

# Structured Concurrency

# From Unit Testing Challenges to Structured Concurrency

# Synchronous implementation:

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

test thread



~ 100ms

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")));
}
```

test thread

worker thread



~ 100ms

"Flakiness safeguard time"

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")));
}
```

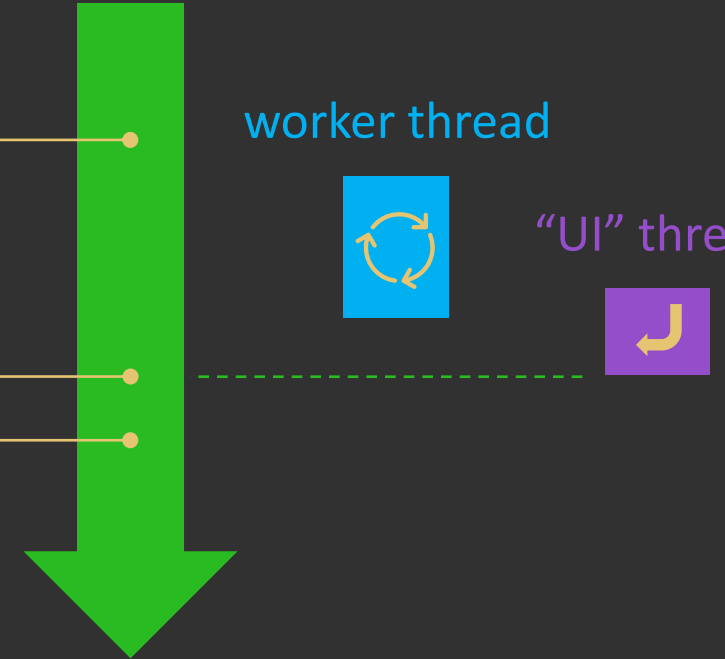
test thread

worker thread

“UI” thread

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")));
}
```

test thread

worker thread

"UI" thread

.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

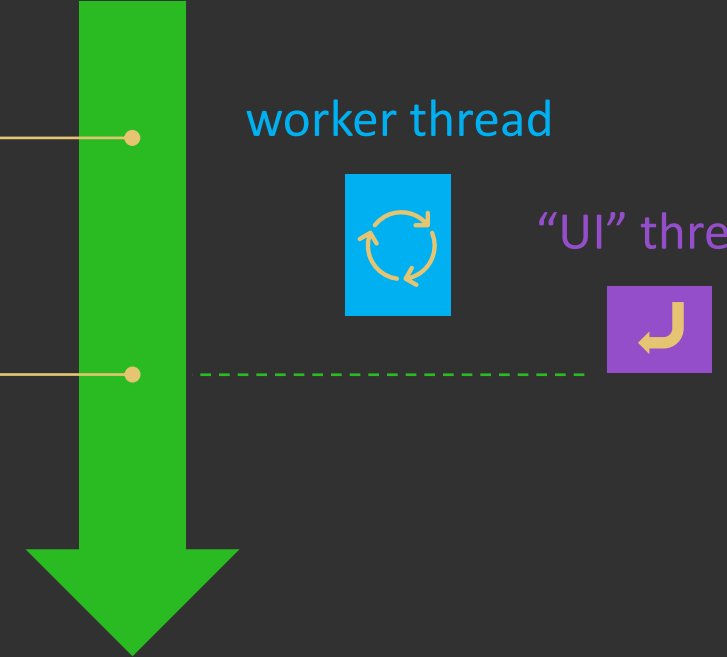
# Structured Concurrency using ThreadPoster:

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

test thread

worker thread

“UI” thread



## 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