Coroutines Mechanics

To make efficient and safe use of Coroutines in complex scenarios, you need to understand their underlying mechanics

Main building blocks of Coroutines framework:

`CoroutineScope`

`CoroutineContext`

`CoroutineDispatcher`

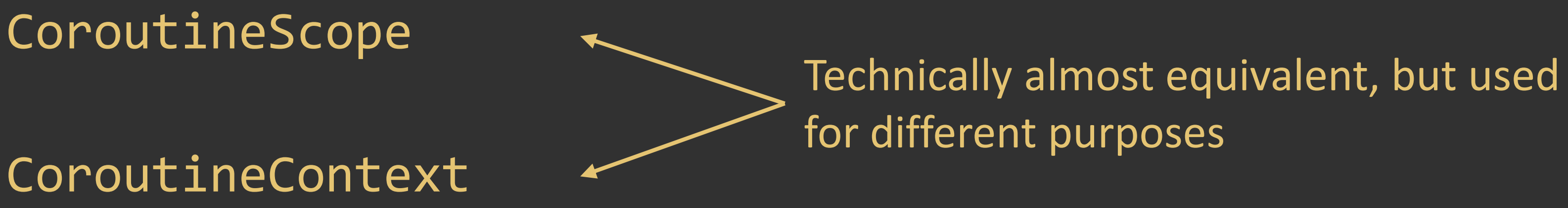
`Job`

You’ve already worked with all these building blocks…

… in this section, you’ll learn about their respective roles and inter-relationships in details

Coroutines Mechanics Summary

Main building blocks of Coroutines framework:



CoroutineScope

CoroutineContext

Technically almost equivalent, but used for different purposes

CoroutineDispatcher

Job

The most important aspect of Coroutines mechanics is
Jobs Hierarchy

Nested withContext:

```
runBlocking {
    val scopeJob = Job()
    val scope = CoroutineScope(scopeJob + Dispatchers.Default)
    val job = scope.launch {
        delay(500)
        println("before nested")
        withContext(Dispatchers.IO) {
            delay(500)
            printJobsHierarchy(scopeJob)
            println("nested")
        }
        println("after nested")
    }
    job.invokeOnCompletion { println("coroutine completed") }
    job.join()
}
```

Jobs hierarchy:

- scope Job
  - coroutine Job
    - context Job

~~Concurrency~~

Structured Concurrency

Nested coroutine:

```
runBlocking {
    val scopeJob = Job()
    val scope = CoroutineScope(scopeJob + Dispatchers.Default)
    val job = scope.launch {
        delay(500)
        println("before nested")
        val nestedJob = launch(Dispatchers.IO) {
            delay(500)
            printJobsHierarchy(scopeJob)
            println("nested")
        }
        nestedJob.invokeOnCompletion {
            println("nested coroutine completed")
        }
        println("after nested")
    }
    job.invokeOnCompletion { println("coroutine completed") }
    job.join()
    delay(1000)
}
```

Jobs hierarchy:

- scope Job
  - coroutine Job
    - nested coroutine Job

Concurrency

Structured Concurrency

Nested coroutine on a standalone scope:

```
runBlocking {
    val scopeJob = Job()
    val scope = CoroutineScope(scopeJob + Dispatchers.Default)
    val job = scope.launch {
        delay(500)
        println("before nested")
        val nestedJob = scope.launch(Dispatchers.IO) {
            delay(500)
            printJobsHierarchy(scopeJob)
            println("nested")
        }
        nestedJob.invokeOnCompletion {
            println("nested coroutine completed")
        }
        println("after nested")
    }
    job.invokeOnCompletion { println("coroutine completed") }
    job.join()
    delay(1000)
}
```

