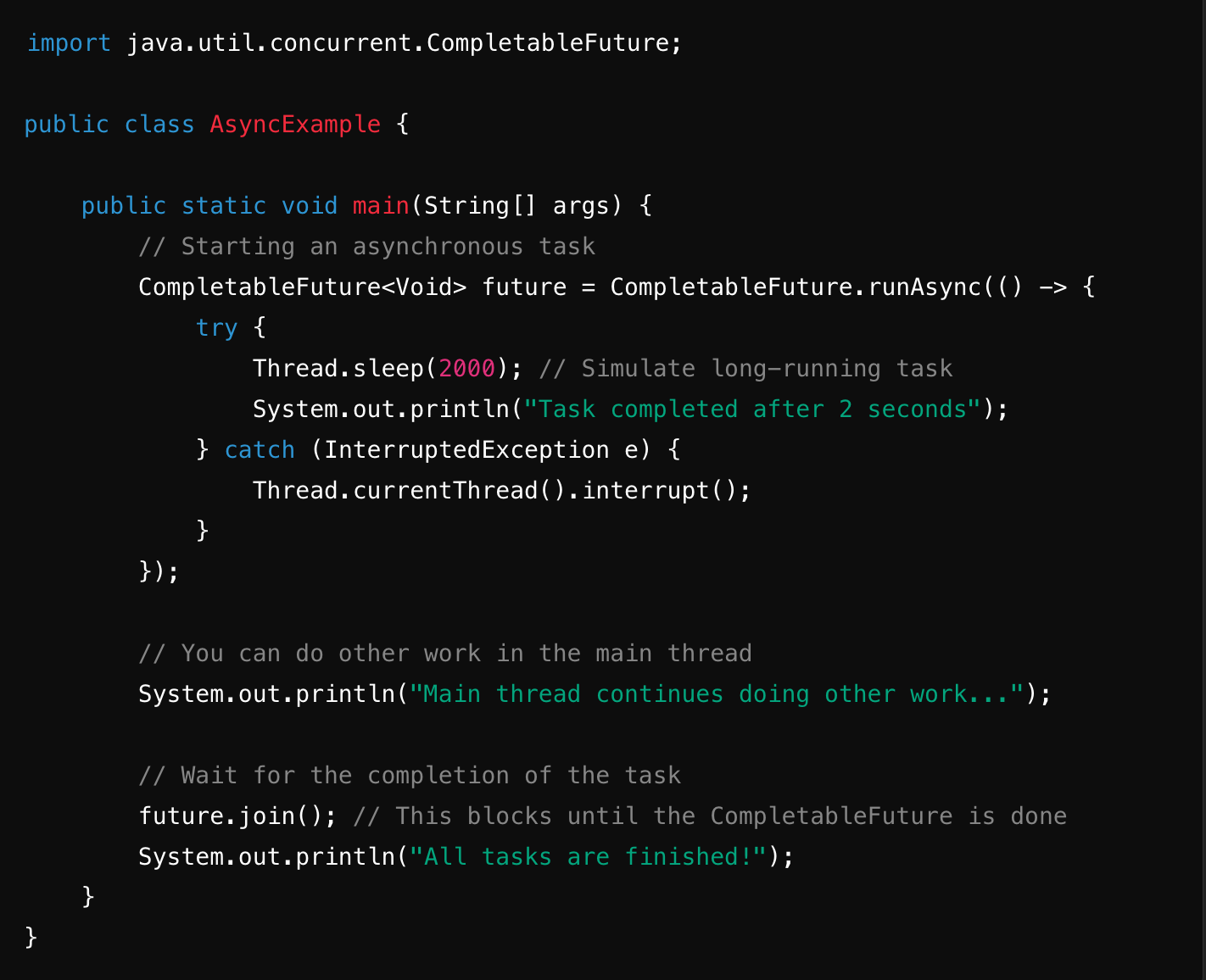
In Java, writing asynchronous functions typically involves leveraging **concurrency** and **parallelism** mechanisms such as threads, CompletableFuture, or the ExecutorService. Java doesn't have native syntax like async/await as seen in JavaScript or Python, but you can still achieve asynchronous behavior using these tools.

