Advanced Java Topics

S.No	Topic Name
1	Exception in Java
2	Exception Handling
3	Try with Multiple Catch Block
4	Exception Hierarchy
5	Throw Keyword
6	Custom Exception
7	Ducking Exception (Throws Keyword)
8	Try With Resources
9	Threads (Basics)
10	Multithreading
11	Thread Priority & Sleep Method
12	Runnable vs Thread
13	Race Condition & Synchronization
14	Thread States & Lifecycle

1. What is Exception in Java

1. Types of Errors in Java

In Java, three types of errors can occur:

1️⃣ Compile-Time Errors

These errors occur **before the program runs**, when you try to compile the code.

- They are mostly **syntax mistakes**.
- Example: forgetting a semicolon, misspelling a variable, wrong method name.
- If compile-time errors exist → **code will not run** because Java won't create the .class file.

Interview line:
"Compile-time errors are detected by the compiler and occur due to syntax issues."

2️⃣ Logical Errors

These happen **after the program runs**, but the **output is wrong**.

- No compiler error, no runtime crash — just wrong results.
- Caused by **wrong program logic** written by the coder.

Example:
Suppose you want to add two numbers but you multiply them.

Interview line:
"Logical errors do not crash the program, but they produce incorrect output."

3️⃣ Runtime Errors (Exceptions)

These occur **while the program is running**.

- They happen due to unexpected situations like:
  - Dividing by zero
  - File not found
  - Accessing invalid array index
- If not handled → program terminates.

Interview line:
"Runtime errors are called Exceptions, and Java provides exception handling to prevent program crash."

What is an Exception?

An **Exception** is an *unexpected event* that stops normal program execution.
Java gives us 5 keywords to handle them:

try
catch
finally
throw
throws

Interview line:
"Exceptions allow us to handle runtime problems gracefully instead of letting the program crash."

Exception Types

Java divides exceptions into **Checked** and **Unchecked**.

A. Checked Exceptions

These are checked **at compile time**.

- Compiler forces you to handle them using:
  - try-catch, OR
  - throws
- Mostly related to:
  - File handling
  - Input/output
  - Databases
  - Networks

Examples:
IOException, SQLException, FileNotFoundException

Interview line:
"Checked exceptions are verified at compile time and must be handled or declared, otherwise the code won't compile."

B. Unchecked Exceptions

These are checked **only at runtime**.

- Compiler **does NOT** force you to handle them.
- Mostly programmer mistakes.

Examples:
NullPointerException,
ArithmeticException,
ArrayIndexOutOfBoundsException

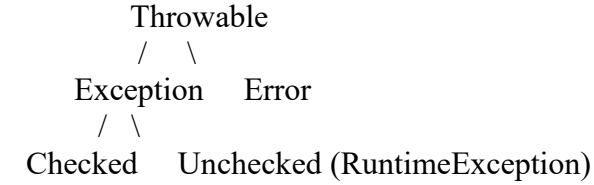
Interview line:
"Unchecked exceptions are not checked at compile time; they occur because of programming errors."

Most Important Interview Difference

Feature	Checked Exception	Unchecked Exception
Checked at	Compile time	Runtime
Need to handle?	Yes	No
Reason	External factors (I/O, DB, network)	Programming errors
Examples	IOException	NullPointerException

Interview line:
"Both checked and unchecked exceptions occur at runtime, but only checked exceptions are verified by the compiler."

Exception Hierarchy



Key points

- **Exception** → Recoverable
- **Error** → NOT recoverable (memory issues, JVM crash)

Interview line:
"Errors cannot be handled, but exceptions can be handled using try–catch."

If you have to explain in 30 seconds (Interview Quick Answer)**

"Java errors are of three types: compile-time errors (syntax issues), logical errors (wrong output), and runtime errors called exceptions. Exceptions are events that disrupt normal execution. Java has two types of exceptions: checked exceptions, which are checked at compile time and must be handled (like IOException), and unchecked exceptions, which are runtime issues caused by programming mistakes (like NullPointerException). Java uses try-catch to handle exceptions."

---

2. Exception Handling

⭐ What is Exception Handling?

Exception handling in Java is a mechanism that allows developers to **detect, handle, and recover from runtime errors** without terminating the entire program.

It helps maintain **normal program flow** even when unexpected situations occur (e.g., dividing by zero, file not found).

⭐ Why Exception Handling?

Without exception handling:

- The exception is thrown to the **JVM**
- JVM **abnormally terminates** the program
- Remaining code **does NOT execute**

With exception handling:

- The exception is **caught and handled**
- JVM does **not terminate the program**
- Remaining statements **execute normally**

⭐ try–catch Block (Basic Syntax)

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle specific exception type
} catch (ExceptionType2 e2) {
    // Handle another exception type
} finally {
    // Optional: runs whether exception occurs or not
}
```

⭐ Example: Handling Exception Using try–catch

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

        int i = 4; // normal statement

        try {
            int a = 10 / 0; // critical statement

            /*
             If you do not handle this exception,
             JVM will terminate the program abnormally and
             the remaining statements will NOT execute.
            */

            /*
             If you handle the exception,
             it will NOT be thrown to the JVM,
             and the program continues normally.
            */

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

        System.out.println("We are coming out of the try-catch block successfully");
    }
}
```

✓ Explanation

- The line int a = 10/0; causes an **ArithmeticException**.
- The catch block handles the exception.
- The last print statement runs successfully because the program does NOT crash.