Jobs hierarchy:
- scope Job
    - coroutine Job
    - nested coroutine Job

Concurrency

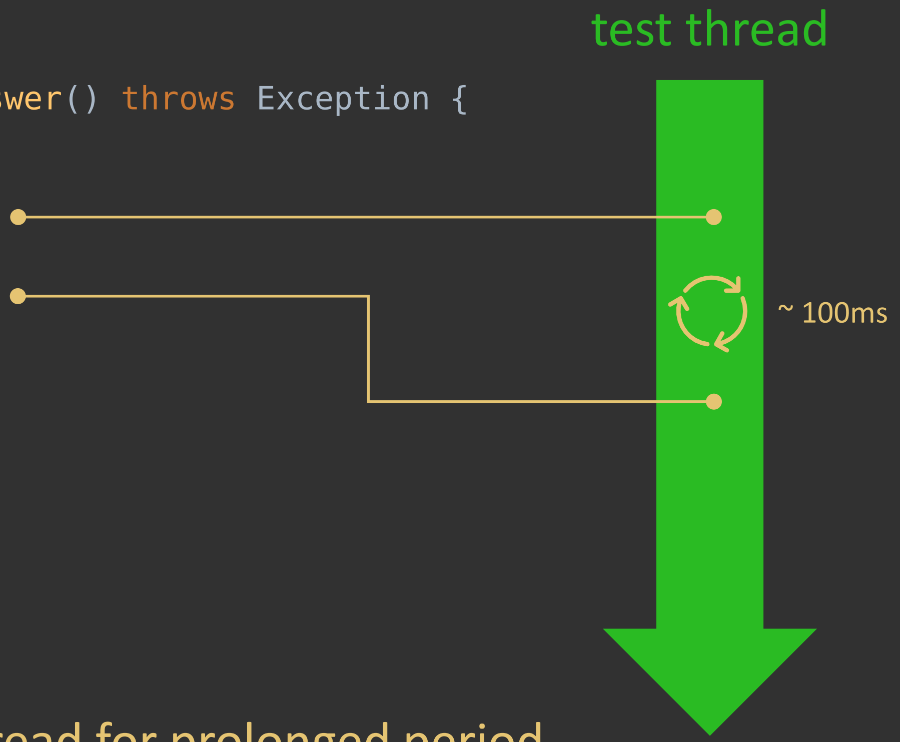
~~Structured Concurrency~~

NonCancellable is “detaching” withContext from its parent Job

NonCancellable is designed for withContext exclusively!

Synchronous implementation:

```
@Test
public void computeFibonacci_10_returnsCorrectAnswer() throws Exception {
    // Arrange
    // Act
    BigInteger result = SUT.computeFibonacci(10);
    // Assert
    assertThat(result, is(new BigInteger("55")));
}
```