Here's a breakdown of how to write asynchronous functions in Java using a few common approaches:

### 1. ****Using**** CompletableFuture ****(Java 8 and above)****

CompletableFuture provides a more modern and flexible way to handle asynchronous tasks in Java, allowing you to write non-blocking code. It is part of the java.util.concurrent package.

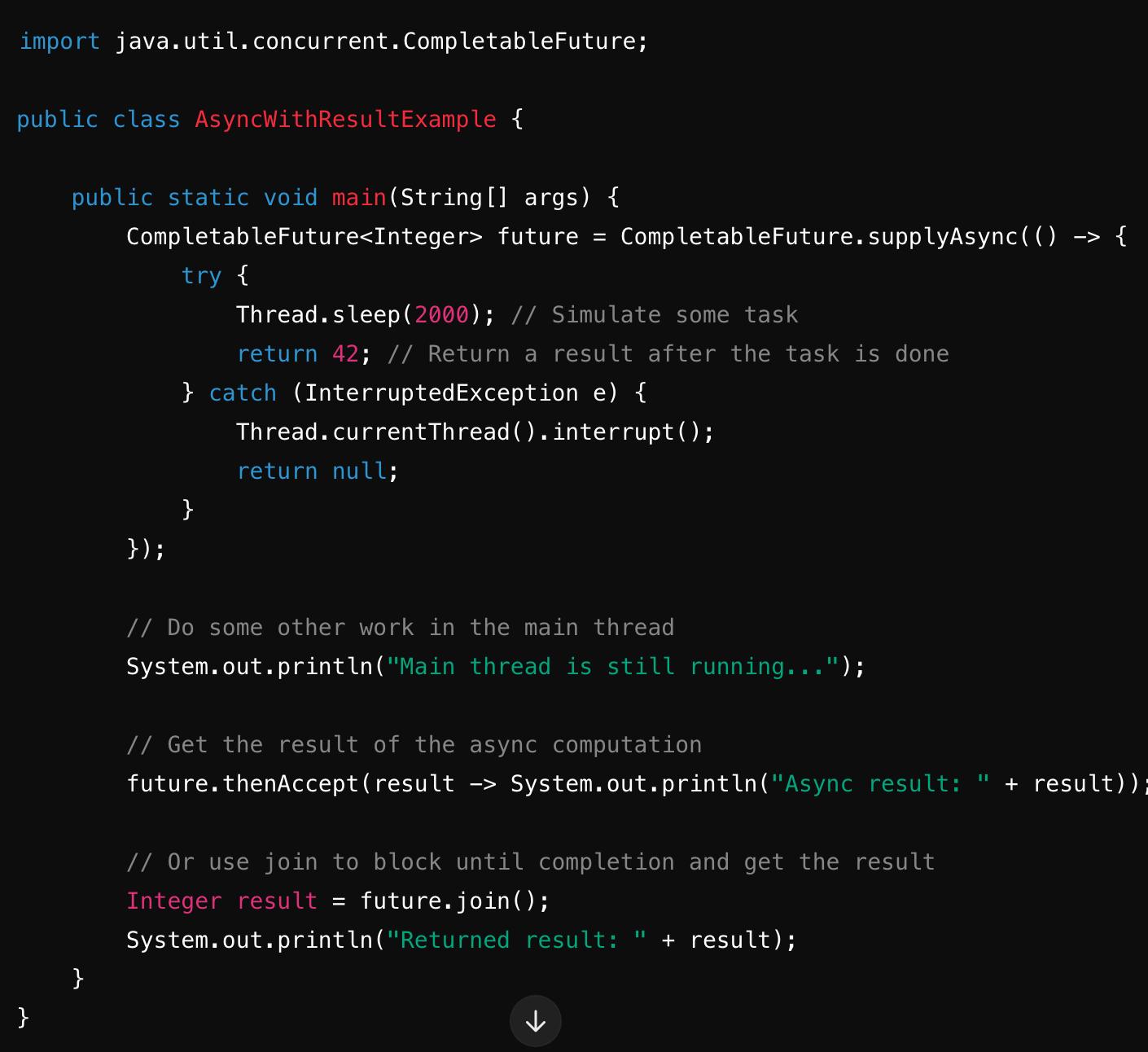
#### Basic Example with CompletableFuture



