Multithreading in Java

Java Multithreading – Introduction

1. What is Multithreading?

Multithreading is a core concept in Java that allows multiple parts of a program (called threads) to run **concurrently**. Each thread runs independently and shares the same memory space.

It is part of Java’s java.lang package through the Thread class and Runnable interface.

Real-life Analogy

Think of your computer as a **restaurant kitchen**:

- A **thread** is like a **chef**.
- Each chef (thread) can cook (execute) a task at the same time.
- All chefs share the same kitchen (memory/resource).

So multithreading = multiple chefs working in the same kitchen = faster task completion

---

Why Use Multithreading?

- Increases performance by executing multiple tasks at once.
- Makes the application **responsive** (e.g., a UI app doesn't freeze during background tasks).
- Useful for **CPU-intensive** or **I/O-intensive** operations.
- Efficient utilization of system resources.

□ Key Concepts

Concept	Description
Thread	A lightweight process; smallest unit of execution
Main Thread	The default thread in any Java program
Concurrency	Tasks are progressing at the same time
Parallelism	Tasks are **actually running at the same time** (multi-core CPU)
Thread Lifecycle	The stages a thread goes through (New → Runnable → Running → Blocked → Dead)

Advantages of Multithreading

- Better CPU utilization
- Enhanced performance for large programs
- Better responsiveness (UI apps)
- Simplifies program structure for asynchronous tasks

Challenges of Multithreading

- Difficult to debug
- Risk of **Race conditions** and **Deadlocks**
- Requires careful synchronization

Creating Threads in Java

In Java, **threads** can be created in two main ways:

---

1. By Extending the Thread Class

You create a new class that **inherits from Thread**, and override its run() method with the code you want to execute in the thread.

□ Steps:

1. Create a class extending Thread.

2. Override the run() method.
3. Create an object of the class.
4. Call .start() to begin execution.

□ **Example:**

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

public class TestThread {
   public static void main(String[] args) {
      MyThread t1 = new MyThread(); // Create thread object
      t1.start(); // Start the thread (calls run() internally)
   }
}
```

□ start() method internally calls run(), and allows Java to execute the thread **asynchronously**.

---

2. By Implementing the Runnable Interface

You define your thread logic in a class that implements Runnable, then pass it to a Thread object.

□ **Steps:**

1. Create a class implementing Runnable.
2. Override the run() method.
3. Create a Thread object with the Runnable object.
4. Call .start() on the Thread.

□ **Example:**

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

```
    }
}

public class TestRunnable {
    public static void main(String[] args) {
        MyRunnable obj = new MyRunnable();     // Step 1
        Thread t1 = new Thread(obj);           // Step 2
        t1.start();                            // Step 3
    }
}
```

□ Thread vs Runnable

Feature	Thread Class	Runnable Interface
Inheritance	Cannot extend any other class	Can extend another class
Flexibility	Less flexible	More flexible (preferable in large apps)
Memory overhead	More (Thread object + task)	Less
Recommended	Not preferred in multi-inheritance cases	Preferred

---

□ When to Use What?

- Use **Runnable** when you want to separate the **task** from the **thread object** or inherit from another class.
- Use **Thread** when you want **simplicity** and don't need to extend another class.

□ Common Mistake

```
t1.run();  // □  runs in the main thread (not a new thread)
t1.start(); // □  starts a new thread
```

Thread Scheduler in Java

What is a Thread Scheduler?

The **Thread Scheduler** is part of the **Java Virtual Machine (JVM)** responsible for **deciding which thread to run next**, when multiple threads are in a **runnable** state.

 It **does not guarantee** the order of thread execution — it is **OS-dependent and JVM-dependent**.

How it Works

When multiple threads are eligible to run, the **scheduler picks one based on a strategy**, such as:

- **Time Sharing** – Threads are given a fixed time slice (quantum) to run.
- **Preemptive Scheduling** – Higher priority threads are given preference and can preempt lower priority threads.

 Only one thread executes at a time on a single-core CPU. In multicore systems, threads may run truly in parallel.

Example – Unpredictable Execution Order

```
class DemoThread extends Thread {
    public void run() {
        System.out.println("Thread: " + Thread.currentThread().getName());
    }
}

public class ThreadSchedulerExample {
    public static void main(String[] args) {
        DemoThread t1 = new DemoThread();
        DemoThread t2 = new DemoThread();
        DemoThread t3 = new DemoThread();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output (May vary every time)

```
Thread: Thread-0
Thread: Thread-2
Thread: Thread-1
```

This proves that **thread scheduling is not deterministic**.

