**Thread and multithreading**

By: Shashi Prasad

We can create a thread in java **3 main ways:**

By extending the Thread class

By implementing the Runnable interface.

By implementing the Callable interface (with ExecutorService).

**What us Runnable Interface?**

The **Runnable interface** in Java is one of the simplest and most important interfaces used for creating threads or defining tasks that can be executed by a thread. It’s a **functional interface**, meaning it has only **one abstract method** → run(). It is present in the java.lang package.

It represents a **task** that can be executed by a Thread or an executor service.

How to use :

class MyRunnable **implements Runnable** {

public void **run()** {

System.out.println("Thread running via Runnable interface");

}

public static void main(String[] args) {

Thread **t1** = new Thread(**new MyRunnable()**);

Thread **t2** = new Thread(new MyRunnable());

**t1**.start(); // starts a new thread that calls run()

**t2**.start(); // starts a new thread that calls run()

}

}

Using Lambda Expression (Java 8+)

public class RunnableLambda {

public static void main(String[] args) {

**Runnable task = () -> System.out.println("Running with lambda!");**

Thread t1 = new Thread(task);

Thread t2 = new Thread(task);

T1.start();

T2.start();

}

}

Runnable the foundation of multithreading in Java.

**What is callable?**

The Callable interface represents a task that can run in a separate thread, return a result, and throw exceptions — used with concurrency utilities like ExecutorService and Future. The **Callable** interface in Java is part of the **java.util.concurrent** package You **cannot** directly pass a Callable to a Thread. Instead, use an **ExecutorService** to run it.

**What is Thread class?**

Thread is part of the **java.lang** package. It implements **Runnable** interface. Java 21, Thread supports both **platform threads** (the “classic” threads tied to OS-level threads) and **virtual threads** (lightweight threads introduced in recent Java versions). Here A virtual thread is also an instance of Thread, but it is not permanently tied to a specific OS thread.

**Key Responsibilities of Thread class.**

Start a thread,

Define what to execute by Override run() method.

Control execution by using sleep(), join(), yield(), and interrupt()

Thread identity by using getName(), getId(), setName()

Thread priority by using getPriority(), setPriority()

Daemon threads by using setDaemon(true) makes background threads that stop when main thread ends

Thread state Tracks states via Thread.State (NEW, RUNNABLE, BLOCKED, etc.)

Synchronization Works with synchronized blocks to ensure thread safety

**When to use thread class and when to use runnable interface?**

The choice between Thread class and Runnable interface depends on your design needs

1 - If want to stop inheritance on class then Must extend Thread class, so you can’t extend any other class other wise use Runnable because it allow to extend other classes as well.

2 – Reusability, when we extend thread class and implements task as run method then Task logic and thread control are mixed together. Which may create reusability issue for the class. BUT when we create thread using Runnable interface Task logic and thread control are separatedso it maintains class reusable**.**

3- Less flexible (tight coupling) when we extend thread class. But with Runnable it is More flexible (loose coupling)

4 Use Thread class only when: **You need to override Thread methods** - If you want to customize methods like interrupt(), join(), or other Thread-specific behaviour

5- **Use Runnable in most cases** because its provide Inheritance flexibility, **Better design** - Separates the task from the thread mechanism, **Thread pool compatibility** - Works seamlessly with ExecutorService and thread pools, Same Runnable can be passed to multiple threads, Doesn't carry unnecessary Thread class overhead.

*Go Default to Runnable interface unless you have a specific reason to extend Thread.*

**What is the Java Memory Model (JMM)?**

The Java Memory Model (JMM) defines how Java threads interact through memory — specifically: *It specifies how and when changes made by one thread become visible to others and how the JVM should order reads and writes to shared variables.*

Without the JMM, threads could see inconsistent or stale data, especially when multiple CPU cores and caches are involved.

As we know Each thread may **cache variables locally** (in CPU registers or caches). Updates to shared variables may not immediately reflect in main memory. Therefore, another thread may see **outdated values**.

JMM ensures consistency between these caches and main memory.

Main Memory vs. Working Memory: In Java's memory model, understanding the difference between main memory and working memory is crucial for writing correct multithreaded code.

**Main Memory** is the shared memory space where: All objects and their instance variables are stored (heap), Static variables reside, all threads can access this shared space, It's the "single source of truth" for variable values

**Working Memory** (Thread Cache): Working Memory is the local cache for each thread: Each thread has its own private working memory (CPU cache/registers) , Threads copy variables from main memory to their working memory, threads work on these local copies for performance,

Changes are eventually written back to main memory.

To deal with JMM we use Volatile, atomic variable, concurrent collection.

**What is Volatile? When to use it?**

It guarantees visibility of changes across threads by establishing a "**happens-before**" relationship, though it doesn't provide full atomicity or mutual exclusion. So its characteristics are:

1. Guarantees **visibility of changes** across threads
2. Prevents thread caching of the variable's value
3. Ensures reads/writes go directly to main memory
4. Provides happens-before guarantee (changes by one thread are visible to others). **Happens-before** is a fundamental concept in Java's memory model that defines the ordering and visibility guarantees between operations in a multithreaded program.

If action A **happens-before** action B, then:

* The memory effects of A are visible to B
* A is guaranteed to execute before B (from B's perspective)
* Any writes done by A will be seen by B

**Without happens-before guarantees**, the JVM can:

* Reorder instructions for optimization
* Cache variables in CPU registers/caches
* Result in one thread not seeing changes made by another thread

So volatile indicate to JVM that does not apply any optimization and caching on variable.

It only guarantees for visibility. **But** if you to apply They provide happens-before guarantees on operation between threads then use **synchronization**

**When to Use Volatile:** Simple flags between threads (Java), Read-mostly scenarios One thread writes, multiple threads read (Java).

**Don't use volatile when**: Performance critical code, Multiple related variables, Complex synchronization like multiple read write by many threads.

**What Are Atomic Variables in Java?**

Atomic variables in Java are special classes provided in the **java.util.concurrent.atomic** package that ensure thread-safe operations on single variables without the need for explicit locks (like **synchronized** blocks). They achieve this through lock-free programming using **hardware-level primitives** such as **Compare-And-Swap (CAS) operations**, which atomically read a value, compare it to an expected value, and update it if they match—all in a single, uninterruptible CPU instruction.

Atomic Classes are: **AtomicInteger, AtomicLong, AtomicBoolean ,AtomicReference** for references to objects.

Atomic operations are guaranteed to be visible across threads (similar to volatile) and indivisible. It have methods like

1. get(): Reads the current value.
2. set(int newValue): Sets the value.
3. compareAndSet(int expect, int update): Updates only if the current value matches expect.
4. incrementAndGet(): Atomically increments and returns the new value.

END