Virtual Threads vs Regular Threads

Welcome, students! This guide will help you understand one of the most exciting new features in Java: **Virtual Threads**. We'll start with the basics and use simple analogies to see why this is a game-changer.

What Is a Thread?

Think of your computer program (like a game or a website) as a big project, like building a house.

A **thread** is like a single **worker** on that project.

- If you have only **one worker** (a single thread), they have to do everything one by one: lay the foundation, then build the walls, then put on the roof. It's slow.
- If you have **many workers** (multi-threading), you can have one group laying the foundation while another group cuts wood for the walls. Your program can do multiple things at the same time, making it much faster.

The Two Types of Java Threads: A Tale of Two Workers

For a long time, Java only had one kind of thread. Now, with Java 21, we have two.

1. Regular Threads (Platform Threads)

A **regular thread** (now called a **platform thread**) is like hiring a **full-time, highly-skilled worker** (like a master carpenter).

- **"Heavyweight":** It takes a lot of time and resources to hire them. In computer terms, this means each thread takes up a significant amount of your computer's memory (RAM).
- **Managed by the OS:** This worker reports directly to the big boss—the computer's **Operating System (OS)**. The OS manages their schedule and has to keep track of them.
- **Limited Supply:** You can't hire millions of these workers. Your OS can only handle a few thousand at most before it gets overwhelmed and your system slows to a crawl.

2. Virtual Threads (The New Way)

A **virtual thread** is like hiring a team of **temporary, lightweight helpers**.

- **"Lightweight":** You can "hire" one in an instant with almost no resources. They take up a tiny, tiny amount of memory.

- **Managed by Java (JVM):** These helpers don't report to the OS. They report to a Java-based manager (the **Java Virtual Machine, or JVM**). This manager has a *few* of those heavy-duty regular threads and cleverly assigns tasks to the helpers.
- **Massive Supply:** You can easily have *millions* of virtual threads. The JVM manager is brilliant at juggling all of them.

A Key Concept: "Blocking" (Waiting)

This is the most important reason virtual threads exist. Let's go back to our worker analogy.

Blocking (The Old Way)

Imagine you ask your expensive, regular worker (a **platform thread**) to go get some wood from a supplier across town. This task involves a lot of *waiting*. They have to drive, wait in traffic, wait in line at the store, and drive back.

While they are doing all this waiting (this is called **"Blocking I/O"**), they are **blocked**. They can't do any other work. They are just sitting in the truck, stuck in traffic. This is a huge waste of a skilled, expensive worker. Your "worker slot" is totally occupied.

Non-Blocking (The Virtual Thread Way)

Now, imagine you ask a lightweight helper (a **virtual thread**) to do the same task.

The *moment* the helper gets in the truck to go to the supplier (starts waiting), the JVM manager says, "Okay, you're waiting. **Step aside.**"

The manager then takes the heavy-duty thread that helper *was* using and gives it to a *different* helper who is ready to do real work (e.g., hammering a nail).

When the first helper gets the wood and returns, they tell the manager, "I'm back!" The manager then finds an available heavy-duty thread to let them continue their *next* task.

The Big Idea: With regular threads, waiting for data (like from a database or a web page) *wastes the whole thread*. With virtual threads, waiting doesn't waste anything. The helper just "steps aside" and lets another helper run.