⭐ Types of Statements in Exception Handling

Java statements can be divided into two types:

1️⃣ Normal Statements

These statements:

- Do NOT need special handling
- Execute in normal sequence
- Do NOT cause exceptions (usually)

Examples

- Variable declarations
- Assignments
- Print statements

Example:

int x = 10;

System.out.println("Hello");

2️⃣ Critical Statements

These statements:

- MAY cause an exception at runtime
- Need **special handling**
- Should be placed inside a try block

Examples

- Division operations
- File handling
- Array operations
- Database/Network calls

Critical statements can be handled using:

- try-catch
- throw
- finally

Example:

int result = 10 / 0; // may throw ArithmeticException

⭐ **Summary (For Placements)**

- try block contains **risk/critical** code.
- catch block **handles** the exception.
- finally block runs **always**, even if exception occurs.
- Without handling, JVM **abnormally terminates** the program.
- Normal statements → safe statements
- Critical statements → may cause exceptions and must be handled.

---

3. Try with Multiple Catch block

⭐ **Why do we use multiple catch blocks?**

Sometimes, we write several lines of code inside a try block, and we **don't know which line may throw an exception**, BUT we do know the possible **types of exceptions** that might occur.

To handle different exception types separately, Java allows us to use **multiple catch blocks**.

⭐ **Example: Handling Multiple Exceptions**

```
int num = 4;
int arr[] = {3, 4, 5};

try {
    int result = 40 / num;
    // This may throw ArithmeticException if num = 0

    System.out.println(arr[result]);
    // This may throw ArrayIndexOutOfBoundsException
    // because valid index range is 0 to 2
}
catch (ArithmeticException ae) {
    System.out.println(ae);
}
catch (ArrayIndexOutOfBoundsException aio) {
    System.out.println(aio);
}
```

✓ Explanation

- The try block contains **critical statements** that may throw exceptions.
- If division fails → ArithmeticException block is executed.
- If invalid array index is accessed → ArrayIndexOutOfBoundsException block executes.
- Only **one** catch block executes per exception.

⭐ Parent and Child Exceptions – Important Interview Concept

Java has an **exception hierarchy**, where:

- Exception is the **parent**
- Classes like ArithmeticException, ArrayIndexOutOfBoundsException are **child exceptions**

When using multiple catch blocks:

❗ Always catch child exceptions BEFORE the parent exception.

If you catch the parent (Exception) first:

- It will catch *all* child exceptions
- Child catch blocks will **never execute**
- Compiler will report an error

❌ Wrong Order (Will Cause Compile-Time Error)

java

```
int a = 10;
int arr[] = {3, 4, 5};

try {
    int b = 3 / a;
    System.out.println(arr[b]);
}
catch (Exception e) {
    System.out.println("parent class of every exception");
}
catch (ArithmeticException e) {
    // ❌ Compile-time error
}
```

✔ Compiler Error:

go

```
error: exception ArithmeticException has alrea ↓ een caught
```

✅ Correct Order (Child First, Parent Last)

java

```
int a = 10;
int arr[] = {3, 4, 5};

try {
    int b = 3 / a;
    System.out.println(arr[b]);
}
catch (ArithmeticException e) {
    // Child exception
}
catch (Exception e) {
    // Parent exception
    System.out.println("parent class of every exception");
}
```

✓ **Explanation:**

- Child exceptions must be caught first.
- Parent exception is kept **last** as a fallback/default handler.
- This structure avoids compile-time errors and handles exceptions correctly.

⭐ **Key Notes (Placement-Important)**

- Multiple catch blocks help handle **different exceptions separately**.
- Catch blocks run **top to bottom** → first match wins.
- **Child exceptions must be caught before parent exceptions**.
- If parent comes first → compiler error: *"exception already caught"*
- Only **one catch block** executes per thrown exception.

---

4. Exception Hierarchy

In Java, **exceptions are objects** that represent abnormal or exceptional conditions that occur during program execution.
Java provides a structured **exception class hierarchy** to categorize and handle different types of runtime issues.

⭐ **Important Points About Exception Hierarchy**

✓ **1. Every class in Java extends Object**

This means the root of all classes (including exceptions) is the **Object class**.

✓ **2. Top-most class in exception system → Throwable**

- Throwable is the **superclass** of all errors and exceptions.
- It directly extends the Object class.

✓ **3. Throwable has two main subclasses:**

1. **Error**
2. **Exception**

⭐ Error

- Represents **serious problems** that applications should *not* try to handle.
- Usually caused by **JVM-level issues**.
- **Program cannot recover.**
- Examples:
    - OutOfMemoryError
    - StackOverflowError
    - VirtualMachineError

Interview line:
➡️ *Errors are not meant to be caught or handled by the application.*

⭐ Exception

- Represents **conditions a program may want to catch and handle**.
- Exceptions are **recoverable**.
- Examples:
    - IOException
    - SQLException
    - ArithmeticException

⭐ Checked vs Unchecked Exceptions

✓ Checked Exceptions

- **Checked at compile-time.**
- Programmer must:
    - Handle with try-catch, OR
    - Declare using throws
- If not handled → compile-time error.
- Examples:
    - IOException
    - SQLException
    - ClassNotFoundException

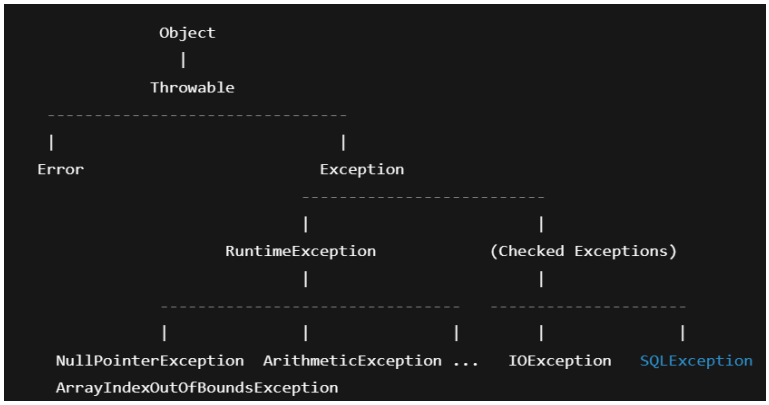
When used?
When dealing with **external resources** like files, DB, networks.

