My self Anjali D J, I'm having 4 years of experince in java development and coming to technical stack currentely I'm working on Java , j2ee , spring, hibernate,springboot and restfull web service . working on the tools like build tool like maven and code repository like git and svn and ticket tracking tool JIRA .

Currently we are working on agile methodology process . every 2 weeks we will be having sprints before starting the sprint we will join the grooming call and estimate the story points after that we will start the sprint . every we will join the scrum call and disccusing about our user stories .

Once user story started we will work on development after that will write the Junit test casses for scenarios after that we will raise the PR . once PR raised i will assign the pR to lead and manager after they will and approve the code changes . once PR got approved
we will be deploying the changes by using CI/CD pipe line by using the jenkins .

Once Deployment is done we will be creating test enviroment url and pointing the test envorment url we will testing our changes . that's all pretty much about me

Java_

1) how to create own Immutable class ?
An **immutable class** is a class whose **object data cannot be changed after creation**.

👉 Once you create the object → its values stay **fixed forever**.

Rules to Create Your Own Immutable Class (VERY IMPORTANT ⭐)

In interviews, say these **5 rules** clearly:

1. **Make class final**
   → so no one can extend and modify it
2. **Make all variables private and final**
3. **Initialize variables using constructor only**
4. **Do NOT provide setter methods**
5. **For mutable objects, return a COPY (defensive copy)**

```
public final class Employee {

  private final int id;

  private final String name;
```

```
    // Constructor
    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
    // Only getters
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
}
```

```
2) Imuttable class will have inside muttable object ?  how we can overcome
this problem
Example : class Employee{
      private int id;
      private String name;
      private double salary ;
      private Address address;
```

Yes ✅ an immutable class **can contain a mutable object**, **BUT** you must handle it carefully.

To maintain immutability, we must use **defensive copying** by creating copies of mutable objects in the constructor and returning copies in getter methods instead of the original reference.

Defensive copying is a technique where we create and return copies of mutable objects to protect the internal state of a class from external modification.

```
public final class Employee {

    private final Address address;

    public Employee(Address address) {
        this.address = new Address(address.getCity()); // defensive copy
    }
```

```
    public Address getAddress() {
        return new Address(address.getCity()); // defensive copy
    }
}
```

3)why is String is immutable

Once a String object is created, **its value cannot be changed**.

String s = "Java";

s.concat(" World");

System.out.println(s); // Java

👉 "Java World" is created as a **new object**.
Original "Java" stays unchanged.

String is immutable to ensure security, enable string constant pool memory optimization, provide thread safety, allow hashcode caching, and maintain JVM integrity.

4)How we can create our own exceptions in project ?

A **custom (user-defined) exception** is an exception **created by us** to represent **project-specific errors**.

👉 Example:

- User not found
- Insufficient balance
- Invalid policy ID
- Employee already exists

Why Do We Need Custom Exceptions? (Interview Reason)

- To give **meaningful error messages**

- To handle **business logic errors**
- To avoid generic exceptions like Exception
-

Step 1: Decide the Type of Exception

There are **two types** of custom exceptions:

1️⃣ Checked Exception

➞ Extend Exception
➞ Must be handled using try-catch or throws

2️⃣ Unchecked Exception (Most used in projects ✅)

➞ Extend RuntimeException
➞ No compile-time restriction

👉 **In real projects, we usually create unchecked exceptions.**

Step 2: Create the Custom Exception Class

Example: InvalidAgeException (Unchecked)

```
public class InvalidAgeException extends RuntimeException {


  public InvalidAgeException(String message) {
    super(message);
  }
}
```

✓ Class name ends with Exception
✓ Extends RuntimeException

Step 3: Throw the Exception Where the Problem Occurs

```
public class VoterService {


  public void checkEligibility(int age) {


    if (age < 18) {
      throw new InvalidAgeException("Age must be 18 or above");
```

```
        }

        System.out.println("Eligible for voting");
    }
}
```

Step 4: Handle the Exception (Optional in Core Java)

Using try-catch

```
public class Test {

    public static void main(String[] args) {

        VoterService service = new VoterService();

        try {
            service.checkEligibility(16);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Step 5: (Optional) Create Checked Custom Exception

Example: LowBalanceException

```
public class LowBalanceException extends Exception {

    public LowBalanceException(String message) {
        super(message);
    }
}
```

Usage

```
public void withdraw(int amount) throws LowBalanceException {
  if (amount > 1000) {
    throw new LowBalanceException("Insufficient balance");
  }
}
```

Steps Summary (Interview Answer 🔥)

1. Decide checked or unchecked exception
2. Create a class extending Exception or RuntimeException
3. Provide a constructor with message
4. Throw the exception using throw keyword
5. Handle it using try-catch or throws (if required)

`5) can we write try block with finally ?`
Yes, we can write a try block with only finally. Catch is optional, but try must be followed by either catch or finally.

Why do we use try with finally?

Because **finally always executes**, whether:

- exception occurs or not
- exception is caught or not
- method returns or not

```
try {
  // risky code
} finally {
  // cleanup code
}
```

6)Exception overriding rules on parent and child classes ?

While overriding, a child method cannot throw broader checked exceptions than the parent, but it can throw the same, a subclass, or no exception at all. Unchecked exceptions have no such restriction.

Rule 1️⃣ : Child CANNOT throw broader checked exception ❌

Parent

```
class Parent {
   void show() throws IOException {
   }
}
```

Child (INVALID)

```
class Child extends Parent {
   void show() throws Exception { // ❌ broader exception
   }
}
```

📌 Compile-time error

Rule 2️⃣ : Child CAN throw same checked exception ✅

```
class Child extends Parent {
   void show() throws IOException { // ✅ same
   }
}
```

Rule 3️⃣ : Child CAN throw subclass of parent exception ✅

```
class Child extends Parent {
   void show() throws FileNotFoundException { // ✅ subclass
   }
}
```

(FileNotFoundException is subclass of IOException)

Rule 4️⃣ : Child CAN throw NO exception at all ✅

```
class Child extends Parent {
```

```
    void show() {  //  no exception
    }
}
```

Rule 5 : If parent does NOT throw checked exception, child CANNOT throw checked exception ❌

Parent

```
class Parent {
    void display() {
    }
}
```

Child (INVALID)

```
class Child extends Parent {
    void display() throws IOException {  //  not allowed
    }
}
```

7) can we have try with Multiplecatch blocks ?
Yes, a try block can have multiple catch blocks, but they must be ordered from specific to general, and only one catch block executes.

