff package)	Other Classes

public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

2. **Why Java is not 100% object-oriented.**

Java is **not 100% object-oriented** because it uses **primitive data types** (like int, char, boolean, byte, float, etc.) which are **not objects**.

3. **What is == vs. .equals()?**

== compares object references, while .equals() compares object content.

4. **Difference between String, StringBuilder, and StringBuffer.**

   String is immutable and thread-safe.

   StringBuilder is mutable and not thread-safe (faster for single-threaded).

   StringBuffer is mutable and thread-safe (slower due to synchronization).

5. **Why is String immutable in Java?**

String is immutable in Java to ensure security, caching, synchronization, and safe sharing across threads without unintended side effects.

6. **How can we achieve immutability in Java?**

To achieve **immutability** in Java, you need to design a class whose instances **cannot be changed** once they are created.

Declare the class as final
Prevents the class from being subclassed.

Make all fields private and final
Prevents direct modification and ensures they are set only once.

Do not provide setter methods
Set values only once using the constructor.

Initialize all fields in the constructor
Assign values when the object is created.

If a field is a mutable object (like Date, List), return a copy instead of the original
Prevents clients from modifying internal objects.

7. **Why String is not a primitive data type?**

   "String is not a primitive data type in Java because it represents a sequence of characters and comes with a rich set of methods like length(), substring(), toUpperCase(), etc., which are only possible if it is implemented as a class.

8. **Difference between Array and ArrayList.**

Feature	Array	ArrayList
Size	Fixed once declared	Dynamic — can grow or shrink
Type	Can hold both primitive & objects	Can hold only objects (non-primitives)
Syntax	int[] arr = new int[5];	ArrayList<Integer> list = new ArrayList<>();

Performance	Slightly faster (less overhead)	Slightly slower (more flexible)
Memory Allocation	Static memory allocation	Dynamic memory allocation
Resizing	Manual (create new larger array & copy)	Handled internally by Java
Part of	Core Java language	java.util package
Methods Available	No built-in methods (only via loops)	Rich set of methods (add, remove, size, etc.)
Generics Support	No	Yes

9. **Difference between Array and Collection.**

Feature	Array	Collection (like ArrayList, HashSet, etc.)
Size	Fixed — must be declared during initialization	Dynamic — grows/shrinks automatically
Type of Data	Can store both primitives and objects	Can store only objects (not primitives like int, char)
Performance	Faster — less overhead	Slightly slower — but more powerful
Flexibility	Rigid — no in-built support for complex operations	Flexible — supports many operations (add, remove, sort, etc.)
Part of	Core Java (no import needed)	java.util package (must import)
Data Structure Support	Just linear arrays	Many types: List, Set, Queue, Map (via Collection Framework)
Methods Available	No built-in methods (use loops)	Rich set of utility methods like add(), remove(), contains()
Usage	Good for static data or performance-critical tasks	Good for dynamic data and complex operations

10. Difference between Collection and Collections.

Feature	Collection (Interface)	Collections (Utility Class)
Type	Interface	Class (final and static utility class)
Package	java.util	java.util
Inheritance	Super interface for List, Set, Queue, etc.	Does **not** implement Collection
Purpose	Root interface for collection types	Utility class for common algorithms on collections
Contains	Method declarations (like add(), remove())	Static methods (like sort(), reverse(), max())
Usage	Used to define data structures	Used to operate on data structures
Example	Collection<String> list = new ArrayList<>();	Collections.sort(list);

11. Difference between List, Set, and Map.

Feature	List	Set	Map
Type	Interface extending Collection	Interface extending Collection	Interface (not a Collection)
Stores	Ordered collection of **elements**	Unordered collection of **unique elements**	Collection of **key-value pairs**
Duplicates	✅ Allows duplicates	❌ No duplicates	✅ Keys must be unique; values can repeat
Order	Maintains insertion order	Doesn't guarantee order (except LinkedHashSet)	Order depends on implementation
Access by index	✅ Yes (e.g., get(index))	❌ No	✅ Access via key (get(key))