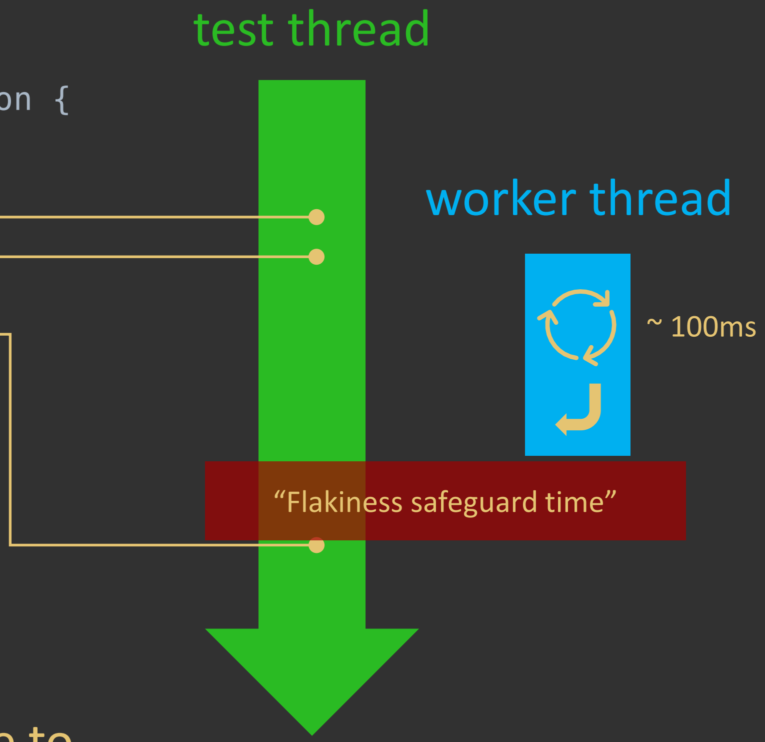


Pros: simple to unit test

Cons: production code blocks the calling thread for prolonged period

Concurrent implementation with async callback:

```
@Test
public void computeFibonacci_10_returnsCorrectAnswer() throws Exception {
    // Arrange
    // Act
    SUT.computeFibonacci(10, mCallback);
    Thread.sleep(200);
    // Assert
    assertThat(lastResult, is(new BigInteger("55")));
}
```