□ Thread Priorities in Java

Every Java thread has a **priority** (integer from 1 to 10).

Priority Constant	Value	Meaning
Thread.MIN_PRIORITY	1	Lowest
Thread.NORM_PRIORITY	5	Default
Thread.MAX_PRIORITY	10	Highest

You can set priorities using:

```
thread.setPriority(Thread.MAX_PRIORITY);
```

Important:

JVM may or may not respect priorities — it's **platform dependent**.

Relationship Between start() and run() Method in Java Threads

---

The run() Method

The run() method contains the **code that should be executed** in a new thread.

Syntax:

```
public void run() {
    // task to be performed
}
```

□ Think of run() as the **body of the thread** — the logic that needs to execute.

The start() Method

The start() method is responsible for **creating a new thread** in memory and **internally calling the run() method**.

Syntax:

```
thread.start(); // JVM creates a new thread and calls run()
```

□ start() triggers the **multithreading mechanism**. It tells the **Thread Scheduler** to execute run() in a new thread.

□ Code Example: Proper Way

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

public class StartVsRun {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Correct way to start a new thread

        System.out.println("Main Thread: " + Thread.currentThread().getName());
    }
}
```

Output:

```
Main Thread: main
Thread running: Thread-0
```

□ Common Mistake: Calling run() Directly

```
t1.run(); // Wrong – runs like a normal method on main thread
```

Output:

```
Thread running: main
Main Thread: main
```

□ This does **not create a new thread**, it just calls run() as a regular method.

□ Difference Between start() and run()

Feature	start()	run()
Thread Creation	Creates a **new thread**	No new thread; called in the **current thread**
Method Type	Native method from Thread class	Override to define task
Behavior	Executes run() in **new thread**	Executes run() in **same thread**
Multithreading	Enabled	Not enabled
Usage	Should always be used to start threads	Should be overridden to define logic

---

□ Best Practice

- Always use .start() to initiate a thread.
- Override .run() to define what the thread will do.

What is the Main Thread in Java?

Definition

The **main thread** is the **default thread** that the Java Virtual Machine (JVM) creates automatically when your program starts.

It is the **first thread** of execution in any standalone Java application.

□ Key Points

- It begins execution with the main(String[] args) method.
- All other threads (user-defined threads) are created and started **from this main thread**.
- It is part of the **Thread Group** named "main".
- The main thread is **non-daemon**, i.e., JVM will wait for it to finish.

□ Example: Viewing Main Thread Details

```
public class MainThreadExample {
    public static void main(String[] args) {
        Thread t = Thread.currentThread(); // Get reference to the current (main) thread

        System.out.println("Name: " + t.getName());
        System.out.println("ID: " + t.getId());
        System.out.println("Priority: " + t.getPriority());
        System.out.println("Group: " + t.getThreadGroup().getName());
    }
}
```

□ Sample Output:

```
Name: main
ID: 1
Priority: 5
Group: main
```

□ Main Thread Responsibilities

- Starts execution of the program.
- Creates other child threads.
- Manages the flow of the program.
- Ends only **after** all code in main() has been executed.

□ Can You Modify Main Thread?

Yes, the main thread is a regular Thread object. You can:

```
Thread.currentThread().setName("MyMainThread");
System.out.println(Thread.currentThread().getName()); // MyMainThread
```

□ Important Notes

- The main thread will **not wait** for child threads **unless you use .join()**.
- If the main thread exits before others, JVM will keep running if **non-daemon** threads are active.

□ Example: Main vs Child Thread

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

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

        System.out.println("Main Thread Running");
    }
}
```

□ Sample Output (Order not guaranteed):

```
Main Thread Running
Child Thread Running
```

Thread Life Cycle in Java

What is Thread Life Cycle?

A **Java thread** goes through various **stages (states)** during its lifetime — from creation to termination. This is known as the **Thread Life Cycle**.

Java provides built-in support to monitor and manage these states through the Thread.State enum.

□ Thread Life Cycle States

1. New

- The thread is **created** but **not yet started**.
- Created using:
  Thread t = new Thread();

2. Runnable

- The thread is **ready to run** and is waiting for CPU allocation by the **Thread Scheduler**.
- Set using:
  t.start();

3. Running

- The thread is **currently executing** its run() method.
- The actual transition from **Runnable** → **Running** is managed by the **Thread Scheduler**.