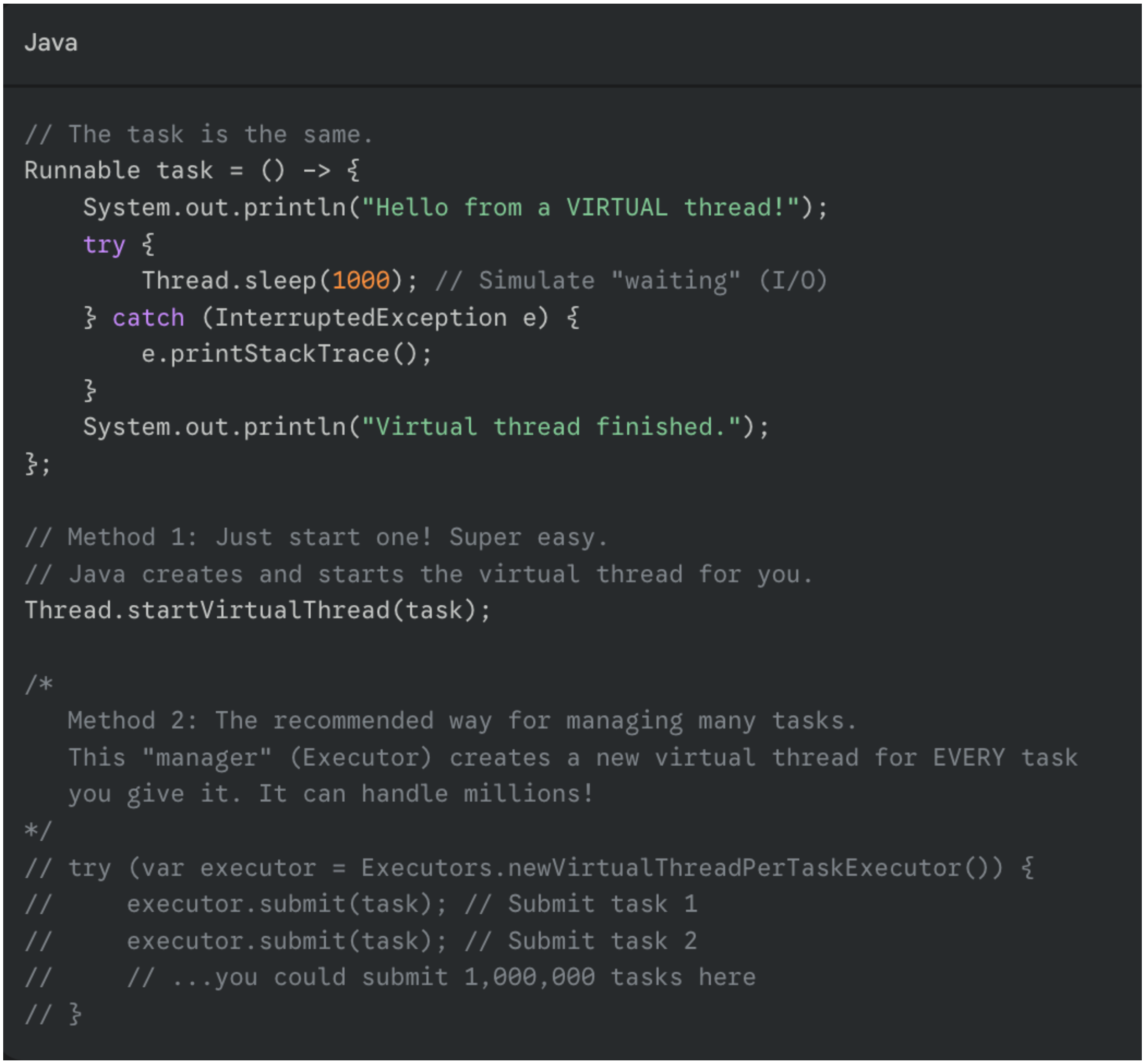
Code Examples: Seeing is Believing

Here is how you would write the code for both.

Example 1: Creating a Regular (Platform) Thread

This is the "old" way. It's still useful, but you can see it's a bit more work.

Java



Java

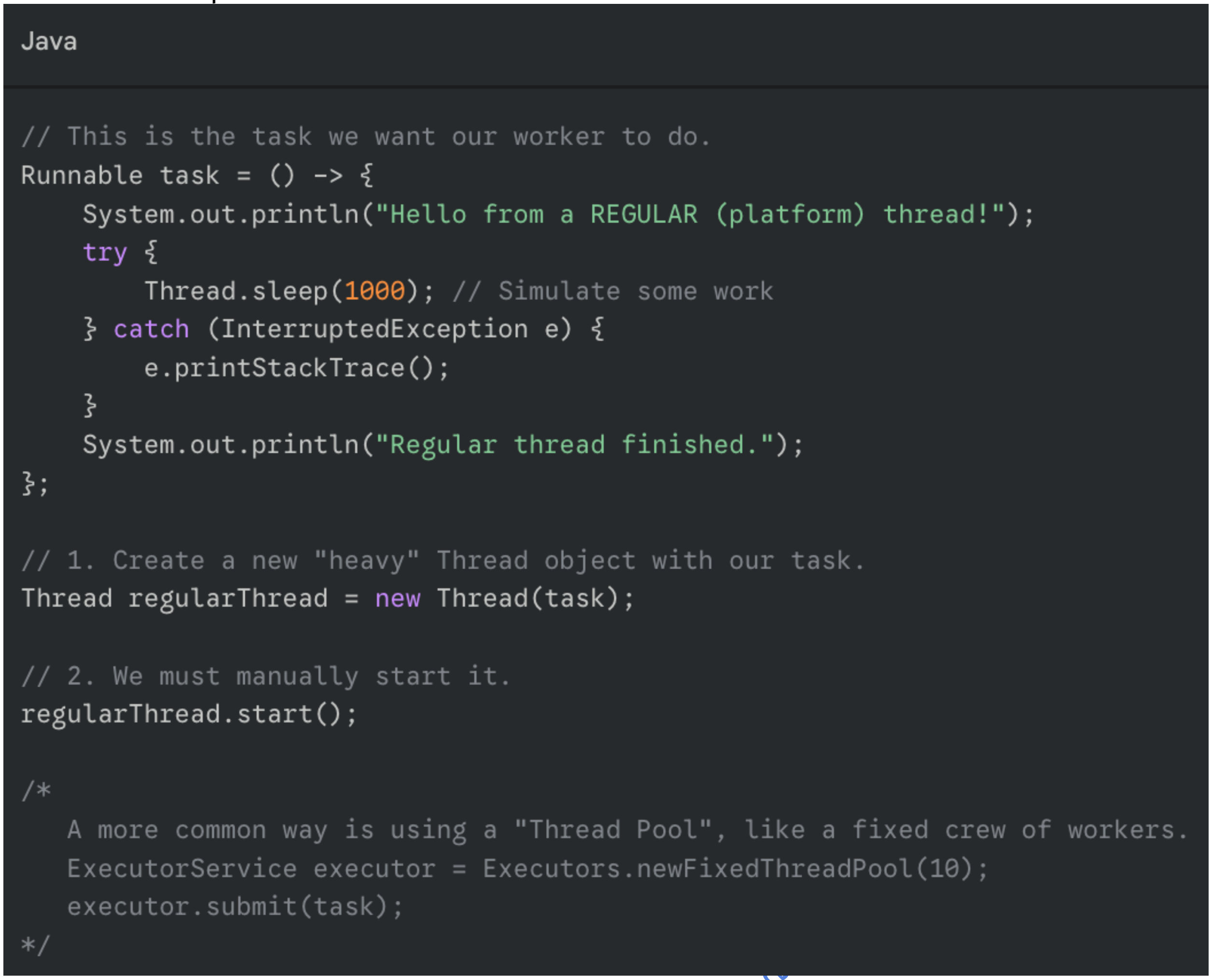
```java
// The task is the same.
Runnable task = () -> {
    System.out.println("Hello from a VIRTUAL thread!");
    try {
        Thread.sleep(1000); // Simulate "waiting" (I/O)
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Virtual thread finished.");
};

// Method 1: Just start one! Super easy.
// Java creates and starts the virtual thread for you.
Thread.startVirtualThread(task);

/*
   Method 2: The recommended way for managing many tasks.
   This "manager" (Executor) creates a new virtual thread for EVERY task
   you give it. It can handle millions!
*/
// try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
//     executor.submit(task); // Submit task 1
//     executor.submit(task); // Submit task 2
//     // ...you could submit 1,000,000 tasks here
// }
```