```
public class Demo {
    public static void main(String[] args) {

        try {
            int a = 10 / 0;          // ArithmeticException
            int[] arr = new int[2];
            System.out.println(arr[5]);  // ArrayIndexOutOfBoundsException
```

```
    } catch (ArithmeticException e) {
        System.out.println("Arithmetic exception occurred");

    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index exception occurred");

    } catch (Exception e) {
        System.out.println("Some other exception");
    }
  }
}
```

8)try with with Multiplecatch blocks shoud write the parent exception (i..e Exception) at last catch ortherwise will get compiletime error

When using **multiple catch blocks**, the **parent exception (Exception) must be written last**, otherwise the compiler throws a **compile-time error**.

❌ **Wrong Order (Compile-Time Error)**

```
try {
   int a = 10 / 0;
}
catch (Exception e) {          // parent first ❌
   System.out.println("Exception");
}
catch (ArithmeticException e) {     // child later ❌
   System.out.println("Arithmetic");
}
```

❗ **Why compile-time error?**

Because:

- Exception can already catch **all exceptions**
- ArithmeticException will **never be reached**

✅ **Correct Order**

```
try {
   int a = 10 / 0;
}
catch (ArithmeticException e) {      // child first ✅
   System.out.println("Arithmetic");
}
catch (Exception e) {            // parent last ✅
   System.out.println("Exception");
}
```

9)try with resorces ?

try-with-resources is a Java feature that automatically closes resources that implement AutoCloseable, eliminating the need for explicit finally blocks.

```
class MyResource implements AutoCloseable {

   public void open() {
      System.out.println("Resource opened");
   }

   @Override
   public void close() {
      System.out.println("Resource closed");
   }
}
```

10)differnce between final,finally and finalize?
final restricts modification, finally is used for cleanup in exception handling, and finalize() is a garbage collection method called before object destruction.

1️⃣ **final (Keyword)**
Used to **restrict modification.**
Usage:

- **final variable** → value cannot change
- **final method** → cannot be overridden
- **final class** → cannot be inherited

2️⃣ **finally (Block)**
Used in **exception handling.**
Purpose:

- Executes **always**
- Mainly used for **resource cleanup**

finalize() (Method)
Called by **Garbage Collector** before destroying an object.
⚠️ **Deprecated / not reliable** (avoid in real projects)

11) can we handle error ?
Errors can be caught because they are throwable, but they should not be handled as they indicate serious JVM-level problems that applications are not expected to recover from.

```
try {
   throw new StackOverflowError();
} catch (Error e) {
   System.out.println("Error caught");
}
```

12)when will get OutOfMemoryError ? when will get StackOverflowError
1️⃣ **OutOfMemoryError**

Definition:
Occurs when JVM **cannot allocate memory** for an object because heap memory is full.

Causes:

1. Creating **too many objects** (memory leak)
2. Large **arrays or collections**
3. Infinite data accumulation

2️⃣ StackOverflowError

Definition:
Occurs when **stack memory is full**, usually due to **infinite or very deep recursion**.

Causes:

1. Infinite recursion
2. Very deep method calls

`13) types of exceptions`
13) Types of Exceptions

Exceptions in Java are of two types: checked (must be handled at compile-time) and unchecked (RuntimeException, occurs at runtime). Errors represent serious JVM issues like OutOfMemoryError or StackOverflowError.

Java exceptions are broadly divided into **two categories**:

1️⃣ Checked Exceptions ✅

- Must be **handled** using try-catch or declared using throws
- Known at **compile-time**
- Usually **recoverable**

Examples:

- IOException → File not found

- SQLException → DB errors
- ClassNotFoundException

```
try {
   FileInputStream fis = new FileInputStream("abc.txt");
} catch (IOException e) {
   e.printStackTrace();
}
```

2️⃣ Unchecked Exceptions ✅

- **Do NOT need to be declared or caught**
- Known at **runtime**
- Usually **programming errors**

Examples:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException

```
int a = 10 / 0; // ArithmeticException
```

14) what is Seralization ? how we can seralize the Employee object into file

14) What is Serialization?
Definition (Simple):
Serialization is the process of **converting a Java object into a byte stream** so that it can be **saved to a file, sent over a network, or stored in a database.**
Why use it?

- Save object state
- Transfer objects over network (RMI, sockets)
- Store objects persistentl

- # How to Serialize an Object
- **Step 1:** Make your class **implement `Serializable`**
- ```
  import java.io.Serializable;
  
  public class Employee implements Serializable {
      private int id;
      private String name;
      private double salary;
  
      public Employee(int id, String name, double salary) {
          this.id = id;
          this.name = name;
          this.salary = salary;
      }
  
      // Getters
      public int getId() { return id; }
      public String getName() { return name; }
      public double getSalary() { return salary; }
  }
  ```
- **Step 2:** Use `ObjectOutputStream` to write object to a file
- ```
  import java.io.FileOutputStream;
  import java.io.ObjectOutputStream;
  
  public class SerializeDemo {
      public static void main(String[] args) throws Exception {
  
          Employee emp = new Employee(101, "Anjali", 50000);
  
          // Step 2: Create file output stream
          FileOutputStream fos = new FileOutputStream("employee.ser");
  
          // Step 3: Create ObjectOutputStream
          ObjectOutputStream oos = new ObjectOutputStream(fos);
  
          // Step 4: Serialize object
          oos.writeObject(emp);
  
          oos.close();
          fos.close();
  
          System.out.println("Employee object serialized
  successfully!");
      }
  }
  ```

15)what is de-Seralization? how we can de-seralize object from file

Deserialization is the process of **reading the byte stream and converting it back into a Java object**.

Basically, it's **reverse of serialization**.

