Kotlin: Using Coroutines

INTRODUCTION

Kevin Jones

@kevinrjones www.rocksolidknowledge.com

Introduction

Introduce you to coroutines

Examine builders and 'suspend' functions

Coordination of coroutines

Returning data from coroutines

Using 'channels'

Writing actors and using 'select'

Using coroutines in UI applications

Why care about asynchronous programming?

Coroutines are experimental, using them with Maven and Gradle projects

Our first coroutine

Examine the cost of coroutines

Computers Are Not Getting Faster

Speed used to double every 18 months

- If your PC was too slow then wait 18 months and it would be quick enough

This stopped around 2005

“Moore's Law” Continues, Though

Still doubling the transistors on a chip every 18 months

- This will stop soon

Now rather than being faster PCs have more cores

- Need to be able to take advantage of these cores

Moore's Law

Show this:

http://www.alleywatch.com/2017/03/preparing-end-moores-law/

More Cores Means

Make your application multi-threaded

Corollary: Threads are heavyweight and hard to manage

How Do We Do This in Java?

Fork/Join Pool

- Introduced in Java 7 (2011)
- Meant for small, related tasks
- Supports work stealing

Calculate a Sum of Values in an Array

```
fun compute(array: IntArray, low: Int, high: Int): Long {
    return if (high - low <= SOME_THRESHOLD) {
        (low until high)
                .map { array[it].toLong() }
                .sum()
    } else {
        val mid = low + (high - low) / 2
        val left = compute(array, low, mid)
        val right = compute(array, mid, high)
        return left + right
    }
}
```

(Reminder to delete)

Demo Next

Show the 1.0 demo – what the sum does

1.1 demo – with Fork Join

1.0 demo – reminder of the code

1.2 demo – The beauty of the Kotlin code

Demo

Using Fork/Join Pool

Issues with the fork/join Code

Conceptually, the code idea is easy

However:

- Lots of ceremony in the code
- fork, join, compute
- Actual functionality lost in ceremony

Same Code with Coroutines

Code looks the same as the non-fork/join code

- Easier to read
- Way less ceremony
- Uses the same underlying code as fork/join code

Asynchronous Programming Styles

Callbacks

Futures

Callbacks

A way to do asynchronous code

- Prevalent in JavaScript, for example

Callback Hell

```
fun addBlog(title: String) {
    authenticate() { id ->
        createBlogAsync(id, title) { blog -> {
                processBlog(blog)
            }
        }
    }
}
```

Using Futures

Java provides 'Future' classes

- Easier than callbacks

Many different libraries

- And so many different approaches

Futures

```
fun addBlog(title: String) {
    authenticate()
        .thenCompose { id ->createBlogAsync(id, title) }
        .thenAccept { blog -> processBlog(blog) }
```

Using Coroutines

Coroutines are more natural

Coroutines

```
suspend fun addBlog(title: String) {
    val id = authenticate()
    val blog = createBlogAsync(id, title)
    processBlog(blog)
}
```

Using Coroutines

Coroutines are more natural

- Looping constructs are natural
- Exception handling is natural

(Reminder to delete)

Demo Next

See https://kotlinlang.org/docs/tutorials/coroutines-basic-jvm.html for the maven bits

Any demo for the gradle bits

Demo

Setting up Kotlin Coroutines in Maven and Gradle

Our First Coroutines

Use the 'launch' coroutine builder

Demo

'launch' coroutine builder

Demo

Running lots of coroutines

Summary

Kotlin Coroutines?

- Provide an asynchronous programming mechanism
- 'Lightweight' threads

What's Next