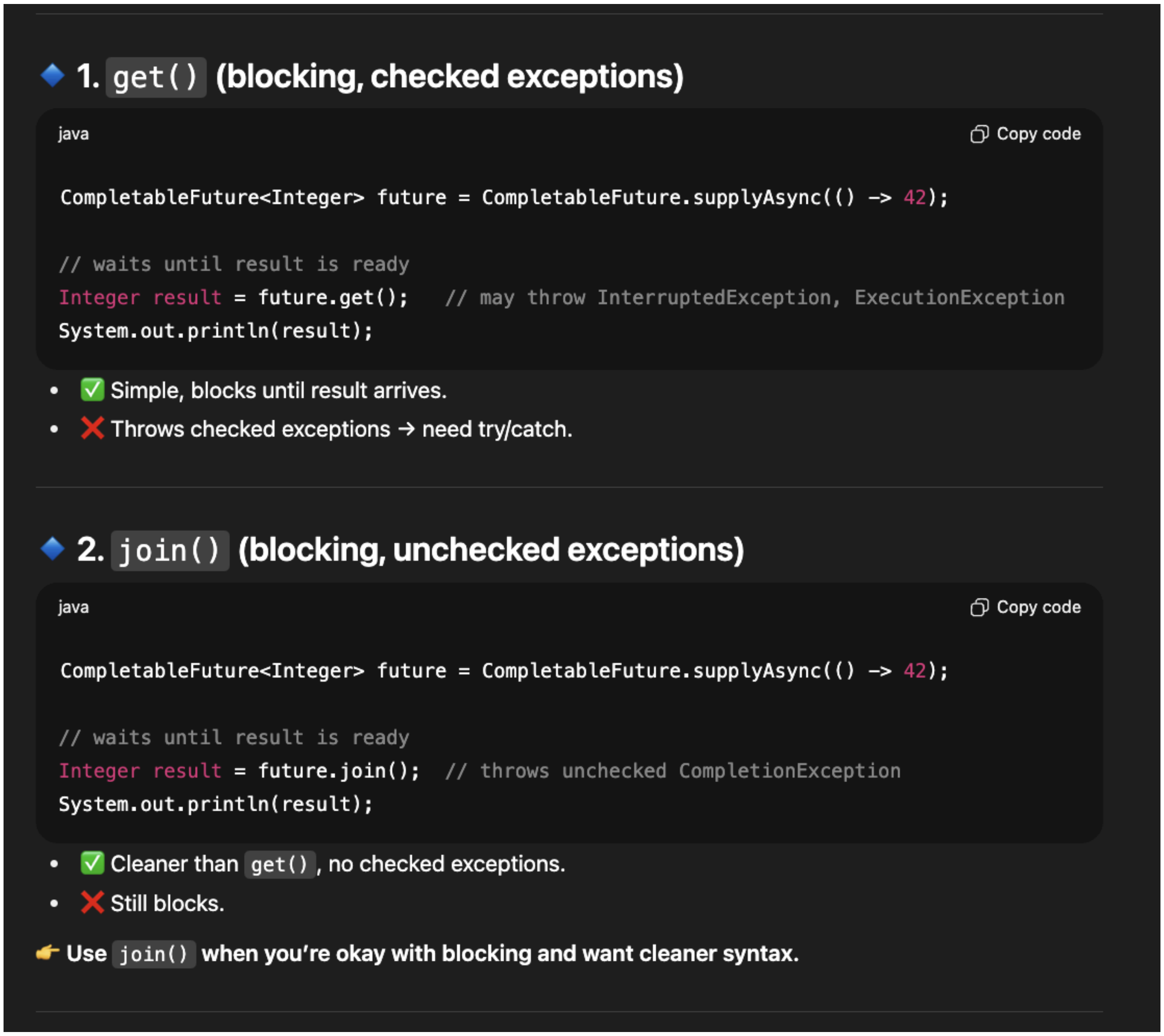


🔷 1. `get()` (blocking, checked exceptions)

java

```java
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 42);

// waits until result is ready
Integer result = future.get();   // may throw InterruptedException, ExecutionException
System.out.println(result);
```

- ✅ Simple, blocks until result arrives.
- ❌ Throws checked exceptions → need try/catch.

