Java Thread Runnable

✅ Overview: `Thread` Class vs `Runnable` Interface in Java

Both **`Thread`** class and **`Runnable`** interface are used to create and manage threads in Java, but they differ in how you use them and what flexibility they offer.

🧵 1. `Thread` Class

- Java provides the **`Thread`** class in the **`java.lang`** package.
- To create a thread by extending **`Thread`**, you override its **`run()`** method.

📌 Example:

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

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();  // Starts a new thread and invokes run()
    }
}
```

✅ Key Points:

- Inherits from **`Thread`** class.
- Cannot extend another class (Java doesn't support multiple inheritance with classes).
- Use when you don't need to extend any other class.

🔁 2. `Runnable` Interface

- **`Runnable`** is a functional interface with a single method: **`run()`**.
- To use it, implement the **`Runnable`** interface and pass the instance to a **`Thread`** object.

📌 Example:

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

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable);
        t1.start();  // Starts thread and invokes run()
    }
}
```

✅ Key Points:

- Implements `Runnable` interface.
- More flexible: allows you to extend another class.
- Recommended approach in real-world applications, especially when using thread pools or executors.

🔍 Comparison Table

Feature	Thread Class	Runnable Interface
Inheritance	Extends `Thread`	Implements `Runnable`
Flexibility	Less (can’t extend another class)	More (can extend other classes)
Recommended for	Simple threads	Scalable & real-world applications
Thread creation	`new MyThread().start()`	`new Thread(new MyRunnable()).start()`
Use with Thread Pools	Not directly	Yes (ExecutorService)

💡 Best Practice

> **Use `Runnable` interface** (or `Callable` if you need a return value) for most practical multi-threading in Java. It's more flexible, cleaner, and better suited for modern concurrency APIs like `ExecutorService`.

Let me know if you'd like to see:

- Example using `ExecutorService`
- `Callable` interface overview
- Thread lifecycle or synchronization examples

🧵 What is a Thread in Java?

A **thread** in Java is a **lightweight, independent path of execution** within a program.
Think of it as a **small unit of a process** that can run **concurrently** with other threads.

🔄 Why Threads?

In a typical Java application, the **main()** method runs in a **single thread** called the **main thread**.
But sometimes we want to do multiple things **at the same time**, like:

- Downloading a file while showing a progress bar
- Running background tasks without freezing the UI
- Handling multiple client requests in a server

➡️ That's where **threads** help — they allow **parallel execution** of code.

🧠 Key Concepts

Concept	Description
Thread	A unit of execution within a process
Multithreading	Running two or more threads concurrently
Main Thread	The default thread started by the JVM
Start()	Starts a thread and invokes its **`run()`** method
Run()	Contains the code to be executed in the thread
Thread Scheduler	Decides which thread to run when (handled by JVM/OS)

🧪 Example

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

    public static void main(String[] args) {
        MyThread t1 = new MyThread();  // Create a new thread
        t1.start();                    // Start the thread
    }
}
```

> **`start()`** creates a **new call stack** for the thread and calls **`run()`** internally.

🔁 Thread Lifecycle

1. **New** – Thread is created but not started.
2. **Runnable** – After **`start()`** is called; ready to run.

3. **Running** – Actually executing.
4. **Blocked/Waiting** – Waiting for a resource or another thread.
5. **Terminated** – Finished execution or stopped.

✅ Real-Life Analogy

Imagine a **kitchen** where multiple **chefs (threads)** are working **at the same time** (concurrently):

- One is **cutting vegetables**
- Another is **boiling water**
- Another is **making dough**

They all work in **parallel** to complete the task faster — that's multithreading!

Let me know if you want to learn:

- Thread lifecycle with diagram
- Thread priorities and scheduling
- Thread synchronization and locks
- Practical examples with multithreading

Great! Let's go step-by-step. Here's a detailed explanation of the **Thread Lifecycle**, **Thread Priorities**, **Thread Scheduling**, and **Thread Synchronization** — with diagrams and examples.

🧬 1. Thread Lifecycle (with Diagram)

A thread in Java goes through **5 main states**:

```sql
+--------+        start()       +-----------+
|  New   | -------------------> | Runnable  |
+--------+                      +-----------+
                                   |   |
                     Scheduled by|   | Running
                         CPU      v   |
                              +---------+
                              | Running |
                              +---------+
                                   |
        +--------------------------+----------------------------+
        |                           |                            |
        v                           v                            v
+-------------+          +----------------+              +--------------+
| Blocked     |          | Waiting        |              | Terminated   |
| (I/O wait)  |          | (e.g., join()) |              | (completed)  |
+-------------+          +----------------+              +--------------+
```

🔄 State Descriptions:

- **New**: Thread is created but not yet started.
- **Runnable**: After calling `start()`, thread is ready to run.
- **Running**: Thread is executing its `run()` method.
- **Blocked/Waiting**: Waiting for resource or another thread (e.g., file I/O or `join()`).
- **Terminated**: Thread has finished execution or was stopped.

🎚️ 2. Thread Priority and Scheduling

Java assigns **priorities** to threads (1 to 10):

Priority Constant	Value
`Thread.MIN_PRIORITY`	1
`Thread.NORM_PRIORITY`	5
`Thread.MAX_PRIORITY`	10

> Higher priority threads **might** get more CPU time, but **not guaranteed**.
> JVM uses **Thread Scheduler**, which depends on the OS.

🧪 Example:

java