* runAsync() starts a task that doesn't return a result (void).
* join() is used to block the main thread until the CompletableFuture task completes.

#### Example with supplyAsync (Returning a Result)

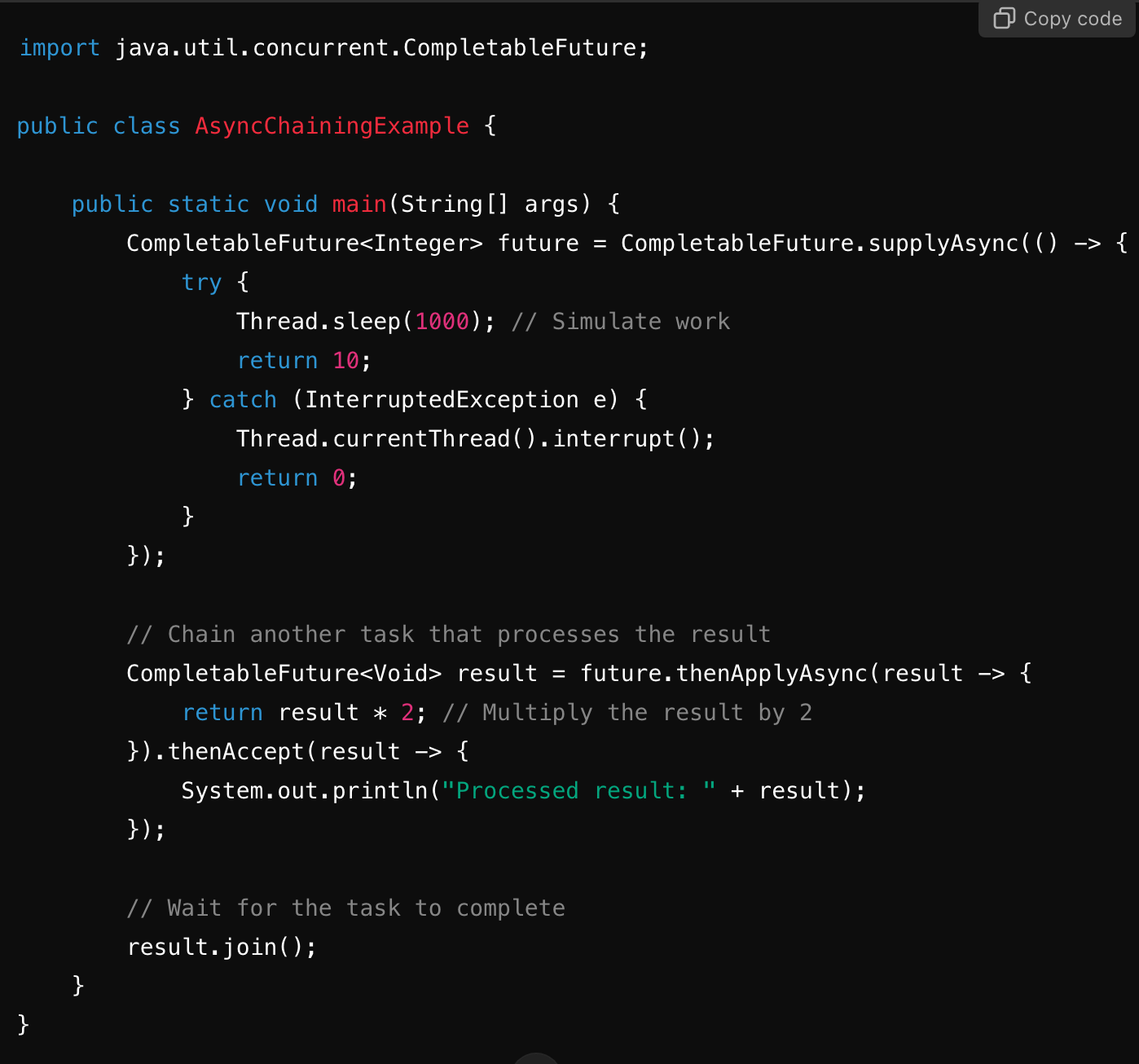
If you want the asynchronous task to return a value, use supplyAsync().