🔷 2. `join()` (blocking, unchecked exceptions)

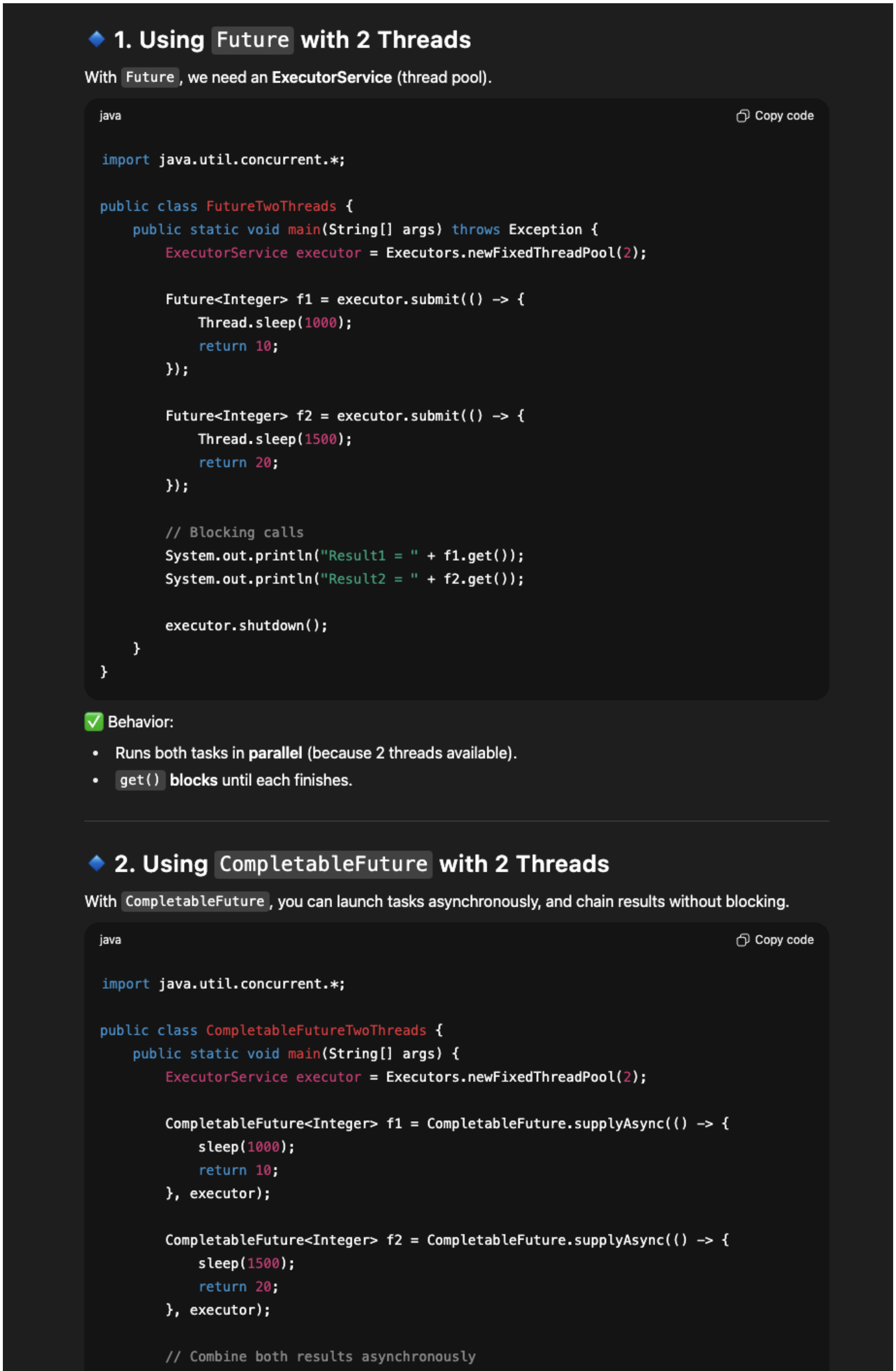
java

```java
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 42);

// waits until result is ready
Integer result = future.join();  // throws unchecked CompletionException
System.out.println(result);
```

- ✅ Cleaner than `get()`, no checked exceptions.
- ❌ Still blocks.

👉 **Use `join()` when you're okay with blocking and want cleaner syntax.**



🔹 1. Using `Future` with 2 Threads

With `Future`, we need an **ExecutorService** (thread pool).

java

```java
import java.util.concurrent.*;

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

        Future<Integer> f1 = executor.submit(() -> {
            Thread.sleep(1000);
            return 10;
        });

        Future<Integer> f2 = executor.submit(() -> {
            Thread.sleep(1500);
            return 20;
        });

        // Blocking calls
        System.out.println("Result1 = " + f1.get());
        System.out.println("Result2 = " + f2.get());

        executor.shutdown();
    }
}
```

✅ Behavior:

- Runs both tasks in **parallel** (because 2 threads available).
- `get()` **blocks** until each finishes.

