An introduction to Kotlin Coroutines

Don’t block
Keep moving

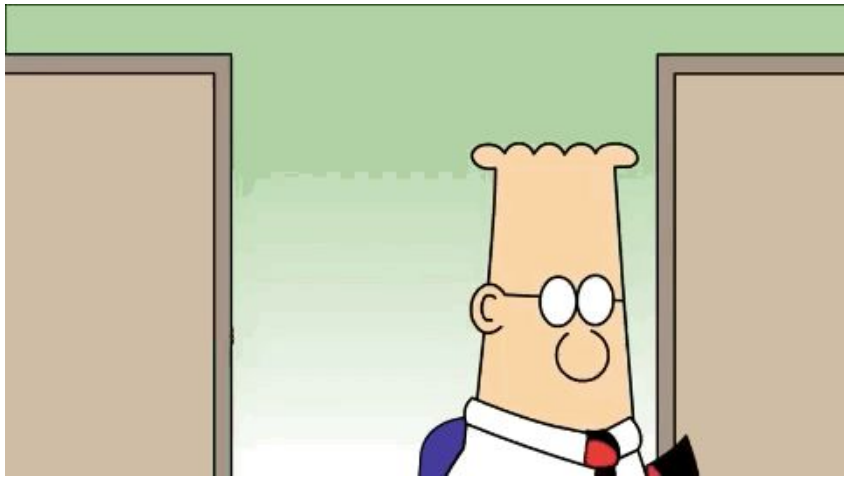
Antonis Lilis, Mobile Engineer

ADVANTAGE
FINANCIAL SYSTEMS EXPERTS

The Problem

How to **prevent** our applications **from blocking**

- Asynchronous or non-blocking programming is the new reality
  - **Fluid** experience on the **client**
  - **Scalable** architecture on the **server**



Approaches

- **Threads**
  - hard to write and maintain
- **Callbacks**
  - series of nested callbacks which lead to incomprehensible code
- **Futures**, **Promises**,...
  - different programming mental model
- **Reactive** Extensions
  - everything is a stream, and it's observable
- **Coroutines**

Coroutines

- Based on the concept of **suspending function**
- The code is still **structured as** if we were writing **synchronous** code
- Are like **light-weight threads**

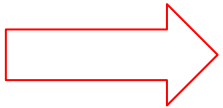
The term 'Coroutine' was coined by Melvin Conway in 1958 (known for Conway's Law)

Kotlin Coroutines

- Kotlin provides **Coroutine support** at the language level
  - Actually it only adds one language keyword (**suspend**)
- Functionality is delegated to **libraries**
  - kotlinx.coroutines is a library developed by JetBrains
- Since Kotlin **1.3** Coroutines are no longer experimental
  - The major feature of this release

Suspending Functions - Continuations

```
1  fun main() {
2      val numbers = sequence {
3          println("one")
4          yield( value: 1)
5
6          println("two")
7          yield( value: 2)
8
9          println("three")
10         yield( value: 3)
11
12         println("...done")
13     }
14
15     for (n in numbers) {
16         println("number = $n")
17     }
18 }
```



```
one
number = 1
two
number = 2
three
number = 3
...done
```

Synchronous - Sequential Code

```
1   import ...
3
4   private fun google(keyword: String): String {...}
11
12  private fun wikipedia(keyword: String): String {...}
23
24  fun main() {
25      val keyword = "Meetup"
26      val gResult = google(keyword)
27      val wResult = wikipedia(keyword)
28      println("Google replied: $gResult \n" +
29              "Wikipedia replied: $wResult")
30  }
```