Example 2: Creating a Virtual Thread (The "New" Way)

This is much simpler and more efficient for most tasks.



Java

```java
// This is the task we want our worker to do.
Runnable task = () -> {
    System.out.println("Hello from a REGULAR (platform) thread!");
    try {
        Thread.sleep(1000); // Simulate some work
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Regular thread finished.");
};

// 1. Create a new "heavy" Thread object with our task.
Thread regularThread = new Thread(task);

// 2. We must manually start it.
regularThread.start();

/*
   A more common way is using a "Thread Pool", like a fixed crew of workers.
   ExecutorService executor = Executors.newFixedThreadPool(10);
   executor.submit(task);
*/
```

Comparison: Regular vs. Virtual Threads

Feature	Regular (Platform) Threads	Virtual Threads
Analogy	Heavy-duty, full-time worker	Lightweight, temporary helper
Managed By	The Operating System (OS)	The Java Virtual Machine (JVM)
Creation Cost	High (Slow, uses lots of memory)	Very Low (Fast, uses tiny memory)
How Many?	Hundreds or Thousands	Millions
When Waiting?	The *entire* thread is blocked (stuck).	The thread "steps aside" (unmounts).
Best For...	**CPU-bound** (heavy calculations)	**I/O-bound** (waiting for data)

When Should I Use Which?

This is the most important question for a developer.

👉 Use VIRTUAL Threads when...

Your code spends *most of its time waiting*. This is extremely common in modern apps.

- Waiting for a database to return data.
- Waiting for a web page (API) to send a response.
- Waiting for a file to be read from a disk.

This is called **"I/O-bound"**. You have *many tasks* that do a *little bit of work* and a *lot of waiting*.

👉 Use REGULAR (Platform) Threads when...

Your code is doing *constant, heavy calculation* and is not waiting for anything.

- Running a complex math algorithm.
- Compressing a large video file.
- Training a machine learning model.

This is called **"CPU-bound"**. You have a *few tasks* that need 100% of the CPU's attention. A virtual thread doesn't help here, because the worker is never "stuck" or "waiting"—it's just busy working.

Summary & Key Takeaways

- A **thread** is a "worker" in your program that can run tasks.
- **Regular threads** are "heavyweight" workers managed by the OS. They are limited in number and get "blocked" (stuck) when waiting for data.
- **Virtual threads** are "lightweight" helpers managed by Java. You can create millions of them, and they *don't get blocked* when waiting.
- **The Golden Rule:** Use **virtual threads** for tasks that *wait a lot* (I/O-bound). Use **regular threads** for tasks that *think a lot* (CPU-bound).
- For most web servers and microservices, virtual threads are a massive improvement, allowing one server to handle many more users at the same time.