✓ Unchecked Exceptions (Runtime Exceptions)

- **Checked at runtime** (NOT at compile-time).
- Compiler does NOT force handling.
- Usually caused by **logic/programming mistakes**.
- Examples:
    - NullPointerException
    - ArithmeticException
    - ArrayIndexOutOfBoundsException
    - ClassCastException

When used?
Logic errors in code.

⭐ Java Exception Hierarchy Diagram



⭐ Quick Interview Summary

- All exception classes extend **Throwable**.
- Error → Serious, unrecoverable issues.
- Exception → Recoverable issues, can be handled.
- Checked exceptions → compiler checks them, must be handled.
- Unchecked exceptions → runtime issues, typically coding mistakes.
- RuntimeException and its subclasses are **unchecked**.

5. Throw Keyword

Java throw Keyword — Placement Notes

The **throw keyword** in Java is used to **explicitly throw an exception**.
This is useful when you want to **manually signal** that something is wrong, even if Java itself does not generate an exception.

⭐ 1. What is the throw Keyword?

- throw is used to **manually create and throw** an exception object.
- It immediately **stops the execution** of the current method.
- Control is transferred to the **nearest matching catch block**.
- If not caught, it is passed to the **calling method**.

✓ Syntax

```java
throw new ExceptionType("Custom message");
```

✔ Example

```java
public void divide(int a, int b) {
    if (b == 0) {
        throw new ArithmeticException("Cannot divide by zero");
    }
    int result = a / b;
    System.out.println(result);
}
```

⭐ 2. throw is for throwing, not catching

- throw **does NOT catch** exceptions.
- To catch exceptions, we MUST use **try–catch**.
- If you throw an exception **inside a method without try–catch**, it is passed to the **caller**.

⭐ 3. Using Parameterized Exception Constructors

When you want to send a meaningful error message along with the exception:

- Use **parameterized constructor**
  new ArithmeticException("Custom message")
- Not
  new ArithmeticException()

✓ Example

```
class Main {
    public static void main(String[] args) {
        int a = 0;

        try {
            if (a == 0)
                throw new ArithmeticException("a should not be zero");
        }
        catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        }
    }
}
```

Output

```
less

Exception caught: java.lang.ArithmeticException: a should not be zero
```

⭐ 4. Another Example With throw and getMessage()

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

        int j = 30;
        int i = 1;

        try {
            j = 18 / i;

            // Manually throw exception
            if (j == 0)
                throw new ArithmeticException("i do not want to print 0");
        }
        catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
            System.out.println(e.getMessage()); // prints only the message
        }
        catch (Exception e) {
            System.out.println("Exception caught");
        }
    }
}
```

✓ Explanation

- If j becomes 0, we **manually throw** an ArithmeticException.
- The catch block catches it.
- e.getMessage() prints the custom message only (not the full exception name).

⭐ 5. Key Points (Placement-Important)

- throw is used to **manually throw** exceptions.
- It is used for validation, input checking, and custom error messages.
- throw stops execution of the current block/method.
- Only **one exception** can be thrown at a time using throw.
- To catch exceptions, use **try–catch**.
- Use **parameterized constructors** for meaningful messages.
- e.getMessage() retrieves only the message passed in the exception.

---

6. Custom Exception

Java provides many built-in exceptions, but sometimes your application needs to **represent errors that are specific to your own logic**.
In such cases, Java allows you to create **custom exception classes**.

⭐ 1. How to Create a Custom Exception?

A custom exception is simply a class that **extends an existing exception class**, usually:

- Exception (for checked exceptions)
- RuntimeException (for unchecked exceptions)

✓ Basic structure:

```java
class MyException extends Exception {
    // custom exception class
}
```

Here, **MyException** is a checked exception because it extends Exception.

⭐ 2. Adding a Constructor to Pass Custom Messages

To pass a custom error message, we define a constructor that accepts a string and sends it to the superclass using super().

✓ Example:

```java
public class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
```

This allows the error message to be retrieved later using e.getMessage().

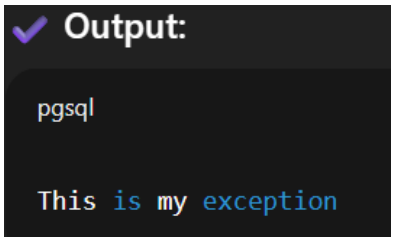
⭐ 3. Throwing a Custom Exception

You can throw your custom exception using the throw keyword.

✓ Example:

```
public class Main {
    public static void main(String[] args) {
        try {
            throw new NavinException("This is my exception");
        }
        catch (NavinException e) {
            System.out.println(e.getMessage());
        }
    }
}

class NavinException extends Exception {
    public NavinException(String s) {
        super(s);
    }
}
```



✓ Output:

This is my exception

This demonstrates how to **create**, **throw**, and **catch** a custom exception.

⭐ 4. What Can Custom Exceptions Extend?

You can extend…	Meaning	Recommended?
Throwable	Not recommended; used for system-level errors	❌ No
Exception	Creates a **checked** exception	✓ Yes
RuntimeException	Creates an **unchecked** exception	✓ Yes

⭐ 5. Recommended Practice

✓ Extend Exception

When you want the compiler to *force* the user to handle it using try-catch.

✓ Extend RuntimeException

When you want an unchecked exception (no need to declare/handle).

❌ Avoid extending Throwable

It is the parent class for both Error and Exception.
Errors should never be thrown manually.

⭐ 6. Why Create Custom Exceptions?

Custom exceptions are useful when:

- You want meaningful error messages.
- You want to enforce business rules.
- Built-in exceptions don't represent your scenario.

✓ Example use cases:

- InvalidAgeException
- InsufficientBalanceException
- UserNotFoundException
- PasswordTooWeakException

⭐ 7. Summary (Placement-Ready)

- Custom exceptions are classes that extend Exception or RuntimeException.
- Use constructors with super(message) to pass custom messages.
- Use throw to explicitly throw the custom exception.
- Catch it using a normal try-catch block.
- Avoid extending Throwable.

---

7. Duckling Exception (Throws Keyword)

⭐ 1. What is the throws Keyword?

- The **throws keyword is NOT the plural of throw**.
- It is used **in method signatures** to indicate that the method *may throw* one or more exceptions.
- It does **not** handle the exception itself.
- It simply **passes the responsibility** of handling the exception to the *caller* of the method.

✓ Syntax:

```java
returnType methodName() throws ExceptionType1, ExceptionType2 {
    // method code
}
```

✓ Key Point:

The **caller** of the method must either:

- Handle the exception using try–catch, **or**
- Continue propagating it using throws.

⭐ 2. When to Use throws Instead of try–catch?

Using throws is useful when:

✓ 1. The method is part of a larger program, and exception handling should happen at a higher level.

✓ 2. When writing reusable code or library functions, where:

- You don't know how the caller wants to handle the error.

✓ 3. When multiple methods have the same exception, and handling it once at a higher level is better.

This concept is called **ducking the exception** (letting it "pass through").

⭐ Example: Ducking Exceptions Through Methods

```
void c() {
    try {
        a();
        b();
    } catch (ArithmeticException e) {
        // handle exception here
    }
}

void a() throws ArithmeticException {}
void b() throws ArithmeticException {}
```

✓ Explanation:

- Methods a() and b() declare that they *may* throw ArithmeticException.
- They do not handle it themselves.
- The exception is **propagated** to method c().
- Finally, c() handles the exception.

⭐ When is throws Recommended?

- **Most recommended for checked exceptions**, like:
    - IOException
    - SQLException
    - ClassNotFoundException
- Less recommended for **unchecked exceptions**, because they don't need to be declared.

✓ Checked Exception Declaration Example:

```
public void myMethod() throws IOException, SQLException {
    // method logic
}
```

⭐ Best Practice

Handle exceptions at the **lowest possible level** where meaningful recovery can occur.

If a method cannot resolve the issue → propagate it using throws.

If the exception reaches main() and still isn't handled:

- The JVM's **default exception handler** terminates the program.

⭐ 3. Exception Propagation (VERY Important for Interviews)

Exception propagation means that when an exception is not handled, it **moves up the call stack** until it finds a matching catch block.

⭐ How Propagation Works (Step-by-Step)

1. A method generates (throws) an exception.
2. If the method has a matching try–catch, it will be handled there.
3. If the method does **not** handle it:
    - The exception is passed to the **calling method**.
4. The calling method either:
    - Handles it, **or**
    - Forwards it up again.
5. This continues up the call stack.
6. If no method handles it:
    - The **default exception handler** (JVM) handles it.
    - Program terminates abnormally.

⭐ Default Exception Handler (JVM)

If no method handles the exception:

- JVM prints the entire **stack trace**, including:
    - Exception type
    - Error message
    - Sequence of method calls leading to the exception
- JVM **terminates the program abnormally**

✓ **Example of stack trace:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at A.a(A.java:10)
    at B.b(B.java:5)
    at Main.main(Main.java:3)
```

⭐ **Summary (Placement-Ready)**

✓ **throw**

- Manually throw an exception.

✓ **throws**

- Declare that a method may throw an exception.
- Pass responsibility to the caller.

✓ **Exception Propagation**

- If not caught → exception travels up the call stack.
- If no catch found → JVM handles it and prints stack trace.
- Program ends abnormally.

8.Try With Resources

Java Try-with-Resources — Placement Notes

Java's **try-with-resources** feature helps developers automatically close resources (like files, connections, readers, writers) **without using a finally block**.
It simplifies resource management and prevents memory leaks.

⭐ **Why do we need try-with-resources?**

❓ **1. What is try-with-resources?**

It's a try block that **automatically closes resources** after use.

❓ **2. Can we close resources without try-with-resources?**

Yes. Before Java 7, we used **try + finally** to manually close resources.

❓ 3. Then why use try-with-resources?

Because it:

- Avoids long boilerplate code
- Automatically closes resources
- Reduces chances of forgetting to close resources
- Works even when exceptions occur

⭐ 1. Closing Resources Using try + finally (Old Method)

You can use **try with finally without catch** to close resources manually.

✓ Example (Old Way):

```
BufferedReader br = null;

try {
    InputStreamReader isr = new InputStreamReader(System.in);
    br = new BufferedReader(isr);

    System.out.println("Enter your name:");
    String name = br.readLine();

    System.out.println("Hello, " + name + "!");
}
finally {
    if (br != null)
        br.close();   // ensure resource is closed
}
```

✓ Key Point

finally **always executes**, whether exception occurs or not.
So before Java 7, this was the main way to close resources.

⭐ 2. try-with-resources (Modern Way — Java 7+)

Java 7 introduced **try-with-resources**, which closes resources **automatically**.