```
import java.io.FileInputStream;

import java.io.ObjectInputStream;


public class DeserializeDemo {

  public static void main(String[] args) throws Exception {


    // Step 1: File input stream

    FileInputStream fis = new FileInputStream("employee.ser");


    // Step 2: Object input stream

    ObjectInputStream ois = new ObjectInputStream(fis);


    // Step 3: Read object

    Employee emp = (Employee) ois.readObject();


    ois.close();

    fis.close();


    // Step 4: Access object data

    System.out.println("ID: " + emp.getId());

    System.out.println("Name: " + emp.getName());

    System.out.println("Salary: " + emp.getSalary());

  }

}
```

16)what is serailversion UID ? how it will work while de-searlize the object ?

serialVersionUID is a unique version identifier used in Java serialization to verify that the sender and receiver of a serialized object have compatible class definitions.
It is declared inside a Serializable class:
private static final long serialVersionUID = 1L;
How it works during deserialization
When an object is deserialized, JVM compares:
serialVersionUID of the serialized object
serialVersionUID of the current class

Cases
If both are same
Deserialization succeeds.
Object is reconstructed normally.

If different
JVM throws InvalidClassException
Because class versions are incompatible.

```
class Student implements Serializable {
    private static final long serialVersionUID = 1L;

    int id;
    String name;
}
```

Object is serialized and saved.
Later, class structure changes (fields added/removed) but UID remains same → object still deserializes successfully.
If UID changes:
private static final long serialVersionUID = 2L;
→ Deserialization fails with InvalidClassException.

16)What is thread ? how many ways we can create the threads ?

A **thread** is the smallest unit of execution inside a process.
It allows a program to perform **multiple tasks concurrently** (multithreading), improving performance and responsiveness.
Example:
In a banking application, one thread can process transactions while another updates account statements simultaneously.

How many ways can we create threads in Java?
There are **2 main ways**:

1. By extending Thread class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Test {
```

```
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

2. By implementing Runnable interface (recommended)

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

Why Runnable is preferred (important interview point)

- Java supports **multiple inheritance via interfaces**, but not via classes.
- If you extend Thread, you cannot extend another class.
- Using Runnable gives better flexibility.

17)can we call strat method twice by using the same thread ?

No, we cannot call the start() method twice on the same thread object. If we try to call it again, Java throws **IllegalThreadStateException**.

Why?

When start() is called:

1. JVM creates a **new thread** and moves it to runnable state.
2. Once the thread finishes execution, it becomes **terminated**.
3. A terminated thread **cannot be restarted**, so calling start() again causes an exception.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();   // First time - works
        t.start();   // Second time - Exception
    }
}
```

18)can we overide the start method in our class ? if we override the start method will achive mutlithreading or not ?
Yes, we **can override** the start() method in our class because it is not final.
If we override start(), will multithreading be achieved?
No (normally).
Multithreading is achieved only when the original Thread.start() method is executed, because it creates a **new separate thread** and then internally calls the run() method.
If we override start() and do not call super.start(), the thread will behave like a **normal method call** and **multithreading will not happen.**

Interview short answer
Yes, we can override the start() method, but if we override it without calling super.start(), multithreading will not be achieved because the JVM creates a new thread only when the original Thread.start() method is executed.

19)what the befinit of extends thread vs implements runnable and calleable interface ?
Benefit of extends Thread vs implements Runnable vs Callable
Java provides three ways to create concurrent tasks, each with different benefits.
1. Extending Thread
Benefit
- Simple and quick way to create a thread.

Limitation
- Java does not support multiple inheritance, so if you extend Thread, you cannot extend another class.
- Not flexible for reusable task design.

Example:
```
class MyThread extends Thread {
    public void run() {
        System.out.println("Task running");
    }
}
```
2. Implementing Runnable (Preferred in most cases)
Benefits
- Allows class to extend another class (supports multiple inheritance via interfaces).
- Better separation between **task (Runnable)** and **thread (Thread).**
- Reusable task logic.

Example:
```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Task running");
    }
}
```

3. Implementing Callable (Advanced / modern approach)
Benefits

- Can **return a result.**
- Can **throw checked exceptions.**
- Used with **ExecutorService** for thread pool execution.

Example:

```
class MyTask implements Callable<Integer> {
    public Integer call() {
        return 10;
    }
}
```

20)what is synchronization ? how may ways we can achieve ?

Synchronization is a mechanism in Java used to **control the access of multiple threads to shared resources** so that **data inconsistency (race condition)** does not occur.

When one thread is executing a synchronized block/method, other threads must wait until the lock is released.

Example problem without synchronization

Two threads updating the same bank account balance may produce incorrect results if executed simultaneously.

How many ways can we achieve synchronization in Java?

There are **3 main ways**:

1. Synchronized Method

```
class Account {
    synchronized void withdraw(int amount) {
        // critical section
    }
}
```

Only one thread can execute this method at a time for the same object.

2. Synchronized Block

```
class Account {
    void withdraw(int amount) {
        synchronized(this) {
            // critical section
        }
    }
}
```

Used when only a portion of code needs synchronization (better performance).

3. Static Synchronization

```
class Demo {
    synchronized static void display() {
        // class-level lock
    }
}
```

Lock is applied on the **class object**, not on individual objects.

21)what is differnce b/w sleep method vs wait method ?
Both are used to pause execution, but they behave differently.

Feature	sleep()	wait()
Package	Thread class	Object class
Purpose	Pause thread for specific time	Wait until notified by another thread
Lock release	❌ Does **not** release lock	✅ Releases the lock
Used in synchronization	Not required	Must be used inside synchronized block
Wake up	Automatically after time	Requires notify() / notifyAll()

sleep()

```
Thread.sleep(2000);  // pauses thread for 2 seconds
```

Thread pauses but **still holds the lock.**

---

wait()

```
synchronized(obj) {
    obj.wait();   // releases lock and waits
}
```

Thread waits until another thread calls:

```
obj.notify();
```

22) why notify () , wait() and notifyAll() methods are belongs to Object class ? why not in Thread class ?

Because these methods are related to **object-level locking (monitor)**, not thread-level operations.
In Java, synchronization works on **objects**, and every object has an intrinsic lock (monitor).
Threads **wait on an object's lock**, and another thread **notifies** waiting threads using the same object.
Therefore, these methods are placed in the **Object class**, so any object can be used for inter-thread communication.

Conceptual understanding

- A thread does not wait by itself.
- A thread waits **on a specific object**:

```
synchronized(obj) {
    obj.wait();      // thread waits on obj's lock
}
```

Another thread:

```
synchronized(obj) {
    obj.notify();    // notifies thread waiting on same object
}
```

Since locking belongs to the **object**, the methods are defined in Object, not in Thread.