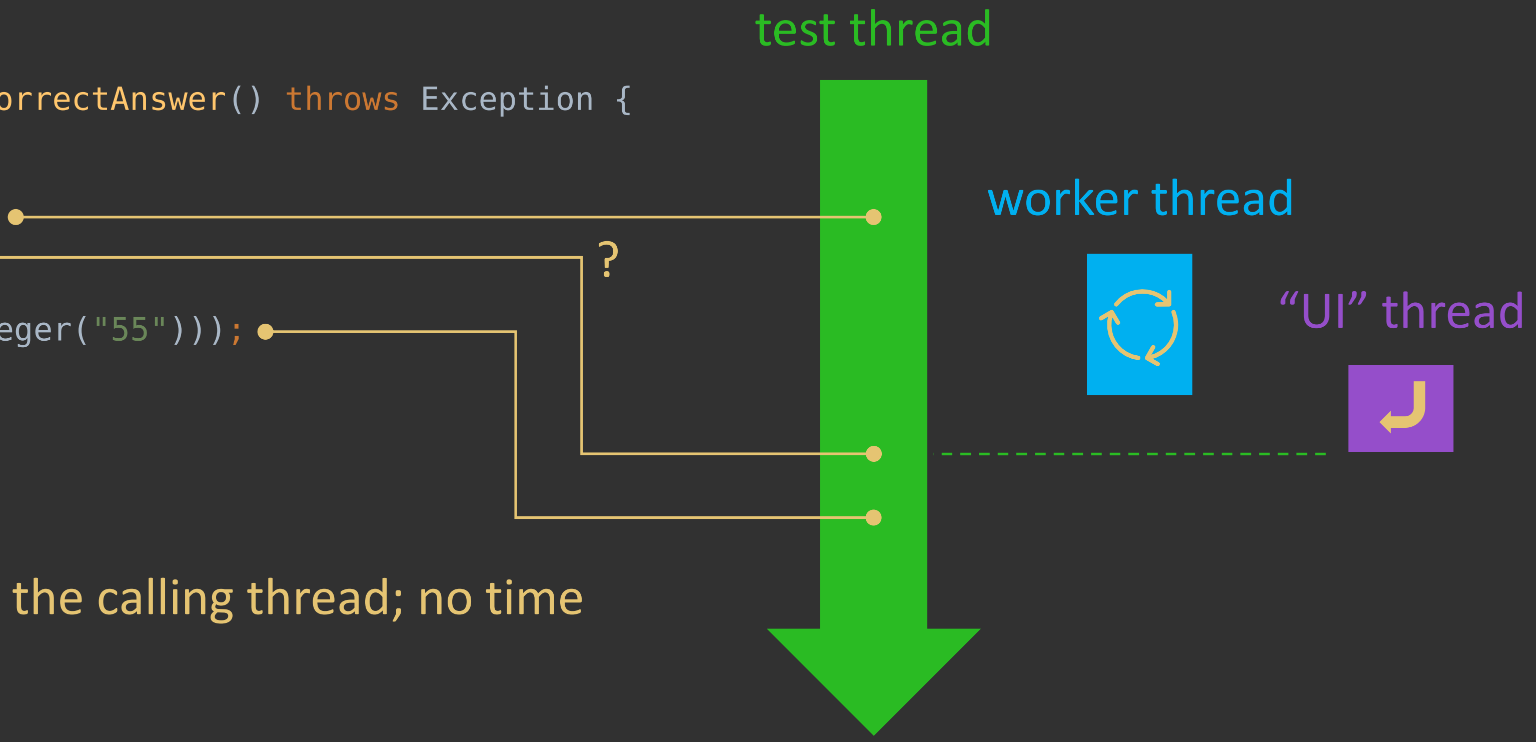


Pros: production code doesn’t block the calling thread

Cons: flaky unit test which takes too much time to execute due to “flakiness safeguard time”; callback invoked on the worker thread

Concurrent implementation with async callback using ThreadPoster:

```
@Test
public void computeFibonacci_10_returnsCorrectAnswer() throws Exception {
    // Arrange
    // Act
    SUT.computeFibonacci(10, mCallback);
    mThreadPostersTestDouble.join();
    // Assert
    assertThat(lastResult, is(new BigInteger("55")));
}
```

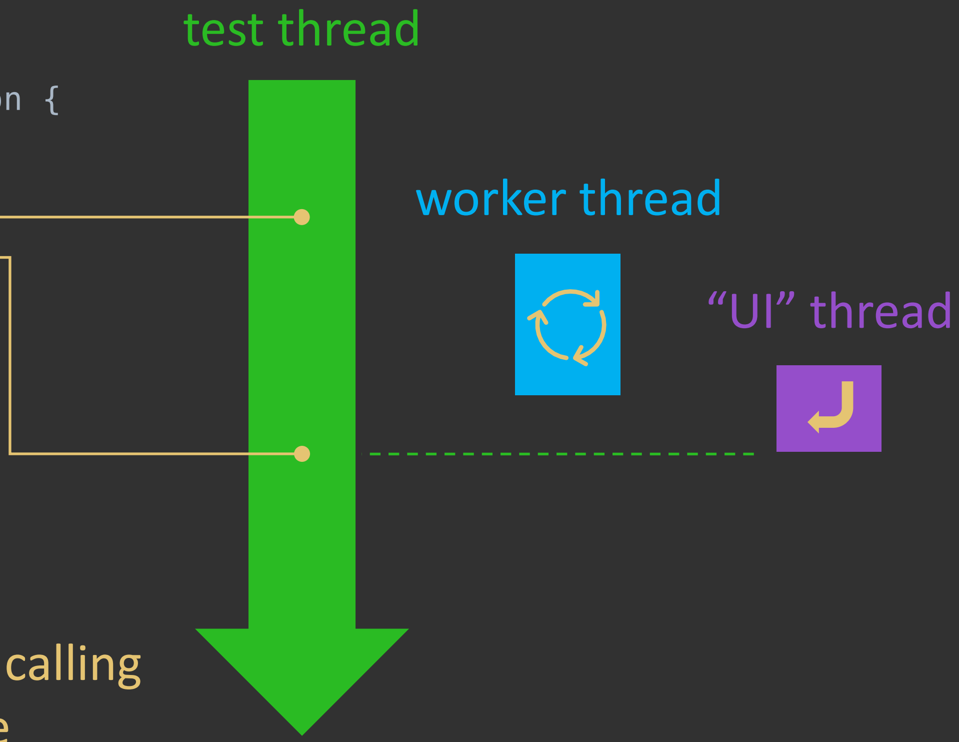


Pros: production code doesn't block the calling thread; no time overhead; relatively simple

