**Virtual Threads in Java 21**

Virtual threads introduce an abstraction layer between operating-system processes and application-level concurrency. At a high level, a thread is managed and scheduled by the operating system, while a virtual thread is managed and scheduled by a virtual machine.

**What is a Platform Thread?**

A platform thread runs Java code on its underlying OS thread. The number of available platform threads is limited to the number of OS threads. Platform threads typically have a large thread stack and other resources that are maintained by the operating system.

**What is a Virtual Thread?**

Like a platform thread, a virtual thread is also an instance of java.lang.Thread. However, a virtual thread isn't tied to a specific OS thread. A virtual thread still runs code on an OS thread.

**Why Use Virtual Threads?**

* Since Virtual threads are managed within the JVM hence, they are quite efficient in both memory and CPU.
* With virtual threads, millions of tasks can be run simultaneously on common computer hardware.
* Use virtual threads only for code that involves blocking, such logging, file I/O, accessing databases, network calls.
* Use virtual threads in high-throughput concurrent applications, especially those that consist of a great number of concurrent tasks that spend much of their time waiting. Server applications are examples of high-throughput applications.
* Virtual threads are not faster threads; they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency).

**Scheduling Virtual Threads and Pinned Virtual Threads**

The operating system schedules when a platform thread is run. When the Java runtime schedules a virtual thread, it assigns or mounts the virtual thread on a platform thread, then the operating system schedules that platform thread as usual. This platform thread is called a carrier. After running some code, the virtual thread can unmount from its carrier. After a virtual thread unmounts from its carrier, the carrier is free, which means that the Java runtime scheduler can mount a different virtual thread on it. A virtual thread cannot be unmounted during blocking operations when it is pinned to its carrier. A virtual thread is pinned in the following situations:

**Advantages of Java virtual threads**

1. Improves application availability

2. Improves application throughput

3. Reduces ‘OutOfMemoryError: unable to create new native thread’

4. Reduces application memory consumption

5. Improves code quality

6. 100% compatible with Platform Threads

* **Platform thread**: The Java wrapper for an Operating System (OS) thread that is scheduled by the thread scheduler of the OS.
* **Virtual Thread**: A lightweight abstraction of a task that can be bound (called "mounting" in Java 21) to a platform thread and is scheduled by the Java virtual thread scheduler.
* **Carrier thread**: The platform thread on which a virtual thread is mounted.

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.ThreadFactory;

**import** java.util.concurrent.TimeUnit;

**public** **class** TestBasicVirtualThread1 {

**public** **void** m1(**int** timeInSec) {

System.***out***.println("Executing m1() ....");

**try** {

TimeUnit.***SECONDS***.sleep(timeInSec);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Execution m1() completed ....");

}

**public** **void** execute1() {

Runnable r1 = () -> m1(5);

ThreadFactory virtualThreadFactory = Thread.*ofVirtual*().factory();

ThreadFactory kernelThreadFactory = Thread.*ofPlatform*().factory(); //kernel or platform

Thread virtualThread = virtualThreadFactory.newThread(r1);

Thread kernelThread = kernelThreadFactory.newThread(r1);

virtualThread.start();

kernelThread.start();

// Platform thread

(**new** Thread(r1)).start();

Thread platformThread = **new** Thread(r1);

platformThread.start();

// Virtual thread

Thread virtualThread1 = Thread.*startVirtualThread*(r1);

Thread ofVirtualThread = Thread.*ofVirtual*().start(r1);

// Virtual thread created with a factory

ThreadFactory factory = Thread.*ofVirtual*().factory();

Thread virtualThreadFromAFactory = factory.newThread(r1);

virtualThreadFromAFactory.start();

}

**public** **void** execute2() {

Runnable r1 = () -> m1(3);

Thread virtualThread = Thread.*startVirtualThread*(r1);

**try** {

virtualThread.join();

System.***out***.println("Task completed ...");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**public** **void** execute3() {

Runnable r1 = () -> m1(3);

**try** {

Thread.*ofVirtual*().name("my-virtual-thread").start(r1).join();

System.***out***.println("Task completed ...");

} **catch** (InterruptedException e) {

e.printStackTrace();

};

}

**public** **void** execute4() {

// Using ExecutorService

Runnable r1 = () -> m1(3);

**try** (ExecutorService executorService = Executors.*newVirtualThreadPerTaskExecutor*()) {

**for** (**int** i = 0; i < 5; i++) {

executorService.submit(r1);

}

}

}

**public** **void** check() {

// execute2();

// execute3();

// execute4();

}

**public** **static** **void** main(String[] args) {

**new** TestBasicVirtualThread1().check();

}

}

**Precise Answer from ChatGPT**

In Java, traditionally we use **platform threads**, which are **mapped 1:1 to OS threads**.  
Each thread consumes significant memory (stack ~1MB) and involves expensive context switching.  
When you create thousands of threads (e.g., for I/O-heavy applications), it becomes inefficient.

**Virtual Threads** solve this by:

* Being **lightweight** (created and managed by the JVM, not the OS).
* Using a **small footprint** (thousands or even millions can run concurrently).
* Automatically **yielding control** when blocked on I/O or waiting for locks.
* Implemented as **user-mode threads** that are scheduled by the JVM on top of a small number of OS threads.

In short:

Virtual Threads = Threads for high concurrency, low cost.

**Key Benefits**

| **Feature** | **Platform Thread** | **Virtual Thread** |
| --- | --- | --- |
| Backed by | OS Thread | JVM (user-space) |
| Memory Usage | ~1 MB per thread | ~ few KB per thread |
| Count | Thousands | Millions |
| Suitable for | CPU-bound tasks | I/O-bound tasks (DB, API calls, etc.) |
| Scheduling | OS | JVM |

Examples are given below

**Simple Virtual Thread**

public class VirtualThreadExample {

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

Thread vt = Thread.ofVirtual().start(() -> {

System.out.println("Running in virtual thread: " + Thread.currentThread());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

throw new RuntimeException(e);

}

});

vt.join();

System.out.println("Main thread done!");

}

}

**Launching Many Virtual Threads**

Let’s see how virtual threads handle massive concurrency:

import java.util.concurrent.\*;

public class VirtualThreadManyExample {

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

ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();

for (int i = 1; i <= 10\_000; i++) {

int taskId = i;

executor.submit(() -> {

System.out.println("Task " + taskId + " running on " + Thread.currentThread());

try {

Thread.sleep(1000); // Simulate I/O

} catch (InterruptedException e) {

throw new RuntimeException(e);

}

return taskId;

});

}

executor.shutdown();

executor.awaitTermination(5, TimeUnit.SECONDS);

System.out.println("All tasks completed!");

}

}

**Explanation:**

* Creates **10,000 virtual threads** (which would be impossible with platform threads).
* Each thread does a simulated I/O task.
* JVM efficiently schedules them using a small pool of carrier threads (OS threads).

**Real-World Use Case**

Virtual threads are great for:

* Web servers (e.g., Spring Boot 3.2+ can use them).
* Handling database or network calls concurrently.
* Replacing complex asynchronous (CompletableFuture) code with simple blocking code — but still scalable.