* supplyAsync() is used when you need to return a result from the asynchronous operation.
* thenAccept() is used to process the result asynchronously when it completes.

#### Chaining Asynchronous Calls

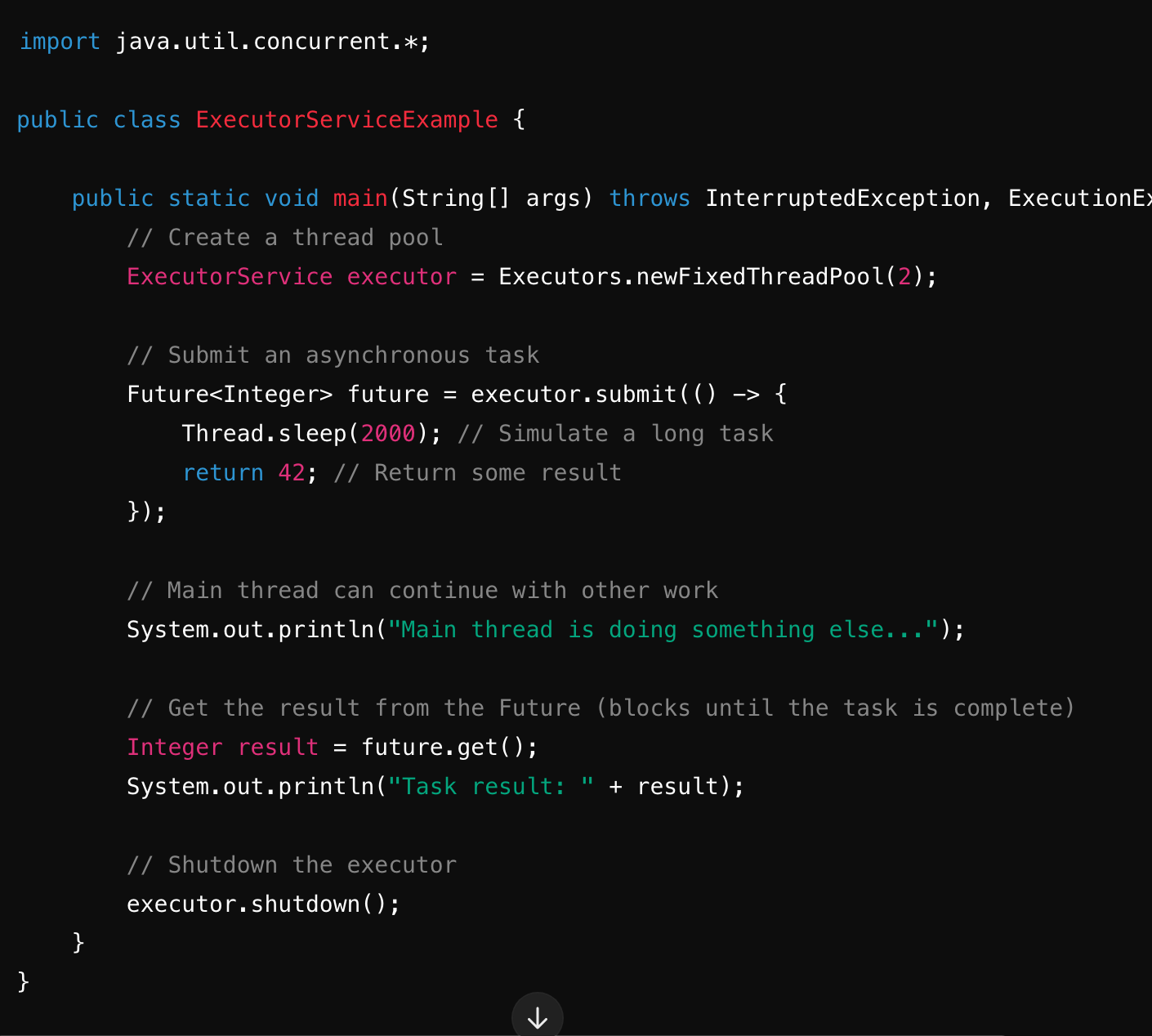
You can chain multiple asynchronous tasks with methods like thenApply(), thenAccept(), etc., for more complex workflows.