Asynchronous - Concurrent Code

```
1   import ...
4
5   private fun google(keyword: String): String {...}
12
13  private fun wikipedia(keyword: String): String {...}
24
25  fun main() = runBlocking { this: CoroutineScope
26      val keyword = "Meetup"
27      launch (Dispatchers.Default) { this: CoroutineScope
28          val gResult = async { google(keyword) }
29          val wResult = async { wikipedia(keyword) }
30          println("Google replied: ${gResult.await()} \n" +
31                  "Wikipedia replied: ${wResult.await()}"
32          )
33      }
34      println("Launched Coroutine")
35  }
```

Coroutines are light-weight

```
1  import kotlinx.coroutines.*
2
3  fun main() = runBlocking { this: CoroutineScope
4      //Launch 1M coroutines
5      repeat( times: 1_000_000) { counter ->
6          launch { this: CoroutineScope
7              print(", $counter")
8          }
9      }
10     //try that with threads!
11 }
```

Suspending Functions (synchronous example)

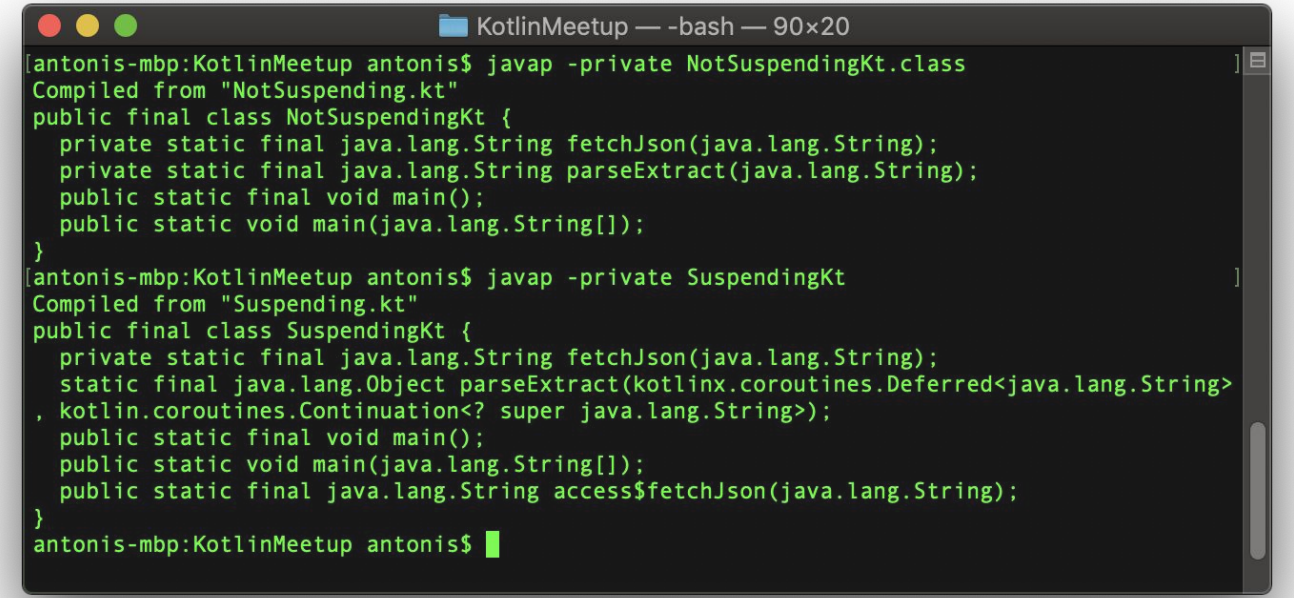
```
1   import com.google.gson.*
2
3   private fun fetchJson(term: String): String {...}
10
11  private fun parseExtract(wikipediaJson: String): String {...}
17
18  fun main()  {
19      val terms = listOf("Kotlin", "Athens", "Meetup")
20      val extracts = mutableListOf<String>()
21      terms.forEach { it: String
22          val json = fetchJson(it)
23          extracts += parseExtract(json)
24      }
25      extracts.forEach { println(it) }
26  }
```

Suspending Functions (example)