4. Blocked

- The thread is **waiting to acquire a lock** (object-level lock).
- Example: When two threads try to enter a synchronized block and one has to wait.

5. Waiting

- The thread is **waiting indefinitely** for another thread to perform a specific action.
- Example: wait(), join() without timeout.

6. Timed Waiting

- The thread is **waiting for a specific amount of time**.
- Example: sleep(1000), join(500), wait(2000)

7. Terminated (Dead)

- The thread has **completed execution** or **was forcefully stopped**.
- Cannot be restarted once terminated.

Thread Life Cycle Diagram

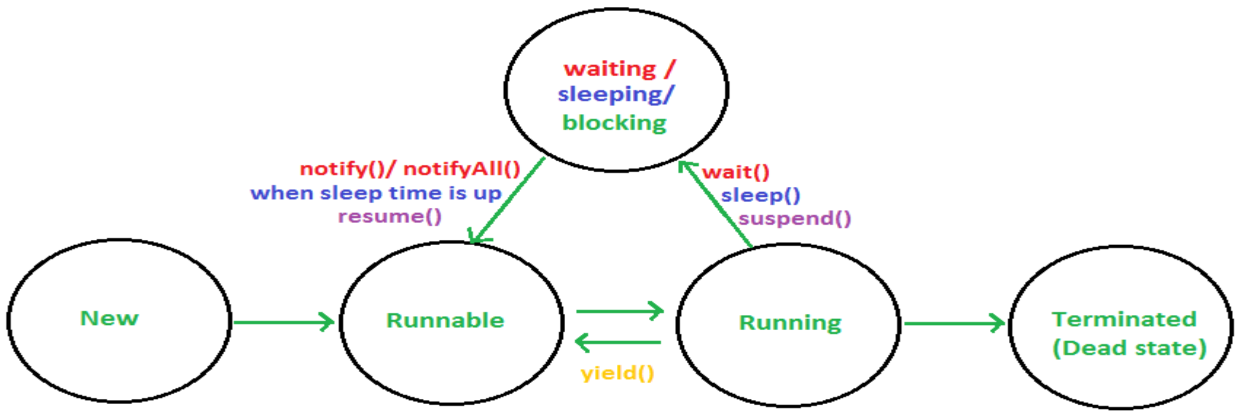

Fig. THREAD STATES

□ Important Notes

- You **can't restart** a thread once it's dead (TERMINATED).
- Only start() moves a thread from NEW to RUNNABLE.
- Manual transitions are **not possible** — they are managed by JVM and OS.

Thread Class Methods in Java

The Thread class in Java (under java.lang package) provides **many useful methods** to manage thread behavior like starting, pausing, joining, naming, etc.

Commonly Used Thread Class Methods

Method	Description
start()	Starts a new thread and calls run()
run()	Contains the thread's task logic
sleep(ms)	Pauses the thread for given milliseconds
join()	Waits for the thread to finish execution
setName(String)	Sets the thread's name
getName()	Returns the thread's name
setPriority(int)	Sets thread priority (1 to 10)
getPriority()	Returns thread priority
isAlive()	Checks if thread is still running
currentThread()	Static method to get current thread reference
interrupt()	Interrupts the thread

Method	Description
isInterrupted()	Checks if thread is interrupted

 Demonstration of Key Methods

 start() and run()

```
Thread t = new Thread(() -> System.out.println("Running thread"));
t.start();  // starts new thread
```

---

 sleep()

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

 sleep() causes the **current thread** to pause.

---

 join()

```
Thread t = new Thread(() -> {
    for (int i = 0; i < 3; i++) {
        System.out.println("Child Thread");
    }
});

t.start();

try {
    t.join();  // Waits for child thread to complete
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Main Thread continues");
```

□ setName() and getName()

```
Thread t = new Thread();
t.setName("MyWorker");
System.out.println(t.getName()); // MyWorker
```

□ setPriority() and getPriority()

```
t.setPriority(Thread.MAX_PRIORITY);
System.out.println(t.getPriority()); // 10
```

□ Priorities range from 1 to 10 (default = 5)

□ isAlive() and currentThread()

```
Thread t = new Thread(() -> {});
System.out.println(t.isAlive()); // false
t.start();
System.out.println(Thread.currentThread().getName()); // main
```

□ Special Method: interrupt()

Used to **interrupt a thread** (especially useful in long-running loops or waiting states):

```
Thread t = new Thread(() -> {
    while (!Thread.currentThread().isInterrupted()) {
        // keep running
    }
});
t.start();
t.interrupt(); // requests to stop the thread
```

Thread Priority in Java

What is Thread Priority?