### 2. ****Using**** ExecutorService

Another way to execute asynchronous tasks is by using an ExecutorService, which provides thread pool management for running tasks concurrently.

Example using ExecutorService



* submit() submits a task for execution and returns a Future object, which can be used to retrieve the result.
* future.get() blocks until the asynchronous task completes and retrieves the result.

### 3. ****Using**** Thread ****Class (Manual Threads)****

You can also manually manage threads, though it is more low-level and not as recommended in modern Java unless you need explicit control over threads.

#### Basic Example with Thread Class

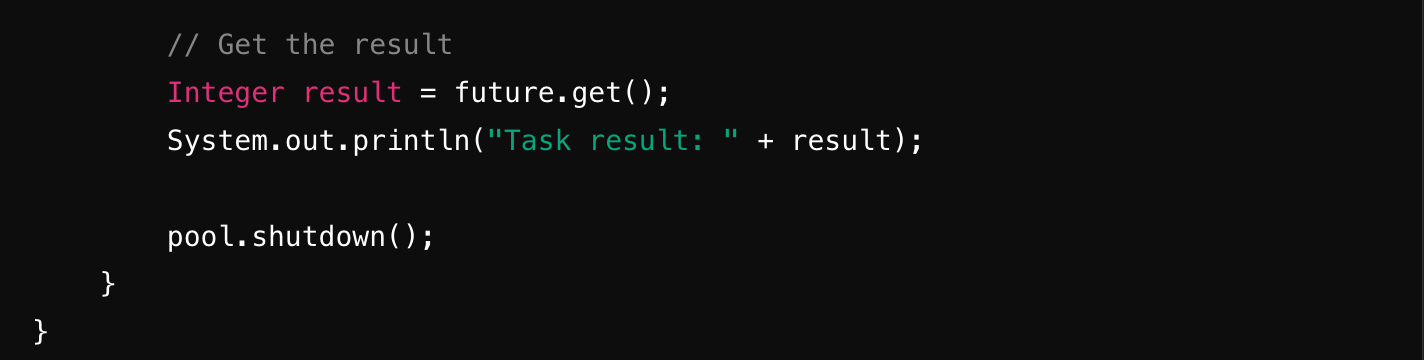
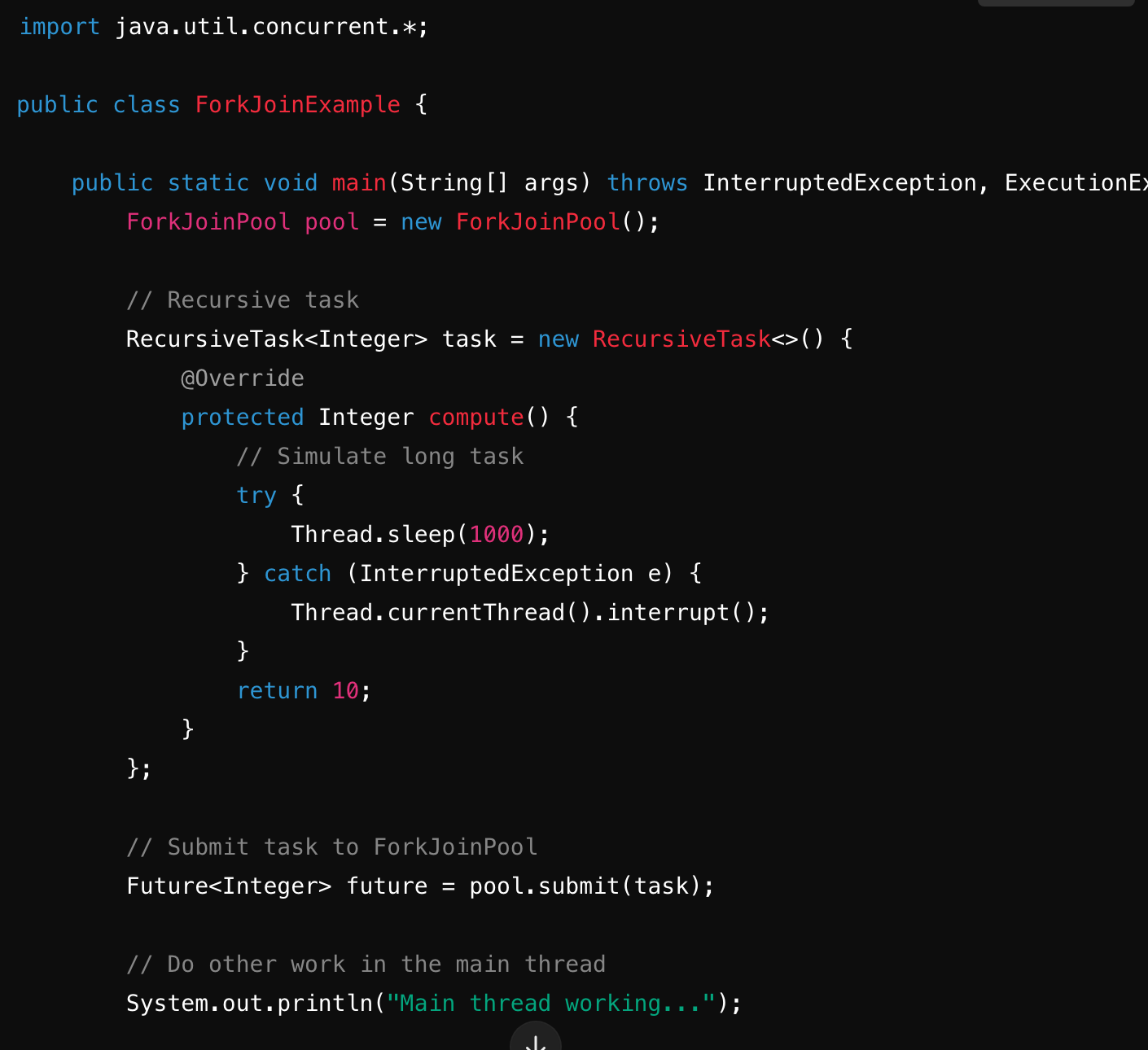


* start() is used to begin execution of the new thread.
* join() is used to block the main thread until the new thread completes.

### 4. ****Using**** ForkJoinPool ****for Parallel Tasks****

If you're dealing with large tasks that can be split into smaller sub-tasks, you can use ForkJoinPool for parallel execution.

#### Example with ForkJoinPool



### Key Takeaways:

1. CompletableFuture: Provides modern ways to write asynchronous code in a more functional style.
2. ExecutorService: Manages a pool of threads and allows you to submit tasks asynchronously.
3. Thread: Basic manual thread management for more explicit control.
4. ForkJoinPool: Used for splitting large tasks into smaller parallel tasks.

Each of these approaches has its use cases, and the choice of which to use depends on the complexity and nature of your asynchronous tasks. For most modern Java applications, CompletableFuture is generally the recommended approach as it provides better abstraction and composition capabilities.