Kotlin Coroutines

HELLO!

I am Adriano Belfort

Platforms Team - PlayKids

@adrianobelfort

“We write systems that communicate”

```kotlin
fun getPageText(url: String): String {
    val response = networkCall(url)

    return response.body
}

fun networkCall(url: String): Response {
    println("Fetching $url")

    Thread.sleep(2000)      // Blocks thread

    return Response(200, url.split('.')[1]))
}
```

```kotlin
fun getPageText(url: String, callback: (String) -> Unit) {
    networkCall(url) { response ->
        callback(response.body)
    }
}

fun networkCall(url: String, callback: (Response) -> Unit) {
    println("Fetching $url")

    executeLater({
        // Fetches data, does not block current thread (…)

        callback(Response(200, url.split('.')[1]))
    })
}
```

```
getPageText("www.google.com") { google ->
    // Do stuff with ‘google'
}
```

```
getPageText("www.google.com") { google ->
    getPageText("www.microsoft.com") { microsoft ->
        getPageText("www.movile.com") { movile ->
            getPageText("www.playkids.com") { playkids ->
                getPageText("www.amazon.com") { amazon ->
                    println("Pages: $google, $microsoft,
                        $movile, $playkids, $amazon")
                }
            }
        }
    }
}
```

Is there a nicer way of writing asynchronous code?

JavaScript Promises

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

RxJava (Android)

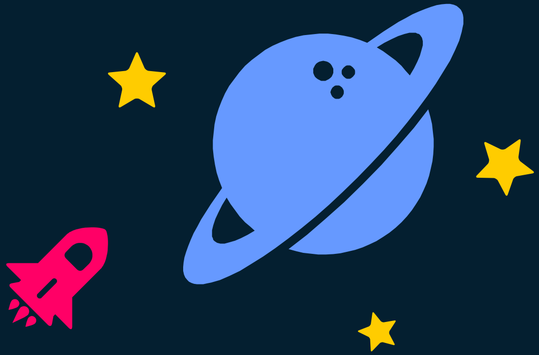
```
animalsObservable
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeWith(animalsObserver));
```

https://www.androidhive.info/RxJava/android-getting-started-with-reactive-programming/

Is there an even nicer way of writing asynchronous code?

```kotlin
val google = getPageText("www.google.com")
val microsoft = getPageText("www.microsoft.com")
val movile = getPageText("www.movile.com")
val playkids = getPageText("www.playkids.com")
val amazon = getPageText("www.amazon.com")

println("Pages: $google, $microsoft, $movile, $playkids, $amazon")
```



COROUTINES

Writing asynchronous code in a synchronous way

Coroutine

“ A coroutine can be thought of as an instance of *suspendable computation*, i.e. the one that can suspend at some points and later resume execution possibly on another thread. ”

(Kotlin Coroutines Design Document)

What’s a coroutine?

Lightweight thread

Coroutines are designed to be *cheap*.

Launch several coroutines with lower effort than threads.

Suspendable code

Instead of blocking, threads are released and can run other tasks.

Async programming

Write asynchronous code in a synchronous fashion.

Treat returns and exceptions in natural way.

No need to learn new APIs

Suspending functions

Suspending functions

- A suspending function can suspend execution and resume when the call is finished
- The thread is free to execute other tasks while the function is suspended
- Have their own modifier keyword: suspend
- Sequential by default

Suspending functions

```
suspend fun getPageText(url: String): String {
    val response = networkCall(url)
    return response.body
}

suspend fun networkCall(url: String): Response {
    println("Fetching $url")

    delay(2000)

    return Response(200, url.split('.')[1])
}
```

Comparing versions

Synchronous

```kotlin
fun getPageText(url: String): String {
    val response = networkCall(url)
    return response.body
}

fun networkCall(url: String): Response {
    println("Fetching $url")

    Thread.sleep(2000)

    return Response(200, url.split('.')[1]))
}
```

Asynchronous

```kotlin
suspend fun getPageText(url: String): String {
    val response = networkCall(url)
    return response.body
}

suspend fun networkCall(url: String): Response {
    println("Fetching $url")

    delay(2000)

    return Response(200, url.split('.')[1])
}
```

With suspending functions

```
suspend fun getPageText(url: String): String {
    val response = networkCall(url)
    return response.body
}

suspend fun networkCall(url: String): Response {
    println("Fetching $url")

    delay(2000)

    return Response(200, url.split('.')[1])
}
```

With callback lambdas

```
fun getPageText(url: String, callback: (String) -> Unit) {
    networkCall(url) { response ->
        callback(response.body)
    }
}

fun networkCall(url: String, callback: (Response) -> Unit)
{
    println("Fetching $url")

    executeLater({
         // Fetches data, does not block current thread (…)
         callback(Response(200, url.split('.')[1]))
     })
}
```

CONTINUATION-PASSING STYLE

(CPS)

CPS

“A function written in continuation-passing style takes an extra argument: an explicit "continuation", i.e. a function of one argument. When the CPS function has computed its result value, it "returns" it by calling the continuation function with this value as the argument.”