Common Classes	ArrayList, LinkedList, Vector	HashSet, LinkedHashSet, TreeSet	HashMap, LinkedHashMap, TreeMap

12. Explain the Java Collection Framework.

The Java Collection Framework is a set of interfaces, classes, and algorithms that help you store, retrieve, and manipulate groups of objects efficiently.
Think of it like a toolbox in Java to handle data structures such as:

- Lists
- Sets
- Maps
- Queues
- Stacks

Interface	Description
Collection	Root of the collection hierarchy.
List	Ordered collection that allows duplicates.
Set	Unordered collection with no duplicates.
Queue	Elements processed in FIFO order.
Deque	Double-ended queue (insert/remove from both ends).
Map	Not part of Collection interface. Stores key-value pairs.

Interface	Implementation Classes	Description
List	ArrayList, LinkedList, Vector	Maintain order, allow duplicates.
Set	HashSet, LinkedHashSet, TreeSet	Unique elements only.
Map	HashMap, LinkedHashMap, TreeMap	Key-value mapping.
Queue	PriorityQueue, ArrayDeque	FIFO logic.

13. Difference between HashMap and Hashtable.

Feature	HashMap	Hashtable
Thread-Safety	❌ Not synchronized (not thread-safe)	✅ Synchronized (thread-safe)
Performance	Faster (no synchronization overhead)	Slower (due to synchronization)
Null Keys/Values	✅ Allows one null key and multiple null values	❌ Does NOT allow null keys or values
Introduced In	Java 1.2	Java 1.0
Inheritance	Extends AbstractMap	Extends Dictionary (legacy class)
Fail-Fast	✅ Iterators are fail-fast (throws ConcurrentModificationException if modified during iteration)	❌ Enumerators are not fail-fast
Usage in Modern Java	Preferred in single-threaded or externally synchronized code	Legacy class, rarely used in new code

14. Difference between HashMap and TreeMap.

Feature	HashMap	TreeMap
Ordering	❌ No guaranteed order of elements	✅ Sorted by natural order of keys or a custom comparator

Null Keys/Values	✅ Allows one null key and multiple null values	❌ Does NOT allow null keys (but allows null values)
Implementation	Based on a hash table	Based on a red-black tree
Performance	Faster in terms of basic operations (put, get, remove)	Slower due to sorting overhead (O(log n) for basic operations)
Thread-Safety	❌ Not synchronized	❌ Not synchronized (same as HashMap)
Time Complexity	O(1) for most operations (average)	O(log n) for most operations due to sorting
Use Case	Best when you don't need ordering and just want fast lookups	Best when you need a sorted order of keys (ascending by default)

15. How does HashSet ensure uniqueness?

When you add an element to a HashSet, it stores that element as the key in the internal HashMap.

16. How does LinkedHashMap maintain insertion order?

LinkedHashMap maintains insertion order by using a doubly linked list that links all entries (key-value pairs) in the order they were inserted.

17. What is a Singleton Design Pattern?

The **Singleton Design Pattern** ensures that a **class has only one instance** in the entire application and provides a **global access point** to that instance.

Examples: Runtime.getRuntime()
Logger class
Spring Beans by default are Singletons

18. What is Vector class in Java?

The Vector class is a legacy class in Java that implements the List interface and is part of the Java Collection Framework.
It is synchronized, meaning it is thread-safe by default.

F. Type Casting, Boxing, Data Types

1. **What is type casting and its types?**

Type casting in Java is the process of converting one data type into another, either automatically by the compiler or manually by the programmer.

Implicit Type Casting (Widening Conversion): Converts smaller (lower) data types to larger (higher) data types.

Explicit Type Casting (Narrowing Conversion): Converts larger data types to smaller ones.

1. **Type casting in Selenium (JavascriptExecutor example).**

In Selenium WebDriver (Java), type casting is often used when we want to access methods from child interfaces like JavascriptExecutor, TakesScreenshot, etc., which are not available in the base WebDriver interface.