In Java, **each thread has a priority**, an integer value from **1 to 10**, that **helps the thread scheduler decide** which thread to run when multiple threads are ready.

 Thread priority is a suggestion to the **Thread Scheduler**, not a command. The actual behavior depends on the **OS and JVM**.

Priority Constants

Java provides 3 predefined constants in the Thread class:

Constant	Value	Meaning
Thread.MIN_PRIORITY	1	Lowest priority
Thread.NORM_PRIORITY	5	Default priority
Thread.MAX_PRIORITY	10	Highest priority

How to Set and Get Priority

Set Priority:

```
Thread t = new Thread();
t.setPriority(Thread.MAX_PRIORITY);  // Set to 10
```

Get Priority:

```
int p = t.getPriority();
System.out.println("Priority: " + p);
```

When to Use Thread Priority?

- Use it when you want **relative importance** among threads.

- Useful in real-time or time-sensitive tasks (but still not fully reliable).
- For **absolute control**, consider Executors or other concurrency APIs.

□ Important Notes

- Priority doesn't guarantee execution order.
- Changing priority of a thread after it has started may not take effect.
- Use priority **only for optimization**, not control flow logic.

Methods to Prevent Thread Execution in Java

In multithreading, there are several methods that can **pause, stop, or delay** the execution of a thread temporarily. These are **not to terminate** a thread, but to control or coordinate its execution flow.

---

1. sleep(long ms)

Purpose: Pauses the current thread for a specific time in **milliseconds**.

Syntax:

```
Thread.sleep(1000); // Sleeps for 1 second
```

Behavior:

- The **current thread pauses** execution.
- The thread goes into the **TIMED_WAITING** state.
- After the time is up, it becomes **Runnable** again.
- Throws InterruptedException.

Example:

```
try {
    System.out.println("Sleeping...");
    Thread.sleep(2000);
    System.out.println("Awake!");
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

---

2. join()

Purpose: Waits for a thread to **finish its execution**.

Syntax:

```
t.join();       // Waits until t finishes
t.join(1000);   // Waits for 1 second max
```

Behavior:

- Used to **pause the calling thread** (e.g., main thread) until the specified thread finishes.
- Thread goes into the **WAITING** or **TIMED_WAITING** state.

Example:

```
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 3; i++) {
        System.out.println("Child Thread");
    }
});

t1.start();
t1.join(); // Main thread waits
System.out.println("Main Thread continues...");
```

---

3. yield()

Purpose: Temporarily **pauses the current thread** to give a chance for other threads of **equal priority** to execute.

Syntax:

```
Thread.yield();
```

Behavior:

- It's a **static method**.
- Suggests the thread scheduler to let other threads run.
- May or may not have any effect (JVM dependent).

Example:

```
for (int i = 0; i < 5; i++) {
    System.out.println("Running...");
    Thread.yield();  // Hint to scheduler
}
```

---

4. wait(), notify(), and notifyAll() (From Object Class)

Purpose: Used for **inter-thread communication** within synchronized blocks.

Method	Behavior
wait()	Pauses current thread until another thread calls notify()
notify()	Wakes up a single thread waiting on the object
notifyAll()	Wakes up all threads waiting on the object

Example:

```
synchronized(obj) {
    obj.wait();     // waits until notify is called
}

synchronized(obj) {
    obj.notify();   // wakes up one waiting thread
}
```

□ These are **advanced mechanisms** used for coordination between threads.

□ Deprecated: stop(), suspend(), and resume()

These methods are **deprecated and unsafe** because:

- stop() can terminate a thread **in the middle of execution**, causing inconsistent state.
- suspend() and resume() can lead to **deadlocks**.

 Use proper **flags or interrupts** to terminate threads safely.

Synchronization in Java

What is Synchronization?

Synchronization in Java is the process of **controlling access to shared resources** (like variables, methods, objects) by multiple threads to **prevent data inconsistency or race conditions**.

 Without synchronization, two or more threads may **interleave** and cause **unexpected results** when accessing the same data.

 Why Synchronization?

Imagine a bank account being updated by two threads at the same time:

- Thread 1: withdraws ₹100
- Thread 2: deposits ₹200
  If not synchronized, the final balance may be **corrupted**.

 Race Condition

Occurs when:

- Multiple threads access the same resource
- At least one thread modifies it
- No proper coordination or locking

Example (Unsynchronized):