23) what is deadlock ? how we can overcome the deadlock ?

Deadlock is a situation where **two or more threads are blocked forever,** each waiting for a resource that is held by another thread.
In simple terms:
Thread-1 waits for Thread-2's resource, and Thread-2 waits for Thread-1's resource → both remain stuck permanently.

Example

- Thread-1 holds **Lock A** and waits for **Lock B**
- Thread-2 holds **Lock B** and waits for **Lock A**
- Neither releases the lock → **deadlock occurs**

How to overcome (prevent) deadlock?

1. Use consistent lock ordering

Always acquire locks in the **same order** in all threads.
Example:

- First lock A
- Then lock B
  (in every thread)
- **2. Avoid nested locks when possible**
- Reduce the use of multiple locks at the same time.
- **3. Use timeout locks (tryLock)**
- If lock is not available within a certain time, release the current lock and retry.
- **4. Minimize synchronized blocks**
- Keep critical sections small to reduce lock holding time.
- **Interview short answer**
- Deadlock is a situation where two or more threads wait indefinitely for each other's resources, causing the program to stop progressing. It can be prevented by maintaining a consistent lock order, avoiding nested locks, minimizing synchronized blocks, and using timeout-based locking mechanisms.
- ---
- **Real-time example**
- In a **bank transfer system**, if two transactions try to lock two accounts in opposite order, both may wait for each other indefinitely, resulting in deadlock.

24) differnce b/w livelock and deadlock?

Difference between Deadlock and Livelock

Both are concurrency problems where threads cannot proceed, but the behavior is different.

Feature	**Deadlock**	**Livelock**
Thread state	Threads are **blocked** and waiting forever	Threads are **active,** but still unable to proceed
Resource holding	Each thread waits for another's resource	Threads continuously retry or respond to each other
CPU usage	Low (threads are stuck)	High (threads keep running)
Progress	No movement	Movement exists but no progress

Deadlock Example

- Thread-1 holds **Lock A** and waits for **Lock B**
- Thread-2 holds **Lock B** and waits for **Lock A**
- Both stop permanently → **deadlock**

Livelock Example

Two people trying to pass in a corridor:

- Person-1 moves left → Person-2 moves right
- Both keep changing sides repeatedly
- They are moving but **still cannot pass → livelock**

Interview short answer

Deadlock occurs when threads are permanently blocked waiting for each other's resources, whereas livelock occurs when threads continuously change states in response to each other but still fail to make progress.

Easy memory trick

- **Deadlock** → threads **stuck**
- **Livelock** → threads **running but not progressing**

25)what is executors in thread?

Executors is a framework in Java (java.util.concurrent) that provides a **high-level way to manage and control threads** using **thread pools**, instead of creating threads manually.

It helps in:

- Reusing threads
- Managing multiple tasks efficiently
- Improving performance and scalability

Why Executors are needed

Creating threads manually:

```
Thread t = new Thread(task);
t.start();
```

If many tasks are created, too many threads will be created → performance issues.

Executors solve this by using a **thread pool** where a fixed number of threads execute multiple tasks.

```
import java.util.concurrent.*;

public class Test {
    public static void main(String[] args) {

        ExecutorService service = Executors.newFixedThreadPool(3);

        service.execute(() -> {
            System.out.println("Task executed by thread");
        });

        service.shutdown();
    }
}
```

Here:

A pool of 3 threads is created

Tasks are assigned to available threads

Interview short answer
Executors is a framework in Java used to manage threads efficiently using thread pools, allowing tasks to be executed without manually creating and managing threads.

Real-time example
In a web server:

- Thousands of requests come
- Executor thread pool processes requests using a limited number of reusable threads

26)how to create singleton object in java ? and it should not break the singleton rule ?
A **Singleton** ensures that **only one instance of a class exists** throughout the application.
The safest and recommended way that **cannot be broken by serialization, reflection, or cloning** is using **Enum Singleton.**

---

1. Recommended (Best) – Enum Singleton

```
public enum Singleton {
    INSTANCE;

    public void show() {
        System.out.println("Singleton object");
    }
}
Singleton obj = Singleton.INSTANCE;
obj.show();
```

Why best?

Prevents multiple objects during serialization
Safe against reflection
Thread-safe by default

2. Classic Singleton (with protections)

```
public class Singleton implements Serializable, Cloneable {

    private static final Singleton instance = new Singleton();

    private Singleton() {
        // prevent reflection
        if (instance != null) {
            throw new RuntimeException("Use getInstance()");
        }
    }
```

```
    public static Singleton getInstance() {
        return instance;
    }

    // prevent cloning
    protected Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }

    // prevent serialization creating new object
    protected Object readResolve() {
        return instance;
    }
}
```

A Singleton object can be created by making the constructor private and providing a static method to return the single instance. To ensure it is not broken by serialization, cloning, or reflection, the best approach is to implement the Singleton using an Enum.

Real-time example

- Logger instance in an application
- Configuration manager
- Database connection manager (single shared instance)

27)what is differnce between StringBuffer vs StringBuilder?
Both are used to create **mutable (modifiable) strings**, but the main difference is **thread safety**.

Feature	**StringBuffer**	**StringBuilder**
Thread safety	Thread-safe (synchronized)	Not thread-safe
Performance	Slower (due to synchronization)	Faster
Use case	Multi-threaded applications	Single-threaded applications
Introduced	Java 1.0	Java 1.5

```
StringBuffer sb1 = new StringBuffer("Hello");
sb1.append(" World");

StringBuilder sb2 = new StringBuilder("Hello");
sb2.append(" World");
```

Both modify the same object instead of creating new strings.

Interview short answer
StringBuffer is synchronized and thread-safe but slower, whereas StringBuilder is not synchronized, faster, and preferred in single-threaded environments.

Real-time understanding

- **StringBuffer**: Used when multiple threads modify the same string (e.g., shared logging buffer).

- **StringBuilder:** Used in most applications where only one thread modifies the string (e.g., building SQL queries, loops).

28)Differnce between List Vs Set?

Both List and Set are interfaces of the **Collection framework**, but they store elements differently.

