Mobile Application Development

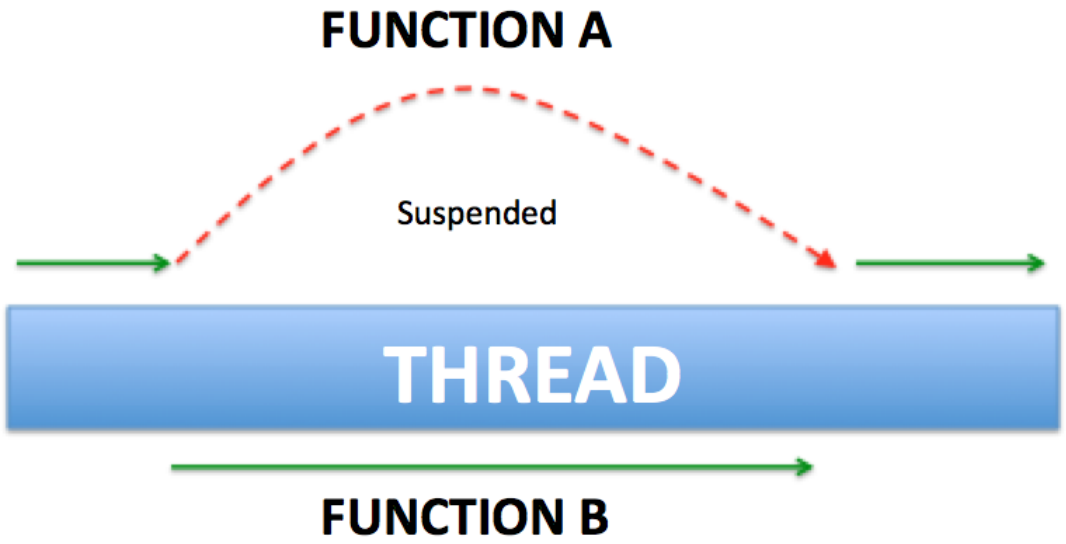
Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Introducing Coroutines



Agenda

- Context
- Background (What, Why & How)

Context

- If you're coming from Java, you probably associate asynchronous code with threads :
  - you've dealt with shared mutable state
  - you've spent countless hours chasing down **deadlocks** and **race conditions**
  - you've taken care when modifying shared state using locking primitives like **synchronized**.
- At this point you've realized that **concurrency** is hard. When you take a single threaded piece of code and make it concurrent, you inherently introduce a tremendous amount of complexity.

Context

The main purpose of **coroutines** is to take care of the complexities in working with asynchronous programming & concurrency

Background - What

Coroutine

From Wikipedia, the free encyclopedia

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.

According to Donald Knuth, Melvin Conway coined the term *coroutine* in 1958 when he applied it to construction of an assembly program.[1] The first published explanation of the coroutine appeared later, in 1963.[2]

- Coroutines provide a way to write asynchronous code sequentially making multithreaded programming more debuggable and maintainable.

Background - Why

- The problem coroutines try to tackle is *how to prevent our applications from blocking*.

- Asynchronous or non-blocking programming is the new reality both on the client side to provide a fluid experience and on the server side for a more scalable architecture.

- There are several approaches to this problem including Threads, Callbacks, Futures/Promises and Reactive extensions (Rx).

Background - Why

- If you've worked with Rx, then you know it takes a lot of effort to get to know it enough, to be able to use it safely

- On the other hand, **AsyncTasks** and **Threads** can easily introduce leaks and memory overhead

- Finally, relying on all these APIs, which use **callbacks**, can introduce a ton of code. Not only that, but the code can become unreadable, as you introduce more callbacks.

Background - How (high level overview)

- Based on the concept of **suspending functions** coroutines approached the problem in a different way.

- Main advantage over competitive approaches is that the **structure of the code is still sequential** and easier to write.

- **Kotlin** provides coroutine support at the language level but the functionality is delegated to libraries.
  - Only one keyword is added to the core language, the ***suspend*** keyword.

Background - How (high level overview)

- Similar to threads, they take a block of code and have a lifecycle - it can be *created*, *started*, *suspended*, *resumed* and *terminated* (cancelled).

- Coroutines are non-blocking and you can have many of them running on a single thread.

- In the world of a coroutine, suspending and resuming work ***replaces the need for callbacks***.

Background - How (high level overview)

How in Kotlin?

References

Sources:

https://antonis.me/2018/12/12/an-introduction-to-kotlin-coroutines/
https://proandroiddev.com/kotlin-coroutines-channels-csp-android-db441400965f
https://superkotlin.com/coroutines/
https://www.codementor.io/jimmy_chuks/introduction-to-coroutines-xr674rfpo


Thanks.