✓ Example (Modern Method):

```
try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {

    System.out.println("Enter your name:");
    String name = br.readLine();

    System.out.println("Hello, " + name + "!");
}
catch (IOException e) {
    // handle exception
}
```

✓ What changed?

- No need for finally
- No need to manually close br
- Cleaner and safer code

⭐ 3. Using Multiple Resources in try-with-resources

You can declare **multiple resources** inside the try parentheses.

✓ Example:

```
try (Resource1 res1 = new Resource1();
     Resource2 res2 = new Resource2()) {

    // code using res1 and res2

}
catch (Exception e) {
    // exception handling
}
```

✓ Requirements:

Resources must implement the **AutoCloseable** interface.

Examples of built-in AutoCloseable classes:

- BufferedReader
- FileInputStream
- Scanner
- Connection (JDBC)
- PreparedStatement, ResultSet

⭐ 4. Passing Pre-Created Resources

If resources are created earlier, they can still be used in try-with-resources:

```
Resource1 r1 = new Resource1();
Resource2 r2 = new Resource2();
try (r1; r2) {
   // use r1 and r2
}
```

⭐ 5. Order of Closing Multiple Resources

Resources are closed in reverse order of their declaration.

Example:

```
try (Resource1 r1 = new Resource1();
    Resource2 r2 = new Resource2();
    Resource3 r3 = new Resource3()) {


}
```

✓ Closing order:

1. **r3** (last declared → closed first)
2. **r2**
3. **r1** (first declared → closed last)

✓ Reason:

If Resource3 depends on Resource2 or Resource1, closing happens safely.

⭐ 6. Why is the Declaration Order Important?

If resources depend on each other (example: DB connection → statement → resultset), then declare in correct order:

```
try (Connection con = getConnection();
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery(query)) {
   // use rs, st, con
}
```

Closing order:

1. rs
2. st
3. con

This prevents resource dependency issues.

⭐ **Summary (Placement-Ready Points)**

- try-with-resources automatically closes resources — no need for finally.
- Works only for resources that implement **AutoCloseable**.
- Multiple resources can be declared.
- Resources are closed in **reverse** order (right → left).
- Cleaner and safer than manual try+finally resource handling.

9.Threads

⭐ 1. How Your Application Runs (OS → Hardware → CPU → Threads)

When you run any Java program, it does **not run directly on hardware**.
There is a full stack underneath:

✓ Software (your program)

⬇

✓ Operating System (OS)

- Manages programs
- Manages memory
- Schedules CPU time
- Supports **multitasking**

⬇

✓ Hardware

Contains mainly:

- **RAM** → Temporary workspace where data and instructions are loaded
- **CPU** → Executes instructions, performs calculations, handles processes/threads

---

⭐ Multitasking (OS Level)

✓ Definition:

Multitasking is the **ability of the OS + CPU to run multiple programs at the same time**.

Examples:

- Running Chrome + Spotify + VSCode simultaneously.

✓ How does multitasking work?

CPU uses:

- **Time-sharing** (each task gets a small slice of CPU time)
- **Context switching** (CPU quickly switches between tasks)

This switching happens so fast that all applications appear to run **simultaneously**, even though the CPU is rapidly switching between them.

⭐ Key Concepts

- CPU runs **one instruction at a time**, but by switching extremely fast, it gives the illusion of parallel execution.
- RAM acts as the **temporary memory** for currently running programs.
- OS keeps track of all processes and assigns CPU time to each.

⭐ 2. Inside One Program: We Can Create Multiple Threads

Just like the OS runs multiple programs, **a single program** can also run multiple small tasks **at the same time**.

✓ Thread = smallest unit of execution

- A **thread** is a lightweight sub-part of a program.
- Multiple threads can run inside one single program.
- Each thread performs a separate task.
- All threads inside a process share:
    - Same memory
    - Same resources
    - Same code

⭐ Multithreading

✓ Definition:

Multithreading is a programming model where **multiple threads** are created within a single process **to perform tasks concurrently**.

✓ Benefits of multithreading:

- Better CPU utilization
- Faster performance for concurrent tasks
- Improved application responsiveness
- Useful for large tasks split into smaller parallel tasks

✓ Real-life examples:

- Video player:
    - One thread → plays video
    - Another → plays audio

    - Another → handles user input
- Web browsers:
    - One tab = one thread
    - UI rendering and network loading run in separate threads
- Games:
    - Threads for movement, background music, physics, rendering

---

⭐ Multitasking vs Multithreading (Interview Point)

Feature	Multitasking	Multithreading
Level	OS Level	Program Level
Meaning	Multiple programs run at same time	Multiple threads run inside one program
Memory	Separate memory per process	Shared memory between threads
Examples	Chrome + Spotify	Separate threads inside Chrome tab

⭐ Final Summary (Placement-Ready)

- Applications run on OS → OS runs on hardware.
- CPU + OS support **multitasking** using time sharing and context switching.
- A **thread** is the smallest execution unit.
- **Multithreading** means running multiple threads inside the same application.
- Multithreading improves performance, responsiveness, and CPU utilization.

10.MultiThreading

.

Multithreading in Java — Creating Multiple Threads

In this lecture, we cover:

- Creating multiple threads
- How parallel programming works
- Difference between start() and run()
- How the OS schedules multiple threads
- Time-sharing between threads

⭐ 1. Why Do We Need Multiple Threads?

When you build an application (web, mobile, Spring Boot, games), frameworks internally create **multiple threads** to run different tasks simultaneously.

✓ Default Execution

- The program begins in the **main thread**.
- Statements inside main() run **one after another** (sequential execution).