Feature	**List**	**Set**
Duplicate elements	Allowed	Not allowed
Order of elements	Maintains insertion order	Does not guarantee order (depends on implementation)
Index-based access	Yes	No
Example implementations	ArrayList, LinkedList, Vector	HashSet, LinkedHashSet, TreeSet

---

Example

List (duplicates allowed)

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("A");
System.out.println(list);   // [A, A]
```

Set (duplicates not allowed)

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("A");
System.out.println(set);   // [A]
```

Interview short answer

List allows duplicate elements and maintains insertion order with index-based access, whereas Set does not allow duplicate elements and generally does not provide index-based access.

Real-time example

- **List:** storing user transaction history (duplicates allowed)
- **Set:** storing unique email IDs or usernames (duplicates not allowed)

29)Differnce between ArrayList Vs Array?

Difference between Array and ArrayList

Feature	Array	ArrayList
Size	Fixed size (cannot change after creation)	Dynamic size (can grow/shrink automatically)
Data types	Can store primitives and objects	Stores only objects (wrapper classes for primitives)
Package	Part of Java language	Part of java.util package
Performance	Faster (no resizing overhead)	Slightly slower (dynamic resizing)
Methods support	Limited (no built-in methods like add/remove)	Many built-in methods (add(), remove(), size())

Example
Array

```
int[] arr = new int[3];
arr[0] = 10;
Size is fixed to 3.
```

ArrayList

```
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
Size grows automatically.
```

Interview short answer
An array has a fixed size and can store both primitive and object types, whereas ArrayList is a resizable collection class that stores only objects and provides built-in methods for dynamic operations like add and remove.

Real-time example

- **Array:** storing fixed number of days in a week
- **ArrayList:** storing user records where the number of users can change dynamically

30)Differnce between ArrayList Vs LinkedList?

Difference between ArrayList and LinkedList

Both are implementations of the **List interface**, but their internal data structures are different.

Feature	ArrayList	LinkedList
Internal structure	Dynamic array	Doubly linked list
Access (get) performance	Fast (O(1))	Slower (O(n))
Insertion/Deletion (middle)	Slower (shifting required)	Faster (no shifting)
Memory usage	Less memory	More memory (extra node pointers)
Best use case	Frequent searching/accessing	Frequent insertion/deletion

---

Example

```
List<Integer> arr = new ArrayList<>();
arr.add(10);

List<Integer> list = new LinkedList<>();
list.add(10);
```

Interview short answer

ArrayList is based on a dynamic array and provides faster random access, whereas LinkedList is based on a doubly linked list and provides faster insertion and deletion operations.

Real-time example

- **ArrayList:** storing student records where frequent reading is required
- **LinkedList:** task queue where items are frequently added and removed

31)how HashSet internall works ? how hashmap internall works

How HashSet works internally

HashSet internally uses a **HashMap** to store elements.

- When you add an element to HashSet, it internally calls:

```
map.put(element, PRESENT);
```

- The element is stored as **key** in HashMap.
- A dummy constant object (PRESENT) is used as the value.
- Duplicate elements are not allowed because **HashMap keys must be unique.**

Steps internally

1. hashCode() of element is calculated.
2. Bucket index is determined.
3. If bucket is empty → element stored.
4. If bucket already has elements → equals() is used to check duplicates.
5. If equal → not inserted.

How HashMap works internally

HashMap stores data as **key–value pairs** using an **array of buckets.**

Steps internally

1. hashCode() of key is calculated.
2. Hash is converted into **bucket index.**
3. Entry stored in that bucket.
4. If collision occurs (multiple keys in same bucket):
    - Java 7 → Linked List
    - Java 8+ → LinkedList converted to **Balanced Tree (Red-Black Tree)** when entries grow

During retrieval:

1. Calculate hashCode of key
2. Go to bucket
3. Use equals() to find exact key
4. Return value

Interview short answer

HashSet internally uses HashMap to store elements as keys with a dummy value, ensuring uniqueness. HashMap works by calculating the hashCode of the key, determining a bucket index, and storing key-value pairs; collisions are handled using linked lists or trees.

Very important interview tip (frequently asked)
HashSet uses

- hashCode() → find bucket
- equals() → check duplicate

HashMap uses

- hashCode() → bucket selection
- equals() → key comparison

32)differnce between HashSet vs LinkedHashSet vs TreeSet?

Difference between HashSet, LinkedHashSet, and TreeSet
All three implement the **Set interface** (no duplicate elements), but they differ in ordering and performance.

Feature	**HashSet**	**LinkedHashSet**	**TreeSet**
Ordering	No ordering (random)	Maintains insertion order	Sorted order (ascending by default)
Internal structure	Hash table (HashMap)	Hash table + Linked list	Red-Black Tree
Performance	Fastest	Slightly slower than HashSet	Slower (due to sorting)
Null values	Allows one null	Allows one null	Does not allow null
Use case	When order is not important	When insertion order is required	When sorted data is required

---

Example

```
Set<Integer> hs = new HashSet<>();
Set<Integer> lhs = new LinkedHashSet<>();
Set<Integer> ts = new TreeSet<>();

hs.add(3); hs.add(1); hs.add(2);
lhs.add(3); lhs.add(1); lhs.add(2);
ts.add(3); ts.add(1); ts.add(2);

System.out.println(hs);   // random order
System.out.println(lhs);  // 3,1,2 (insertion order)
System.out.println(ts);   // 1,2,3 (sorted)
```

Interview short answer
HashSet stores unique elements without maintaining order, LinkedHashSet maintains insertion order, and TreeSet stores unique elements in sorted order using a tree structure.

Easy memory trick

- **HashSet** → No order
- **LinkedHashSet** → Insertion order
- **TreeSet** → Sorted order

33)hashmap vs Hashtable

Difference between HashMap and Hashtable

Feature	**HashMap**	**Hashtable**
Thread safety	Not synchronized (not thread-safe)	Synchronized (thread-safe)
Performance	Faster	Slower (due to synchronization)
Null keys/values	Allows **one null key** and multiple null values	Does **not allow null** key or value
Introduced	Java 1.2 (Collections Framework)	Legacy class (Java 1.0)
Iterator support	Uses Iterator (fail-fast)	Uses Enumerator (not fail-fast by default)

Example

```
HashMap<Integer, String> map = new HashMap<>();
map.put(null, "A");   // allowed

Hashtable<Integer, String> table = new Hashtable<>();
table.put(null, "A"); // NullPointerException
```

Interview short answer

HashMap is non-synchronized, faster, and allows one null key and multiple null values, whereas Hashtable is synchronized, slower, and does not allow null keys or values.