🔹 2. Using `CompletableFuture` with 2 Threads

With `CompletableFuture`, you can launch tasks asynchronously, and chain results without blocking.

java

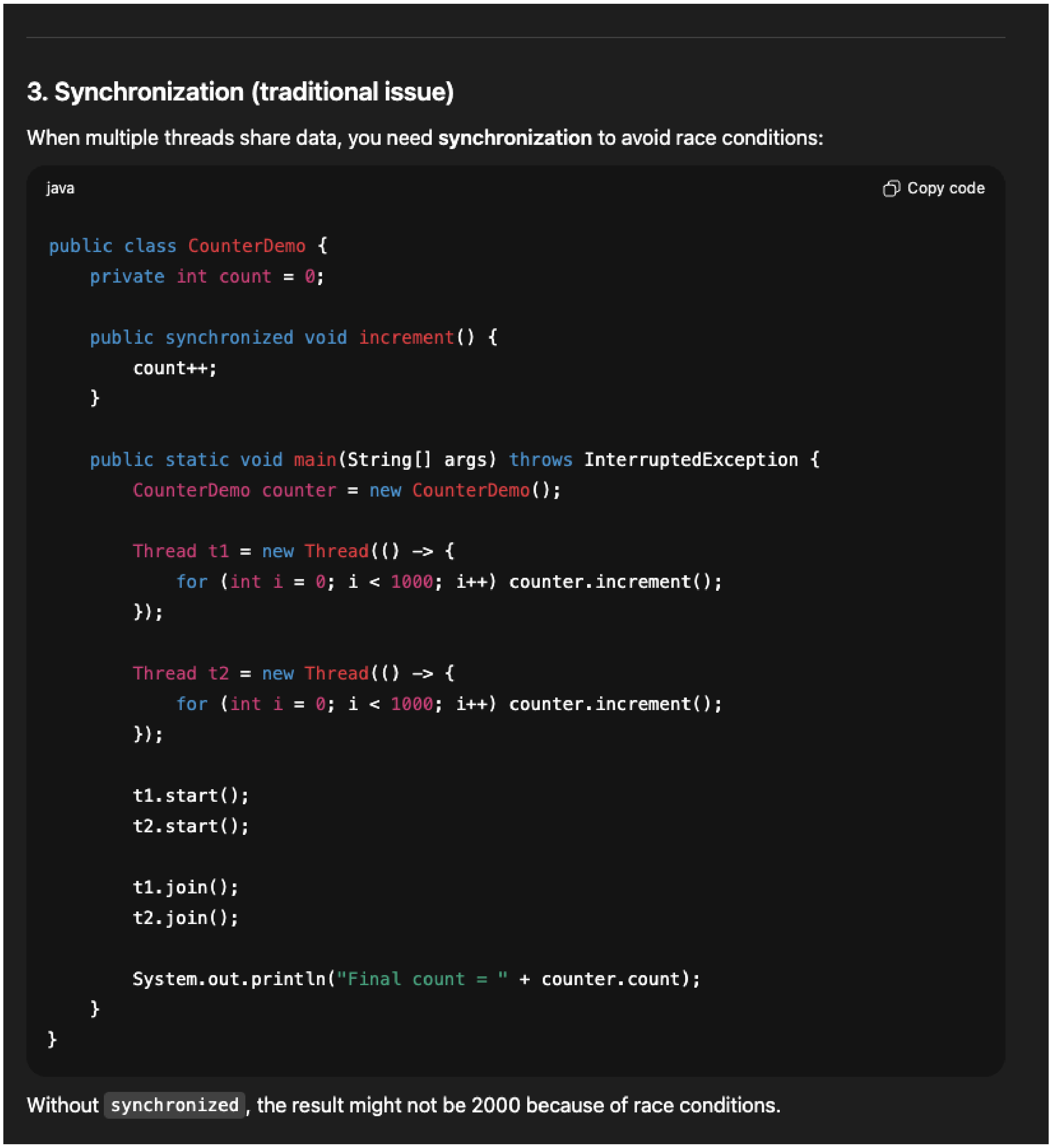
```java
import java.util.concurrent.*;

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

        CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> {
            sleep(1000);
            return 10;
        }, executor);

        CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(() -> {
            sleep(1500);
            return 20;
        }, executor);

        // Combine both results asynchronously
```



3. Synchronization (traditional issue)

When multiple threads share data, you need **synchronization** to avoid race conditions:

```java
public class CounterDemo {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public static void main(String[] args) throws InterruptedException {
        CounterDemo counter = new CounterDemo();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

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

        t1.join();
        t2.join();

        System.out.println("Final count = " + counter.count);
    }
}
```

Without `synchronized`, the result might not be 2000 because of race conditions.