- Can be used inside coroutines just **like regular functions**
- They can **call other Suspending functions**
- **Waits tasks to complete** before returning

```
 4      private fun fetchJson(term: String): String {...}
11
12      private suspend fun parseExtract(wikipediaJson: Deferred<String>): String {...}
18
19      fun main() = runBlocking { this: CoroutineScope
20          val terms = listOf("Kotlin", "Athens", "Meetup")
21          val extracts = mutableListOf<String>()
22          terms.forEach { it: String
23              val json = async { fetchJson(it) }
24              extracts += parseExtract(json)
25          }
26          extracts.forEach { println(it) }
27      }
```

Suspending Functions (behind the scenes)



Coroutine Builders

- Create a coroutine and **provide a CoroutineScope**
- Examples are runBlocking, launch, async etc
- GlobalScope.launch creates a top-level coroutine (like a Thread)

```
3  suspend fun main() {
4      // launch new coroutine and keep a reference to its Job
5      val job = GlobalScope.launch { this: CoroutineScope
6          delay( timeMillis: 1000L)
7          println("World!")
8      }
9      println("Hello,")
10     job.join() // wait until child coroutine completes
11 }
```

CoroutineScope

- Coroutines are launched in the scope of the operation we are performing
- We can declare a scope using coroutineScope builder

```
3  fun main() = runBlocking { this: CoroutineScope
4      launch { this: CoroutineScope
5          delay( timeMillis: 200L)
6          println("Kotlin")
7      }
8
9      coroutineScope { this: CoroutineScope
10         launch { this: CoroutineScope
11             delay( timeMillis: 300L)
12             println("Athens")
13         }
14
15         delay( timeMillis: 100L)
16         println("Hello")
17         //coroutineScope does NOT block the current thread while waiting for all children to complete
18     }
19
20     println("Meetup")
21     //runBlocking DOES block the current thread while waiting for all children to complete
22 }
```

CoroutineContext

- Is an an **optional parameter** of all coroutine builders
- Includes a coroutine dispatcher that determines the **execution thread**
- **inherited** from the CoroutineScope if not defined

```
3  fun main() = runBlocking<Unit> { this: CoroutineScope
4      launch { // context of the parent, main runBlocking coroutine
5          println("main runBlocking: Thread ${Thread.currentThread().name}")
6      }
7      launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
8          println("Unconfined: Thread ${Thread.currentThread().name}")
9      }
10     launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
11         println("Default: Thread ${Thread.currentThread().name}")
12     }
13     launch(newSingleThreadContext( name: "MyOwnThread")) { // will get its own new thread
14         println("newSingleThreadContext: Thread ${Thread.currentThread().name}")
15     }
16 }
```

Coroutine Cancelation

```
3  fun main() = runBlocking { this: CoroutineScope
4      //Cancel with timeout
5      try {
6          withTimeout( timeMillis: 1000L) { this: CoroutineScope
7              repeat( times: 1000) { counter ->
8                  print(" $counter")
9                  delay( timeMillis: 100L)
10             }
11         }
12     } catch (e: TimeoutCancellationException) {
13         print(" Timeout\n")
14     }
15
16     //Cancel manually
17     val job = launch { this: CoroutineScope
18         try {
19             repeat( times: 1000) { counter ->
20                 print(" $counter")
21                 delay( timeMillis: 50L)
22             }
23         } catch (e: CancellationException) {
24             print(" CancellationException\n")
25         }
26     }
27     delay( timeMillis: 1000L)
28     job.cancel()
29     job.join()
30 }
```

- A coroutine code has to **cooperate** to be cancellable
- All the suspending functions in kotlinx.coroutines are **cancellable**

Concurrency is not Parallelism

- **Parallelism** is about the execution of **multiple tasks at the same time**
- **Concurrency** tries to break down tasks which we don’t necessarily need to execute at the same time
- Concurrency's primary goal is **structure**, not parallelism.
- Concurrency makes the use of **parallelism easier**