✓ When do we need threads?

If you want **two pieces of logic to run at the same time**, you need to create additional threads.

Example:

- One thread for file downloading
- One thread for UI updates
- One thread for calculations

✓ Important Rule:

➡️ **Normal objects cannot run in parallel. Only threads can.**

⭐ 2. How to Create a Thread in Java?

Java provides the **Thread** class to create and manage threads.

Two main ways to create a thread:

1. **Extending Thread class**
2. **Implementing Runnable (recommended in real projects)**

✓ Extending Thread Class

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

✓ Important Note

➡️ **Just extending Thread does NOT create a new thread.**
It only *creates a class capable of becoming a thread*.
You must call **start()** to really start a new thread.

⭐ 3. start() vs run() — MOST IMPORTANT TOPIC

✓ run()

- Contains the code a thread executes.
- If you call run() directly → **no new thread is created**.
- Runs like a normal method inside the main thread.

✓ start()

- Present inside the Thread class.
- Creates a **new thread**.
- Internally calls run() on that thread.

✓ Code Example

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

public class Main {
    public static void main(String[] args) {
        A obj = new A();
        obj.start();  // creates a new thread and runs run()
    }
}
```

✓ **Key Point**

➡️ **start() = new thread + calls run()**
➡️ **run() = normal method call (no new thread)**

⭐ 4. Parallel Execution & Time Sharing

Multiple threads **do not** run exactly at the same time.
The CPU switches between them extremely fast.

This mechanism is called **time-sharing**.

✓ **How it works:**

- Each thread gets a small CPU time slice.
- CPU switches between threads very quickly → feels like parallel execution.
- The OS contains a **Scheduler** that decides:
  - Which thread runs next
  - How long a thread gets CPU time
  - When to pause and switch to another thread

✓ **Key Point:**

➡️ **Scheduler = thread manager of the OS**

Threads are not guaranteed to run in any fixed order.

⭐ Summary (Placement Ready)

1. Application frameworks internally create many threads.
2. To execute two tasks simultaneously → use threads.
3. Extend Thread or implement Runnable to create a thread.
4. start() creates a new thread and calls run().
5. Calling run() directly does NOT start a new thread.
6. OS uses **time-sharing** to rapidly switch threads.
7. Scheduler decides which thread runs and when.

---

11. Thread Priority

⭐ 1. Thread Priority in Java

In Java, every thread has a **priority**, which is used by the OS **Scheduler** to decide (or suggest) which thread should run first.

✓ Can we control the Scheduler?

❌ No.
We cannot control the OS scheduler.
We can only **suggest** priorities to it.
The actual execution order depends on:

- OS
- JVM implementation
- CPU load

⭐ Priority Range

Java gives every thread a priority between **1 and 10**:

Priority Value	Meaning
1	MIN_PRIORITY
5	NORM_PRIORITY (default)
10	MAX_PRIORITY

✓ Default Priority

Every thread by default has **priority = 5**.

⭐ Important Methods

✓ getPriority()

Returns the current priority of the thread.

```
Thread t = new Thread();
System.out.println(t.getPriority());  // 5
t.setPriority(8);
System.out.println(t.getPriority());  // 8
```

⭐ Does priority guarantee execution order?

❌ **No.**
Different OS schedulers use different algorithms.
Priority is just a **suggestion**, not a command.

Example:

- A short task with low priority may run before a long task with high priority.
- The scheduler always tries to optimize CPU efficiency.

⭐ 2. sleep() Method in Threads

You can make a thread **pause execution** for some time using the sleep() method.

✓ Syntax:

Thread.sleep(milliseconds);

✓ Example:

```
try {
    Thread.sleep(1000);  // thread sleeps for 1 second
}
catch (InterruptedException e) {
    e.printStackTrace();
}
```

⭐ What happens when a thread sleeps?

- The thread goes into **WAITING / TIMED_WAITING** state.
- After the sleep time finishes → thread becomes **RUNNABLE** again.
- It does **not** lose its priority.
- The scheduler decides when to resume it.

⭐ Why does sleep() throw InterruptedException?

Because another thread can interrupt the sleeping thread.
Example: another thread calls interrupt() on it.

Hence, we must use **try-catch**.

⭐ 3. Waiting State in a Thread

Sleep is one of the ways to put a thread into a **waiting or blocked state**.

✓ A thread enters waiting when:

- You call sleep()
- You call wait()
- You use join()
- The thread is blocked for I/O
- The thread waits for a lock (BLOCKED state)

During waiting:

- Thread does **not** run
- Scheduler does **not** schedule it
- Control returns after the waiting time or event

⭐ 4. Important Concept (Interview-Important)

❌ Programmers cannot control the CPU or threads.

✓ We can only suggest, optimize, and request, not force.

Examples:

- Priority is a suggestion
- sleep() is a request
- Scheduler has full control

⭐ Summary (Placement Ready)

✓ Thread Priority

- Range: **1 to 10**
- Default: **5**
- Methods: getPriority(), setPriority()
- Scheduler may ignore priority

✓ sleep() Method

- Pauses thread temporarily
- Uses milliseconds

- Throws InterruptedException
- Moves thread to waiting state

✓ **Waiting State**

- Thread is paused / not eligible for execution
- Happens due to sleep(), join(), wait() etc.

✓ **Final Note**

➡️ Programmers cannot control threads fully—only the OS scheduler decides.

---

12. Runnable VS Thread

Creating Threads Using Runnable Interface — Placement Notes

In this lecture, we learn:

- Using threads through the Runnable interface
- Starting a thread using Runnable
- Difference: **extending Thread vs implementing Runnable**
- Using **anonymous class** with Runnable
- Creating threads using **lambda expressions**

⭐ **1. Why Use Runnable Instead of Extending Thread?**

✓ **Java does NOT support multiple inheritance.**

If a class already extends another class, it **cannot** also extend Thread.

This makes extending Thread a bad design choice because it **blocks inheritance**.

✓ **Thread is already a class which implements Runnable**

Internally:

Thread implements Runnable

Runnable is a *functional interface* that contains a **single method**:

public void run();

✓ Creating a Thread using Runnable