```
class Counter {
    int count = 0;
    void increment() {
        count++; // Not thread-safe
    }
}
```

Synchronized Methods

Use synchronized keyword to lock the **entire method** so only one thread can execute it at a time.

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;  // Thread-safe
    }
}
```

Synchronized Blocks

Use synchronized(obj) to lock a **specific block of code** (for better performance):

```
class Counter {
    int count = 0;
    Object lock = new Object();

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

 Good practice when only part of the method needs to be synchronized.

** Java's Locking Mechanism**

When a thread enters a synchronized method/block:

- It **acquires a lock (monitor)** on the object.
- Other threads trying to access the same locked method/object must **wait**.
- Once the thread exits the block/method, the lock is **released**.

□ Only one thread will print the full table at a time due to synchronization.

□ **Notes**

- Synchronization affects **performance** (only one thread at a time).
- Use **minimal locking** — lock only what's necessary.
- To achieve more advanced control, use ReentrantLock (from java.util.concurrent.locks).

Synchronization Method vs Block – Which is Better Practice?

---

1. Synchronized Method

□ **Syntax:**

```
synchronized void myMethod() {
   // entire method is locked
}
```

Pros:

- Simple and easy to implement.
- Automatically locks the **current object** (this).

Cons:

- **Locks the entire method**, even if only a part of it needs synchronization.
- Can reduce **performance**, especially in long methods.

---

2. Synchronized Block

 Syntax:

```
void myMethod() {
   // some code

   synchronized(lockObject) {
      // only this part is synchronized
   }

   // other code
}
```

Pros:

- Gives **fine-grained control** over what needs to be locked.
- Improves **performance** by locking only **critical section**.
- You can choose which **object to lock** (not always this).

Cons:

- Slightly more complex to implement.
- Needs a proper lockObject.

12. Inter-Thread Communication in Java

(wait(), notify(), notifyAll())

What is Inter-Thread Communication?

Inter-thread communication allows threads to **coordinate** and **share data** safely. It lets one thread **pause** its execution and **resume** when another thread performs a specific action.

Why Use It?

- Prevents **polling** (constant checking).

- Improves **efficiency** in producer-consumer or task-waiting scenarios.
- Allows **cooperation** between threads.

Core Methods (From Object Class)

Method	Description
wait()	Tells the current thread to wait and release the lock until another thread calls notify() or notifyAll()
notify()	Wakes up one thread waiting on the object
notifyAll()	Wakes up **all** threads waiting on the object

□ Important Rules

- These methods must be called **inside a synchronized block or method**.
- The calling thread must own the **object's monitor/lock**.
- If called without synchronization, you'll get:

  java.lang.IllegalMonitorStateException

□ wait() vs sleep()

Feature	wait()	sleep()
Belongs to	Object class	Thread class
Releases lock?	Yes	No
Used for	Inter-thread communication	Delay/pause
Requires synchronized block	Yes	No

□ notify() vs notifyAll()

Method	Wakes up
notify()	One thread (arbitrary choice by JVM)
notifyAll()	All threads waiting on the object

 Use notify() when **only one thread** needs to proceed.
Use notifyAll() when **multiple threads** may proceed.

Common Mistakes

- Calling wait() or notify() **outside** a synchronized block.
- Forgetting to use while(condition) instead of if(condition) for waiting.
- Not handling InterruptedException.

Best Practice Pattern

```
synchronized(obj) {
    while (!condition) {
        obj.wait();
    }
    // perform task
    obj.notify();
}
```

13. Advanced Multithreading in Java

---

1. Lock and ReentrantLock (From java.util.concurrent.locks)

Java introduced **explicit locking** via the Lock interface for **better control** than synchronized.

What is ReentrantLock?

A **ReentrantLock** is an implementation of Lock that allows the **same thread to acquire the lock multiple times** without deadlocking.

Key Methods:

Method	Description
lock()	Acquires the lock
unlock()	Releases the lock
tryLock()	Attempts to get the lock without waiting
lockInterruptibly()	Allows thread to be interrupted while waiting

---

2. ExecutorService – Managing Threads Efficiently

What is it?

ExecutorService is a **thread pool manager** from java.util.concurrent. It creates and reuses a pool of threads, instead of creating new threads manually.

□ Common Types:

Executor Type	Description
newFixedThreadPool(n)	Reuses a fixed number of threads
newCachedThreadPool()	Creates new threads as needed
newSingleThreadExecutor()	One thread only

---

3. Callable and Future

What is Callable?

- Similar to Runnable, but it **returns a result** and can **throw exceptions**.
- Generic: Callable<T> returns type T.

What is Future?

- Represents the **result of a computation** that may not have completed yet.
- Provides methods like get(), isDone(), and cancel().

Lock vs synchronized in Java

Which is better? When to use what?

---

1. Basic Definitions

Feature	synchronized	Lock (like ReentrantLock)
Type	Keyword (built-in)	Interface (java.util.concurrent.locks)
Locking	Implicit	Explicit (manual)
Introduced In	Java 1.0	Java 1.5

---

2. Syntax Comparison

□ synchronized