```
Thread t1 = new Thread(() -> System.out.println("Thread 1"));
Thread t2 = new Thread(() -> System.out.println("Thread 2"));

t1.setPriority(Thread.MAX_PRIORITY); // 10
t2.setPriority(Thread.MIN_PRIORITY); // 1

t1.start();
t2.start();
```

🔒 3. Thread Synchronization

When **multiple threads** access **shared resources**, you must **synchronize** them to avoid data inconsistency.

🧪 Example (Without Synchronization - ❌):

java

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

✅ With Synchronization:

java

```
class Counter {
    int count = 0;

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

> **`synchronized`** ensures only **one thread** can access the method/block at a time.

🤹‍♀️ 4. Practical Multithreading Example

java

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.setName("Worker-1");
        t2.setName("Worker-2");

        t1.start();
        t2.start();
    }
}
```

🔎 Output (May vary):

```
python-repl


Worker-1 - 1
Worker-2 - 1
Worker-1 - 2
Worker-2 - 2
...
```

Would you like me to explain:

- **`join()`**, **`sleep()`**, **`yield()`**
- Deadlocks and how to prevent them
- Using **`ExecutorService`** instead of threads directly
- Or give you a mini-project using multithreading?

Awesome! Let's continue with these advanced multithreading topics — one by one:

⚙️ 1. Thread Methods: `join()`, `sleep()`, `yield()`

✅ `sleep(long ms)`

- Pauses the current thread for the given **milliseconds**.
- Doesn’t release any lock.

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

✅ `join()`

- Makes the **current thread wait** until the specified thread completes.

```java
Thread t1 = new Thread(() -> {
    System.out.println("Thread A running...");
});

t1.start();
t1.join();  // Main thread waits until t1 finishes
System.out.println("Main continues after thread A");
```

✅ `yield()`

- Suggests the current thread is **willing to pause**, giving a chance to other threads with same or higher priority.
- It’s **just a hint** to the scheduler.

```java
Thread.yield();  // No guarantee another thread will take over
```

🔁 2. Deadlocks and How to Avoid Them

❌ Deadlock: When two or more threads are waiting on each other’s locks, and none proceeds.

🧪 Example:

```java
class A {
    synchronized void methodA(B b) {
        System.out.println("Thread 1: Locked A");
        try { Thread.sleep(100); } catch (Exception e) {}
        b.last(); // waiting for lock on B
    }

    synchronized void last() {
        System.out.println("Inside A.last()");
    }
}

class B {
    synchronized void methodB(A a) {
        System.out.println("Thread 2: Locked B");
        try { Thread.sleep(100); } catch (Exception e) {}
        a.last(); // waiting for lock on A
    }

    synchronized void last() {
        System.out.println("Inside B.last()");
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();

        new Thread(() -> a.methodA(b)).start();
        new Thread(() -> b.methodB(a)).start();
    }
}
```

✅ Avoid Deadlock:

- Always **acquire locks in a fixed order**
- Use **`tryLock()`** from **`java.util.concurrent.locks.Lock`**
- Use **timeouts**

🚀 3. Using `ExecutorService` Instead of Raw Threads (Recommended)

Java provides a better thread management system via **Executors** (thread pools).

📌 Example:

```
java

import java.util.concurrent.*;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Runnable task1 = () -> System.out.println("Task 1 running");
        Runnable task2 = () -> System.out.println("Task 2 running");

        executor.submit(task1);
        executor.submit(task2);

        executor.shutdown(); // No more tasks will be accepted
    }
}
```

✅ Advantages:

- Better performance (reuses threads)
- Easier to scale and manage
- Supports scheduling and delayed execution

🛠️ 4. Mini Project: File Download Simulation using Multithreading

🗂️ Objective:

Simulate multiple file downloads (each in a separate thread).

✅ Code:

```java
java

class FileDownload extends Thread {
    private String fileName;

    FileDownload(String fileName) {
        this.fileName = fileName;
    }

    public void run() {
        System.out.println("Downloading: " + fileName);
        try {
            Thread.sleep(2000); // Simulate time to download
        } catch (InterruptedException e) {
            System.out.println("Download interrupted: " + fileName);
        }
        System.out.println("Downloaded: " + fileName);
    }

    public static void main(String[] args) {
        String[] files = {"file1.zip", "file2.zip", "file3.zip"};

        for (String file : files) {
            new FileDownload(file).start();  // Start each download in a new thread
        }
    }
}
```

🧪 Output (Parallel):

```makefile
makefile

Downloading: file1.zip
Downloading: file2.zip
Downloading: file3.zip
Downloaded: file2.zip
Downloaded: file1.zip
Downloaded: file3.zip
```

✅ What's Next?

Would you like:

- A real-world **multi-user chat app simulation** with threads?
- Deep dive into **`Callable`**, **`Future`**, **`ScheduledExecutorService`**?
- Thread-safe **collections** (**`ConcurrentHashMap`**, **`CopyOnWriteArrayList`**)?
- Interview-level multithreading scenarios and questions?

Let me know what you'd like to learn next!