```
class A implements Runnable {
    public void run() {
        System.out.println("Thread task");
    }
}
```

This is better than extending Thread because:

- Your class is free to extend another parent class
- Cleaner object-oriented design
- Runnable separates **task** from **thread mechanics**

⭐ 2. How to Start a Thread Using Runnable

Runnable interface **does NOT contain** the start() method.
So we cannot start a thread directly with a Runnable object.

✓ How do we start the thread then?

Thread class has a constructor:

Thread(Runnable target)

We pass our Runnable object to Thread.

✓ Example:

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

public class Main {
    public static void main(String[] args) {
        Runnable r = new A();     // Step 1: Create Runnable object
        Thread t = new Thread(r); // Step 2: Pass to Thread
        t.start();                // Step 3: Start thread
    }
}
```

⭐ 3. Two Ways to Create a Thread

Method	How	Good for
✓ **Extending Thread**	class A extends Thread	small/simple programs
✓ **Implementing Runnable**	class A implements Runnable	real projects, scalable design

✓ Key Difference:

Feature	Extending Thread	Implementing Runnable
Multiple inheritance	❌ Not possible	✓ Possible
Separation of task & thread	❌ No	✓ Yes
Reusability	Low	High
Industry preference	Low	High

Conclusion:

➡️ **Runnable** is the recommended approach.

⭐ 4. Using Anonymous Class with Runnable

We can create a Runnable without writing a separate class.

✓ Anonymous Class Example:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Thread using anonymous class");
    }
});

t.start();
```

No need to create a named class — useful for small tasks.

⭐ 5. Runnable with Lambda Expression (BEST & MODERN WAY)

Since Runnable is a **functional interface**, we can use **lambda expressions**.

✓ Lambda Example:

```
Thread t = new Thread(() -> {
    System.out.println("Thread using lambda");
});

t.start();
```

This is clean, short, and preferred in Java 8+.

⭐ Summary (Placement Ready)

✓ Why Runnable?

- Avoids multiple inheritance problems
- Better design & reusability
- Recommended in real-world apps

✓ How to start a thread?

- Runnable has only run() → no start()
- Must pass Runnable to Thread
- Then call start()

✓ Thread Creation

1. Extend Thread (not preferred)
2. Implement Runnable (preferred)

✓ Advanced Features

- Runnable used with **anonymous classes**
- Runnable used with **lambda expressions** (Java 8+)

13.Race Condition

⭐ 1. Threads and Mutations

✓ Threads

- A **thread** allows you to run multiple tasks at the same time (parallel / concurrent execution).
- Most threads in real applications (Spring, Android, servers) are created by the **framework**, not manually.
- Threads are used to increase speed and responsiveness.

✓ Mutations

- **Mutation** means *changing state*.
- Primitive variables (int, float, etc.) are *mutable*.
- Objects whose state can change are *mutable*.
- **Strings are immutable** — once created, their value cannot change.

❗ Problem:

Using **threads + mutable data** together is risky.
If multiple threads modify the same data at the same time → instability.

⭐ 2. Thread Safety in Java

✓ What is Thread Safety?

A piece of code is **thread-safe** if it works correctly even when accessed by multiple threads simultaneously.

Thread safety ensures:

- Only **one thread** works on shared data at a time
- Other threads must **wait**

✓ Example Scenario

Suppose two threads both call:

count++;

each 1000 times.

Expected output: **2000**
Actual output: **varies each run (1900, 1800, 2000, etc.)**

✓ **Why does this happen?**

Because main thread prints the value of count **before** other threads finish.

✓ **Solution**

Use **join()** to ensure main thread waits for worker threads.

⭐ **3. join() Method and synchronized Keyword**

✓ **join()**

join() tells a thread to **wait** for another thread to complete.

t1.join();

t2.join();

This ensures:

- Main thread pauses
- t1 and t2 finish completely
- Only then main thread continues
- Prevents inconsistent output

✓ **Why does join() throw InterruptedException?**

Because another thread may *interrupt* it.
So join() requires **try-catch** or **throws clause**.

⭐ **When Two Threads Access the Same Method at the Same Time**

If two threads access a shared method simultaneously:

- They may **overwrite each other's values**
- Some operations get **lost**
- Data becomes **corrupted**

Example:

- Two threads increment a shared variable
- Both run at the same time
- The increment steps overlap and some increments are lost

This is where **synchronized** is required.

⭐ synchronized Keyword

synchronized ensures that:

- Only **one thread** accesses the method at a time
- Other threads must **wait**
- Fixes data corruption
- Prevents inconsistent results

✓ synchronized Example:

```
synchronized void increment() {
   count++;
}
```

✓ What synchronized does:

- Acquires a **lock** on the object
- Other threads wait until lock is released
- Ensures safe updates to shared data

✓ Synchronized is used for:

- Methods
- Blocks of code
- Shared resources
- Critical sections

⭐ 4. Race Condition (VERY IMPORTANT)

A **race condition** happens when:

- Two or more threads try to access the **same shared resource**
- At the **same time**
- Without synchronization
- Causing **incorrect**, unstable, or unpredictable output

✓ Race Condition Example:

```
count++;  // critical section
```

If two threads enter this line at the same time → incorrect results.

✓ Solution:

➡️ Use **synchronized** to control access.
➡️ This is why synchronization exists — to **prevent race conditions**.

⭐ **Summary (Placement Ready)**

✓ Threads and Mutations

- Threads run tasks in parallel
- Mutable data + threads = instability

✓ Thread Safety

- Ensures only one thread updates shared data at a time

✓ join()

- Makes main thread wait for other threads
- Prevents premature output

✓ synchronized

- Allows only one thread to execute a method / block at a time
- Prevents inconsistent results
- Ensures thread safety

✓ Race Condition

- Happens when multiple threads access critical code simultaneously
- Causes incorrect behavior
- Fixed using **synchronized**

---

14. Thread State

.

⭐ 1. Different States of a Thread

A thread does not run immediately after creation.
It moves through the following states:

1️⃣ NEW State

- When a thread object is created using new, it is in **NEW** state.
- It has not started executing yet.

Thread t = new Thread(); // NEW

2️⃣ RUNNABLE State

When you call start(), the thread moves into **RUNNABLE** state.

✓ What Runnable Means:

- The thread is **ready to run**.
- It is waiting for the **scheduler** to give CPU time.
- It *may or may not* be executing at this moment.

Important:
➡️ Runnable does **not** mean the thread is running.
➡️ It only means it is *eligible* to run.

3️⃣ RUNNING State

This is when:

- The thread gets CPU time
- The scheduler picks this thread
- The run() method is executing

✓ **Runnable vs Running**

Runnable	**Running**
Thread is ready to run	Thread is actually executing
Waiting for CPU time	Using CPU time
Scheduler decides when to run	The thread is on CPU

Only the scheduler decides when a thread becomes Running.

4️⃣ WAITING / BLOCKED / TIMED_WAITING State

A thread enters the waiting state when:

- sleep() is used
- wait() is used
- Thread is waiting for I/O
- Thread is waiting for another thread (via join())
- Thread is waiting for lock (BLOCKED state)

✓ **sleep()**

Moves thread to **TIMED_WAITING** state.

✓ **wait()**

Moves thread to **WAITING** state.

✓ **notify() / notifyAll()**

Moves thread back to **RUNNABLE** state.

5️⃣ DEAD (TERMINATED) State

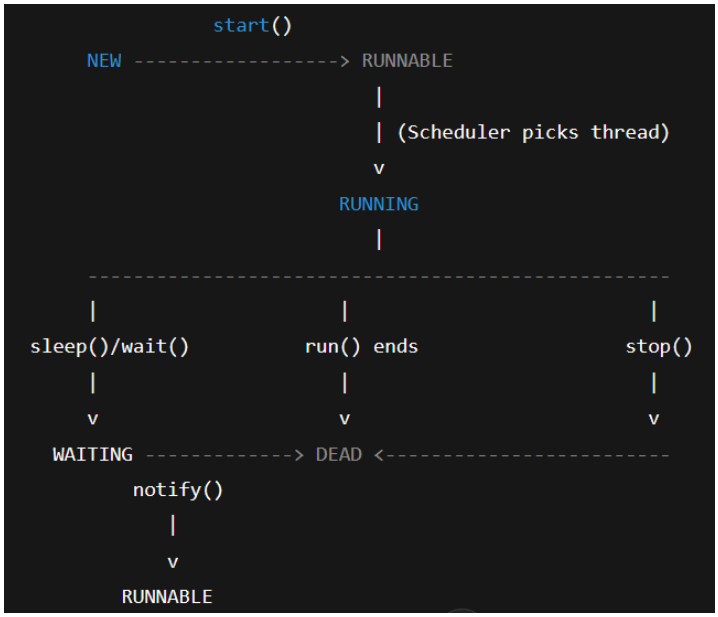
A thread enters this state when:

- Its run() method completes normally
- Or you call stop() (deprecated and unsafe)

Once a thread is dead:
❌ It cannot be restarted again.

⭐ 2. Lifecycle Flow — Text-Based Diagram

Here is your corrected, clean flowchart representation:



⭐ 3. Important Notes on Thread States

✓ A thread becomes RUNNING only when the scheduler selects it

You cannot force a thread to run.

✓ sleep() or wait() moves thread to WAITING state

During WAITING:

- Thread is **not running**
- Thread does **not** get CPU time
- Scheduler does not pick it until notified or timeout ends

✓ notify() and notifyAll() bring thread back to RUNNABLE

From there, scheduler decides execution order.

✓ stop() is deprecated

It forces thread termination and may cause data corruption.
Modern Java uses:

- interrupt()
- Flags/conditions
- Shared variables for controlled stopping

⭐ 4. Runnable vs Running (Interview GOLD Question)

Feature	Runnable	Running
Definition	Ready to run	Actively executing
CPU Time	Not yet assigned	Assigned
Controlled by	Programmer (start(), notify())	Scheduler only
State	Runnable	Running

Simple Explanation:

➡️ Runnable → "I am ready to run."

➡️ Running → "I am actually running on the CPU."

⭐ Summary (Placement-Ready)

- NEW → Thread created.
- RUNNABLE → Thread ready to run.
- RUNNING → Thread executing run().
- WAITING → Thread paused via sleep(), wait(), join(), lock.
- DEAD → Thread finished task.
- start() → NEW → RUNNABLE
- run() executes only when thread is RUNNING
- sleep()/wait() → WAITING
- notify() → WAITING → RUNNABLE
- Scheduler decides the execution order
- stop() kills a thread and is unsafe