(Wikipedia)

Revisiting async code

```
fun getPageText(url: String, callback: (String) -> Unit) {
    networkCall(url) { response ->
        callback(response.body)
    }
}

fun networkCall(url: String, callback: (Response) -> Unit) {
    println("Fetching $url")

    executeLater({
         // Fetches data, does not block current thread (…)

         callback(Response(200, url.split('.')[1]))
     })
}
```

Revisiting async code

```
fun getPageText(url: String, callback: (String) -> Unit) {
    networkCall(url) { response ->
        callback(response.body)
    }
}

fun networkCall(url: String, callback: (Response) -> Unit) {
    println("Fetching $url")

    executeLater({
         // Fetches data, does not block current thread (…)

         callback(Response(200, url.split('.')[1]))
    })
}
```

Continuation

Continuation in Kotlin

```kotlin
public interface Continuation<in T> {
    /**
     * Context of the coroutine that corresponds to this continuation.
     */
    public val context: CoroutineContext

    /**
     * Resumes the execution of the corresponding coroutine passing
     * successful or failed [result] as the return value of the
     * last suspension point.
     */
    public fun resumeWith(result: Result<T>)
}
```

Continuation in Kotlin

- The compiler generates a continuation for each suspension point
- Method signature changes in the bytecode

```kotlin
fun getPageText(url: String, cont: Continuation<String>): Any?
```

Coroutine Context

- Set of objects to help coroutine execution
- Holds job, dispatcher, and other user-defined objects
- Comprised of singleton elements

How to create a coroutine?

Coroutine Builders

launch

Creates and starts a fire-and-forget coroutine

runBlocking

Creates and starts a coroutine where suspension means blocking the current thread

async

Creates and starts a coroutine that eventually produces a result

launch

```
launch {
    println("Launched coroutine")
    delay(1000)
    println("Delay happened without blocking thread")
}
```

launch

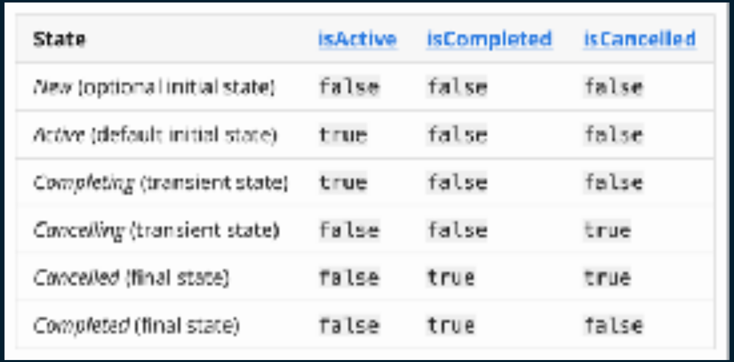
```
// Create pool of threads where coroutines will run

val dispatcher =
    Executors.newFixedThreadPool(2).asCoroutineDispatcher()


val job = GlobalScope.launch(dispatcher) {
    println("Launched coroutine")
    delay(1000)
    println("Delay happened without blocking thread")
}
```

Job

- Cancellable object with a (coroutine) lifecycle



State	isActive	isCompleted	isCancelled
New (optional initial state)	false	false	false
Active (default initial state)	true	false	false
Completing (transient state)	true	false	false
Cancelling (transient state)	false	false	true
Cancelled (final state)	false	true	true
Completed (final state)	false	true	false

https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html

runBlocking

```kotlin
fun main() = runBlocking {
    println("Before delaying")
    delay(1000)  // Blocks current thread
    println("After delaying")
}
```

async

```kotlin
suspend fun downloadFile(location: String): Content { … }

fun processPhoto(photo: Content) { … }


// Inside coroutine

val photoDeferred = async(ioDispatcher) {
    downloadFile("s3://bucket/photo.jpg")
}

// Do more stuff

processPhoto(photoDeferred.await())
```

Deferred

- Cancellable object with a lifecycle
- Holds a result

Review

Coroutine

A lightweight instance of a suspendable computation that runs suspending functions

Suspending function

A function that can suspend and resume without blocking a thread

Continuation

Is a lambda that receives a result and processes that result

Context

A user-defined indexed set of elements to aid coroutine execution

Coroutine Builder

A function that creates a coroutine and runs a suspending lambda within it

Job & Deferred

A handle to a background operation. Deferred provides a value with await()

Concurrency

Concurrency with coroutines

Hard

There’s no free lunch.

Shared mutable state continues to be a problem.

Communication primitives might be used to help

Explicit & Opt-in

Nested builders inside coroutines.

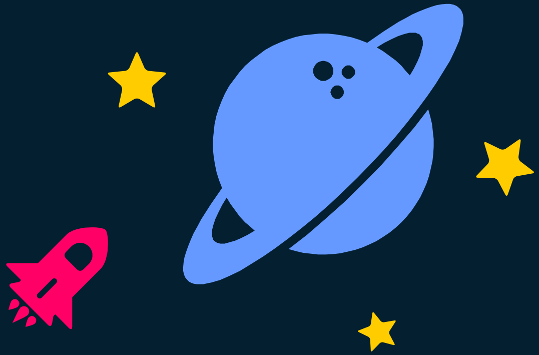
Concurrent coroutines can be created normal coroutine builders.

One must await for async results.

Limited is best

Several requests in a short amount of time might exhaust external resources.

No back pressure handling.



DEMO TIME!

THANKS!

Any questions?

You can find me at:
@adrianobelfort · adriano.belfort@playkids.com

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Startupstockphotos