2. **What is unboxing and autoboxing? (With Selenium examples)**

In Java, **autoboxing** and **unboxing** are features related to **wrapper classes** that help convert between **primitive data types** and **their object counterparts** automatically.

Concept	Description
Autoboxing	Automatic conversion of **primitive → wrapper object**
Unboxing	Automatic conversion of **wrapper object → primitive**

3. **Wrapper classes and their importance in Selenium.**

Wrapper classes are Java classes that **wrap primitive data types** (like int, char, boolWean) into **objects** so that they can be used in Java collections, generics, and other object-oriented constructs.

In Selenium automation (Java-based), wrapper classes are useful in the following scenarios:

1. Storing Primitive Values in Collections.

```
List<Integer> positions = new ArrayList<>();
positions.add(element.getLocation().getX()); // int → Integer (Autoboxing)
```

2. Working with JSON APIs or External Data

When integrating REST APIs or reading JSON/XML test data, values are returned as wrapper types (Integer, Boolean).

```
Integer statusCode = response.getStatusCode(); // from REST Assured
```

4. **What are primitive data types?**

Primitive data types in Java are the most basic built-in types provided by the language. They **hold simple values** (not objects) and are **not derived** from any class.

These types are **faster** and use **less memory** compared to objects, making them ideal for performance-critical tasks.

5. **Why is String not a primitive type?**

String is not a primitive data type in Java because it is a class, not a built-in type. It provides methods and behaviors, which primitive types do not support.

G. Multithreading and Synchronization

1. **What is synchronization in Java?**

Synchronization in Java is a mechanism used to control access to shared resources by multiple threads in a multithreaded environment. Its primary purpose is to prevent data inconsistencies and race conditions that can arise when multiple threads attempt to modify the same data concurrently.

Key Concepts of Java Synchronization:

- **Shared Resources:**
  These are data or objects that can be accessed and modified by multiple threads, such as instance variables, static variables, or files.
- **Race Conditions:**
  This occurs when the outcome of a program depends on the unpredictable timing of multiple threads accessing and modifying shared resources, potentially leading to incorrect results.
- **Thread Safety:**
  A class or method is considered thread-safe if it can be used concurrently by multiple threads without causing data corruption or unexpected behavior. Synchronization is a key tool for achieving thread safety.

2. **What is multithreading in Java? Real-time multithreading.**

Multithreading is a Java feature that allows the **concurrent execution of two or more threads** (smaller units of a process). It helps you perform multiple tasks simultaneously to make better use of CPU resources.

Key Concepts:

- A **thread** is a lightweight subprocess.
- Java's multithreading is built on the **java.lang.Thread** class and **Runnable** interface.
- Threads can run **independently**, but share the same memory space.
- **Main goal**: Improve performance and responsiveness, especially in high-load or I/O-bound applications.

Can You Override Static Block?

- Static blocks execute during class loading and **cannot be overridden** (only hidden via re-declaration in subclasses).

Can We Make a Class Static?

- Only **nested classes** can be static (e.g., static class Nested). Top-level classes cannot be static.

Difference Between Composition and Aggregation.

- **Composition**: Child objects cannot exist without the parent (e.g., Car and Engine).
- **Aggregation**: Child objects exist independently (e.g., Department and Employees).

Why is the Collection Framework Introduced in Java?

- To standardize data structure handling, improve performance, and provide reusable algorithms (e.g., Collections.sort()).

What is the Difference Between ArrayList and LinkedList?

- **ArrayList**: Faster random access (index-based).
- **LinkedList**: Faster insertions/deletions (node-based).

What is the Difference Between HashSet and TreeSet?

- **HashSet**: Unordered, faster (O(1) average).
- **TreeSet**: Sorted order, slower (O(log n) due to red-black tree).

How Does Java Handle Method Resolution in Multiple Interfaces?

- If two interfaces have identical default methods, the implementing class **must override** the method to resolve ambiguity.