Important interview tip

Modern applications prefer **HashMap.**
If thread safety is needed, use:
Collections.synchronizedMap(new HashMap<>());
or **ConcurrentHashMap** (recommended).

34)how we can iterate the hashmap?

There are **several ways** to iterate a HashMap.
Assume:

```
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "A");
map.put(2, "B");
map.put(3, "C");
```

---

1. Using entrySet() (Most recommended)

```
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " " + entry.getValue());
}
```

Efficient because key and value are accessed together.

---

2. Using keySet()

```
for (Integer key : map.keySet()) {
    System.out.println(key + " " + map.get(key));
}
```

Here key is retrieved first, then value using get().

3. Using forEach() (Java 8+)

```
map.forEach((key, value) -> {
    System.out.println(key + " " + value);
});
```

---

4. Using Iterator

```
Iterator<Map.Entry<Integer, String>> itr = map.entrySet().iterator();

while (itr.hasNext()) {
    Map.Entry<Integer, String> entry = itr.next();
    System.out.println(entry.getKey() + " " + entry.getValue());
}
```

Interview short answer

A HashMap can be iterated using entrySet(), keySet(), forEach() (Java 8), or an Iterator, with entrySet() being the most efficient method because it provides direct access to both key and value.

35)what is concurrentHashMap ? why we need to use ?

What is ConcurrentHashMap?

ConcurrentHashMap is a **thread-safe implementation of Map** provided in java.util.concurrent package that allows **multiple threads to read and write simultaneously** without locking the entire map.

Why do we need ConcurrentHashMap?

- HashMap → Not thread-safe
- Hashtable → Thread-safe but locks the **entire map**, causing poor performance
- ConcurrentHashMap → Thread-safe with **better performance**, because it uses **segment-level / bucket-level locking** allowing multiple threads to operate concurrently.

Example

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
map.put(1, "A");
map.put(2, "B");
```

Multiple threads can safely update this map at the same time.

Key features (important interview points)

- Thread-safe without full map locking
- Better performance than Hashtable
- Does not allow **null key or null value**
- Supports concurrent read and write operations

Interview short answer

ConcurrentHashMap is a thread-safe Map implementation that allows concurrent read and write operations with high performance. It is used instead of Hashtable when multiple threads need to access and modify a map simultaneously.

Real-time example
In a **web application cache**, multiple user requests update and read cached data simultaneously, so ConcurrentHashMap is used to ensure thread safety without performance bottlenecks.

36)TreeMap vs TreeSet
Difference between TreeMap and TreeSet
Both use a **Red-Black Tree** internally and store elements in **sorted order**, but they serve different purposes.

Feature	**TreeMap**	**TreeSet**
Type	Map (key-value pairs)	Set (only values)
Duplicates	Keys cannot be duplicate (values can)	No duplicate elements
Ordering	Sorted based on keys	Sorted based on elements
Null values	Does not allow null key (values allowed)	Does not allow null
Usage	When key-value sorted data is needed	When unique sorted elements are needed

---

Example
TreeMap
```
TreeMap<Integer, String> map = new TreeMap<>();
map.put(2, "B");
map.put(1, "A");
System.out.println(map);   // sorted by keys
```

TreeSet
```
TreeSet<Integer> set = new TreeSet<>();
set.add(2);
set.add(1);
System.out.println(set);   // sorted elements
```

Interview short answer
TreeMap stores key-value pairs in sorted order based on keys, whereas TreeSet stores only unique elements in sorted order.

Easy memory trick

- **TreeMap** → Sorted **key-value**
- **TreeSet** → Sorted **values only**

37)List,Set,Map those are non-sycronized , if we wan to make them as sychnronized what we have to do ?
Making List, Set, and Map synchronized
List, Set, and Map implementations like **ArrayList**, **HashSet**, **HashMap** are **not synchronized** by default.
To make them synchronized, we use the **Collections utility class.**

---

1. Synchronized List
```
List<String> list = Collections.synchronizedList(new ArrayList<>());
```

2. Synchronized Set

```
Set<String> set = Collections.synchronizedSet(new HashSet<>());
```

---

3. Synchronized Map

```
Map<Integer, String> map = Collections.synchronizedMap(new HashMap<>());
```

These wrappers ensure that only **one thread at a time** can access the collection.

Interview short answer

Non-synchronized collections like List, Set, and Map can be made synchronized by using the Collections.synchronizedList(), Collections.synchronizedSet(), and Collections.synchronizedMap() methods.

Important interview tip

For **high-performance multithreaded applications**, instead of synchronized wrappers, Java provides concurrent collections such as:

- CopyOnWriteArrayList
- ConcurrentHashMap

38)java8 features ?

Important Java 8 Features

Java 8 introduced several powerful features mainly for **functional programming, performance, and readability.**

---

1. Lambda Expressions

Allows writing anonymous functions in a concise way.

```
Runnable r = () -> System.out.println("Running");
```

---

2. Functional Interfaces

Interfaces with **only one abstract method** (e.g., Runnable, Comparator).

```
@FunctionalInterface
interface MyInterface {
    void show();
}
```

---

3. Stream API

Used for **processing collections in a functional style** (filter, map, reduce).

```
list.stream()
    .filter(x -> x > 10)
    .forEach(System.out::println);
```

---

4. Default and Static Methods in Interfaces

Interfaces can have method implementations.

```
interface Demo {
    default void show() {
        System.out.println("Default method");
    }
}
```

---

5. Method References

Shorter way of writing lambda expressions.