```
synchronized(obj) {
    // critical section
}
```

□ Lock (ReentrantLock)

```
lock.lock();
try {
    // critical section
} finally {
    lock.unlock(); // must release
}
```

---

3. Key Differences

Criteria	**synchronized**	**Lock**
Lock Acquisition/Release	Automatically acquired & released	Manual: must call lock() & unlock()
Try Locking	Not possible	tryLock() avoids blocking
Interruptible Lock	No support	lockInterruptibly() supported
Fairness	No fairness options	Can create fair locks (new ReentrantLock(true))
Read/Write Locking	Not supported	Supported via ReadWriteLock
Condition Waiting	Uses wait(), notify()	Uses Condition object (await(), signal())
Performance	Simple and sufficient for basic use	More flexible and scalable for complex apps

---

4. When to Use

Situation	**Use**
Simple critical section	synchronized is enough
Multiple lock conditions	Prefer Lock
Need to try for lock (non-blocking)	Use tryLock() from Lock
Require timeout or interruptible lock	Use Lock
Want fine-grained control	Use Lock

ExecutorService – Thread Pool Methods in Java

What is ExecutorService?

ExecutorService is part of java.util.concurrent and provides a **flexible thread pool management system**, allowing better control over **thread creation, reuse, and shutdown**.

 Think of it as a **thread manager** that efficiently handles background tasks using a **pool of threads**.

Why Use Thread Pools?

- Reuse threads instead of creating new ones each time.
- Improve performance and reduce memory overhead.
- Manage threads systematically.
- Prevents thread exhaustion.

Common Factory Methods (from Executors class)

Method	Description
Executors.newFixedThreadPool(int n)	Creates a pool with a fixed number of threads
Executors.newCachedThreadPool()	Creates threads as needed, reuses idle ones
Executors.newSingleThreadExecutor()	Executes tasks one by one (sequentially)
Executors.newScheduledThreadPool(int n)	Schedules tasks to run after delay or periodically

---

Key ExecutorService Methods

Method	Description
submit(Runnable/Callable)	Submits a task for execution (may return Future)
execute(Runnable)	Submits a task (no result expected)
shutdown()	Gracefully shuts down the executor (no new tasks)
shutdownNow()	Attempts to stop all tasks immediately
isShutdown()	Returns true if shutdown() was called
isTerminated()	Returns true if all tasks have finished
awaitTermination(timeout, unit)	Waits for existing tasks to finish

Future vs Callable in Java

---

1. What is Callable?

- Callable is a **functional interface** used to **create tasks that return a result** and may **throw exceptions**.
- Introduced in Java 5 (java.util.concurrent).

Syntax:

```
Callable<String> task = () -> {
   return "Result";
};
```

- Used with ExecutorService.submit(Callable) to execute tasks.
- Returns a Future object.

---

2. What is Future?

- Future is an **interface** that represents the **result of an asynchronous computation**.

- It is returned when a task is submitted using Callable (or Runnable) to an ExecutorService.

 Key Methods:

Method	**Description**
get()	Waits and returns the result
isDone()	Returns true if task is finished
cancel()	Tries to cancel the task
isCancelled()	Checks if task was cancelled

Key Differences: Callable vs Future

Feature	**Callable**	**Future**
Type	Interface	Interface
Purpose	To create tasks that return a value	To **retrieve results** of submitted tasks
Returns	Result (via call() method)	Result (via get() method)
Throws Exception	Yes	No (wraps exceptions from call() inside ExecutionException)
Submitted to	ExecutorService.submit()	Returned by submit() method
Example Use	Creating task logic	Handling task result