Cons: requires developers to follow additional conventions

Concurrent implementation with async callback using ThreadPoster:

```
@Test
public void computeFibonacci_10_returnsCorrectAnswer() throws Exception {
    // Arrange
    // Act
    SUT.computeFibonacci(10, mCallback);
    mThreadPostersTestDouble.join();
    // Assert
    assertThat(lastResult, is(new BigInteger("55")));
}
```



`.join()` functionality: do not proceed with execution (block calling thread) until all flows, including the concurrent ones, complete

➡ **Structured Concurrency!**

Structured Concurrency:

an ability to “pause” code execution and “wait” for all concurrent flows which can be traced back to a specific “ancestor” to complete

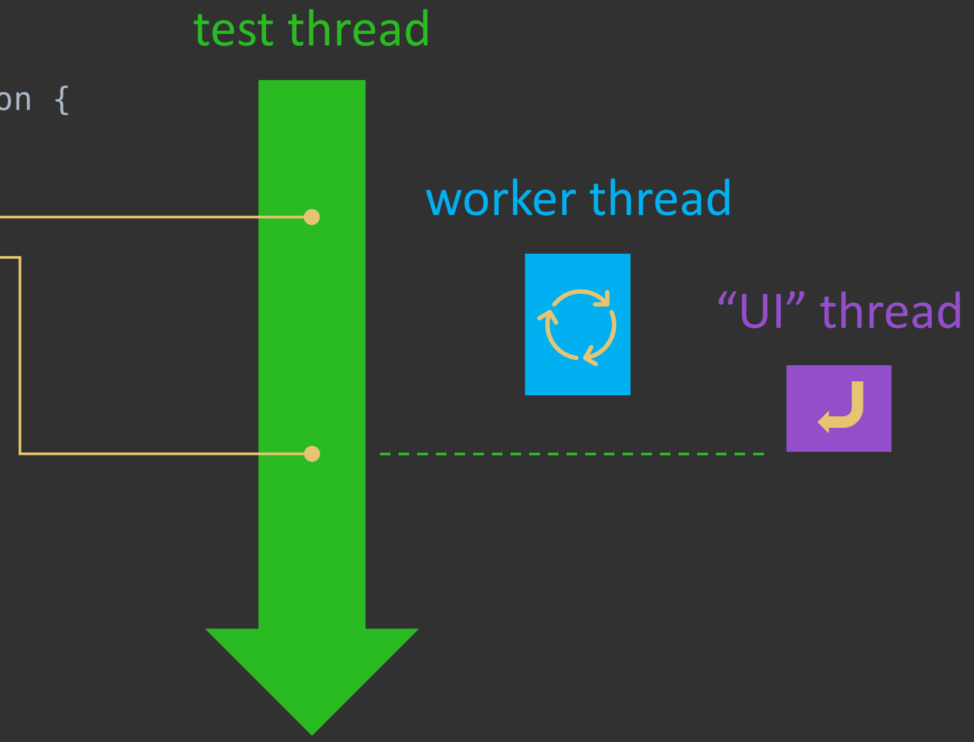
ThreadPoster provides very basic support for Structured Concurrency in unit tests

Kotlin Coroutines provide advanced support for
Structured Concurrency everywhere

Structured Concurrency Summary

Structured Concurrency using ThreadPoster:

```
@Test
public void computeFibonacci_10_returnsCorrectAnswer() throws Exception {
    // Arrange
    // Act
    SUT.computeFibonacci(10, mCallback);
    mThreadPostersTestDouble.join();
    // Assert
    assertThat(lastResult, is(new BigInteger("55")));
}
```



Structured Concurrency:

an ability to “pause” code execution and “wait” for all concurrent flows which can be traced back to a specific “ancestor” to complete

Kotlin Coroutines provide advanced support for Structured Concurrency everywhere

Does Structured Concurrency make concurrent code safer?

I don’t think so!

Structured Concurrency allows for more straightforward implementation of some concurrent flows