```
list.forEach(System.out::println);
```

---

6. Optional Class
Helps avoid NullPointerException.
```
Optional<String> name = Optional.ofNullable(null);
```

---

7. Date and Time API (java.time package)
Improved date/time handling.
```
LocalDate today = LocalDate.now();
```

Interview short answer
Java 8 introduced major features such as Lambda expressions, Functional Interfaces, Stream API, Default and Static methods in interfaces, Method references, Optional class, and the new Date and Time API.

39)stream vs collection

Difference between Stream and Collection

Feature	Collection	Stream
Purpose	Stores data	Processes data
Storage	Holds elements in memory	Does not store elements
Modification	Can add/remove elements	Cannot modify original data
Traversal	Can be traversed multiple times	Consumed once (cannot reuse)
Operations	Basic operations (add, remove, get)	Functional operations (filter, map, reduce)
Introduced	Earlier Java versions	Java 8

Example
Collection
```
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
```
Stores elements.

Stream
```
list.stream()
    .filter(x -> x > 10)
    .forEach(System.out::println);
```
Processes elements without storing new data.

Interview short answer
A Collection is used to store and manage data in memory, whereas a Stream is used to process data from collections in a functional and pipeline manner without storing it.

Easy memory trick

- **Collection** → Storage
- **Stream** → Processing

40)what optional class and usage ?
Optional is a container object introduced in **Java 8** that may **contain a value or may be empty**.
It is mainly used to **avoid NullPointerException** by handling null values safely.
Package:
java.util.Optional

Why Optional is used?
Instead of returning null, methods can return an **Optional object**, forcing the developer to check whether the value is present or not.

Example

```
Optional<String> name = Optional.ofNullable(null);

if (name.isPresent()) {
    System.out.println(name.get());
} else {
    System.out.println("Value not present");
}
```

Common methods (important interview points)

- of() → creates Optional with non-null value
- ofNullable() → allows null value
- isPresent() → checks value exists
- get() → retrieves value
- orElse() → default value if null

Example:

```
String result = Optional.ofNullable(name)
                        .orElse("Default Name");
```

Interview short answer
Optional is a container class introduced in Java 8 that represents the presence or absence of a value and is mainly used to avoid NullPointerException by handling null values safely.

Real-time example
When fetching a user from database:

- If user exists → return Optional<User>
- If not → Optional.empty() instead of returning null

41)why default methods introduced in java 8?

Default methods were introduced in **interfaces** to allow adding new methods to existing interfaces **without breaking the classes that already implement them** (backward compatibility).
Problem before Java 8

- Interfaces could contain only **abstract methods**.
- If a new method was added to an interface, **all implementing classes** had to implement that new method.
- This caused **compilation errors** in large existing codebases.

Solution: Default methods

Java 8 introduced **default methods** (methods with implementation inside interface using default keyword).

This allows:

- Adding new functionality to interfaces without forcing all implementing classes to change.
- Providing a **default implementation** that classes may override if needed.

Example

```
interface Vehicle {

    void start();

    default void fuelType() {
        System.out.println("Petrol");
    }
}
```

Implementing class:

```
class Car implements Vehicle {

    public void start() {
        System.out.println("Car started");
    }
}
```

Car does not need to implement fuelType() because the interface already provides a default implementation.

Key reasons

- Backward compatibility
- Interface evolution without breaking existing code
- Support for functional programming features (like streams, collections enhancements)

42)why Static methods introduced in java 8?

Why were static methods introduced in Java 8 interfaces?

Static methods were introduced in interfaces to allow **utility/helper methods related to the interface** to be defined **inside the interface itself**, instead of placing them in separate utility classes.

Reasons

1. To keep **related helper methods together** with the interface.
2. To avoid creating separate utility classes (like Collections, Arrays).
3. Provide **method implementation that belongs to the interface but should not be overridden by implementing classes**.

Example

```
interface MyInterface {

    static void show() {
        System.out.println("Static method in interface");
    }
}
```

Calling:

```
MyInterface.show();   // called using interface name
```

Implementing classes **cannot override** this static method.

Short interview answer
Static methods were introduced in Java 8 interfaces to allow defining **utility methods inside interfaces**, improving code organization and preventing unnecessary utility classes.

43)what is method refernce in java8?

Method Reference in Java 8
A **method reference** is a shorthand notation of a lambda expression used to call an existing method directly.
Instead of writing a lambda expression, we can refer to the method by name using ::.

Syntax
ClassName::methodName

Example (Lambda vs Method Reference)
Lambda
```
list.forEach(x -> System.out.println(x));
```
Method Reference
```
list.forEach(System.out::println);
```

Types of Method References
1. **Reference to static method**
ClassName::staticMethod
2. **Reference to instance method of particular object**
object::instanceMethod
3. **Reference to instance method of arbitrary object**
ClassName::instanceMethod
4. **Reference to constructor**
ClassName::new

Short Interview Answer
A **method reference** in Java 8 is a compact way of writing lambda expressions by directly referring to an existing method using the :: operator.

44)what are all the intermediate and terminal operation in stream ?
In **Java Streams**, operations are divided into **Intermediate operations** and **Terminal operations**.

Intermediate vs Terminal Operations
1. Intermediate Operations
- These **transform the stream**
- They are **lazy** (not executed until terminal operation is called)
- They **return another Stream**

Common Intermediate Operations

- filter()
- map()
- flatMap()
- distinct()
- sorted()
- limit()
- skip()
- peek()

Example

```
list.stream()
    .filter(x -> x > 10)
    .map(x -> x * 2)
(No execution yet)
```

---

2. Terminal Operations

- They **produce result**
- They **trigger execution** of stream pipeline
- After terminal operation, stream **cannot be reused**

Common Terminal Operations

- forEach()
- collect()
- count()
- reduce()
- findFirst()
- findAny()
- anyMatch()
- allMatch()
- noneMatch()
- min()
- max()

Example

```
list.stream()
    .filter(x -> x > 10)
    .collect(Collectors.toList());
```

Simple Interview Line

- **Intermediate operations** return another stream and are lazy.
- **Terminal operations** produce final result and start the execution of the stream.

Quick Memory Trick (very useful in interviews)

- **Intermediate → returns Stream**
- **Terminal → returns Result**

45) java 8 map vs flatMap ?

Java 8 map() vs flatMap()

Both are intermediate stream operations used for transformation, but they behave differently.

---

map()

- **Used to transform each element into another element**
- **Returns one output element for each input element**
- **Result: Stream<T> → Stream<R>**

Example

Convert list of names to uppercase:

```
List<String> names = Arrays.asList("Anu","Ravi","John");

names.stream()
     .map(name -> name.toUpperCase())
     .forEach(System.out::println);
```

Output:

```
ANU
RAVI
JOHN
```

Here each element becomes one transformed element.

---

flatMap()

- **Used when each element produces multiple elements (nested collections)**
- **It flattens the result into a single stream**
- **Result: Stream<Stream<T>> → Stream<T>**

Example

List of lists:

```
List<List<String>> list = Arrays.asList(
        Arrays.asList("A","B"),
        Arrays.asList("C","D")
);

list.stream()
    .flatMap(x -> x.stream())
    .forEach(System.out::println);
```

Output:

```
A
B
C
D
```

Simple Interview Difference (Best Answer)

- **map() → one-to-one transformation**
- **flatMap() → one-to-many transformation and flattens nested streams**

Very Short Memory Trick

- **map() → Transform**
- **flatMap() → Transform + Flatten**

46)what is lambda experession ? how we can execute give me one example ?

Lambda Expression (Java 8)

A **lambda expression** is a **short way to write anonymous functions** (functions without name) mainly used to implement **functional interfaces** (interfaces having only one abstract method).

It reduces boilerplate code and makes code more readable.
Syntax
(parameters) -> { body }

Example (Without Lambda)
```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread running...");
    }
};

new Thread(r).start();
```

Example (Using Lambda)
```
Runnable r = () -> {
    System.out.println("Thread running...");
};

new Thread(r).start();
```
Here lambda replaces the anonymous class.

Another Simple Example (Functional Interface)
```
interface Add {
    int sum(int a, int b);
}

public class Test {
    public static void main(String[] args) {
        Add a = (x, y) -> x + y;
        System.out.println(a.sum(5, 3));
    }
}
```
Output:
8

Interview-ready Short Answer
A **lambda expression** is a concise way to implement a functional interface using (parameters) -> expression, introduced in Java 8 to support functional programming and stream operations.

47)how we can overcome multiple inheritance problem in java 8 interface defult methods ?
In Java 8, interfaces can have **default methods**.
If a class implements **two interfaces having the same default method**, ambiguity occurs (multiple inheritance problem).
Java resolves this by forcing the **implementing class to override the method explicitly**.

Example (Multiple inheritance conflict)
```
interface A {
    default void show() {
        System.out.println("A show");
```

```
    }
}

interface B {
    default void show() {
        System.out.println("B show");
    }
}

class Test implements A, B {
    public void show() {        // must override
        System.out.println("My own implementation");
    }
}
```

Here the class **must override show()**, otherwise compilation error occurs.

If we want to call specific interface method

We can use:

```
A.super.show();
B.super.show();
```

Example:

```
class Test implements A, B {
    public void show() {
        A.super.show();   // calling A method
        B.super.show();   // calling B method
    }
}
```

Interview-ready Answer

When multiple interfaces contain the same default method, Java resolves the multiple inheritance ambiguity by requiring the implementing class to **override the method explicitly**, and the class can call a specific interface method using InterfaceName.super.method().

48) In your project where you have implemented the java 8 featuers?

In my project, Java 8 features were mainly used to **improve code readability, reduce boilerplate code, and process collections efficiently**.

1. Streams & Lambda Expressions

Used for filtering, sorting, and transforming data from database results or collections.

Example:

```
List<User> activeUsers =
    users.stream()
         .filter(u -> u.isActive())
         .collect(Collectors.toList());
```

2. Optional

Used to avoid NullPointerException while fetching optional data.

```
Optional<User> user = userRepository.findById(id);
user.ifPresent(u -> System.out.println(u.getName()));
```

3. Method References

Used for cleaner lambda expressions.

```
users.forEach(System.out::println);
```

4. Default Methods in Interfaces

Used to add common reusable logic in service interfaces without breaking existing implementations.

Short 3-line Quick Interview Version

In my project, Java 8 features like **Streams, Lambda expressions, Optional, and Method References** were used for collection processing, null handling, and writing cleaner functional-style code, especially in service and repository layer data transformations.

49)what is functional interface ? how we can create the functional interface ? can fucctional inteface can extend another functional interface ?

A **functional interface** is an interface that contains **exactly one abstract method** (SAM – Single Abstract Method).

It can have **multiple default and static methods**, but only **one abstract method.**

Examples: Runnable, Callable, Comparator

How to create a Functional Interface?

We create it using @FunctionalInterface annotation (optional but recommended).

```
@FunctionalInterface
interface MyFunctional {
    void show();   // only one abstract method
}
```

Using lambda:

```
MyFunctional obj = () -> System.out.println("Hello");
obj.show();
```

Can a Functional Interface extend another Functional Interface?

Yes, it can extend another functional interface **only if the total number of abstract methods remains ONE.**

Example (valid):

```
@FunctionalInterface
interface A {
    void show();
}

@FunctionalInterface
interface B extends A {
    // no new abstract method
}
```

Invalid example:

```
interface B extends A {
    void test();   // now 2 abstract methods -> not functional interface
}
```

Interview-ready Short Answer

A functional interface is an interface with **only one abstract method,** used mainly for lambda expressions. It can be created using

`@FunctionalInterface annotation, and it can extend another functional interface` **`only if it still contains exactly one abstract method.`**

`50)Predicate vs consumer ? can you tell predefine interfaces in java 8 ?`
Both are **predefined functional interfaces** in java.util.function package, but they serve different purposes.

Feature	**Predicate<T>**	**Consumer<T>**
Purpose	Represents a **boolean-valued function** (tests a condition)	Represents an **operation on an object** (performs an action)
Abstract method	boolean test(T t)	void accept(T t)
Return type	boolean	void
Example use	Filtering collections	Printing, modifying, or performing actions

---

Example

Predicate

```
Predicate<Integer> isEven = x -> x % 2 == 0;

System.out.println(isEven.test(10)); // true
```

Consumer

```
Consumer<String> print = s -> System.out.println(s);

print.accept("Hello"); // prints Hello
```

Predefined Functional Interfaces in Java 8

1. **Predicate<T>** – test a condition → boolean test(T t)
2. **Consumer<T>** – performs an action → void accept(T t)
3. **Supplier<T>** – provides a result → T get()
4. **Function<T,R>** – transforms → R apply(T t)
5. **UnaryOperator<T>** – special case of Function (T → T)
6. **BinaryOperator<T>** – special case of BiFunction (T,T → T)

7. **BiFunction<T,U,R>** – takes two inputs, returns a result
8. **BiConsumer<T,U>** – takes two inputs, returns nothing

Short Interview Answer

- **Predicate**: tests a condition and returns boolean.
- **Consumer**: performs an action on an object and returns nothing.
- **Common Java 8 functional interfaces**: Predicate, Consumer, Supplier, Function, UnaryOperator, BinaryOperator, BiFunction, BiConsumer.