Structured Concurrency

```
1   import kotlinx.coroutines.*
2   import org.jsoup.Jsoup
3
4   val countries : List<String>  = listOf(...)
134
135 fun google(keyword: String): String {...}
142
143 suspend fun google(keywords: List<String>) = coroutineScope {
144     for (keyword in keywords) {
145         println("Googling $keyword")
146         launch { this: CoroutineScope
147             val result = google(keyword)
148             println("Result for $keyword: $result")
149         }
150     }
151 }
152
153 suspend fun main() {
154     google(countries)
155 }
```

- *launch* is a **child** of *coroutineScope*
- the scope **waits for the completion** of all children
- in case of a crash the scope **cancels** all children
- the suspend function has **no leaks**

Exceptions

```
3  fun main() = runBlocking { this: CoroutineScope
4      val handler = CoroutineExceptionHandler { _, exception ->
5          println("Caught $exception")
6      }
7      val supervisor = SupervisorJob()
8      with(CoroutineScope(supervisor)) { this: CoroutineScope
9          val child1 = launch(handler) { this: CoroutineScope
10             println("Child1 is failing")
11             throw AssertionError( detailMessage: "child1 cancelled")
12         }
13         val child2 = launch { this: CoroutineScope
14             child1.join()
15             println("Child1 cancelled: ${child1.isCancelled}")
16             println("Child2 isActive: $isActive")
17             try {
18                 delay(Long.MAX_VALUE)
19             } finally {
20                 println("Finally Child2 isActive: $isActive")
21             }
22         }
23         child1.join()
24         println("Cancelling supervisor")
25         supervisor.cancel()
26         child2.join()
27     }
28 }
```

- *An **exception** other than CancellationException in a coroutine **cancels its parent***
- *A Coroutine**ExceptionHandler** may be passed to the context to replace try /catch blocks*
- *If we want cancellation to be propagated **only downwards** we use **SupervisorJob** or **supervisorScope***

State

```
1   import org.jsoup.Jsoup
2   import java.lang.Thread.sleep
3
4   val countries : List<String> = listOf(...)
134
135 fun google(keyword: String): String {...}
142
143 object Cache {
144
145     private val cache : MutableMap<String, String> = mutableMapOf<String, String>()
146     private val requested : MutableSet<String> = mutableSetOf<String>()
147
148     fun googleWithCache(keyword: String): String {
149         return cache[keyword] ?: if (requested.add(keyword)) {
150             val result = google(keyword)
151             cache.put(keyword, result)
152             requested.remove(keyword)
153             return result
154         } else {
155             sleep( millis: 2000) //wait and retry?
156             return googleWithCache(keyword)
157         }
158     }
159
160 }
```

- shared mutable state
  ➔ share by communicating
- classes/objects
  ➔ coroutines
- synchronization primitives
  ➔ communication primitives

Channels (experimental)

```
137  fun google(keyword: String): String {...}
144
145  val mutex : Mutex = Mutex()
146  val cache : MutableMap<String, String> = mutableMapOf<String, String>()
147
148  fun CoroutineScope.cache(keywords: ReceiveChannel<String>):  ReceiveChannel<String> = produce { this: ProducerScope<String>
149      for(keyword in keywords) {
150          send(cache.getOrElse(keyword) {
151              val result = google(keyword)
152              mutex.withLock { cache[keyword] = result }
153              return@getOrElse result
154          })
155      }
156  }
157
158  fun CoroutineScope.getCountries(): ReceiveChannel<String> = produce { this: ProducerScope<String>
159      for (country in countries) send(country)
160  }
161
162  fun main() = runBlocking { this: CoroutineScope
163      val countries = getCountries()
164      val google = cache(countries)
165      for (i in 1..5){ //get five results
166          println("Result $i: ${google.receive()}")
167      }
168      println("One more... ${google.receive()}")
169  }
```

Actors (class or function)

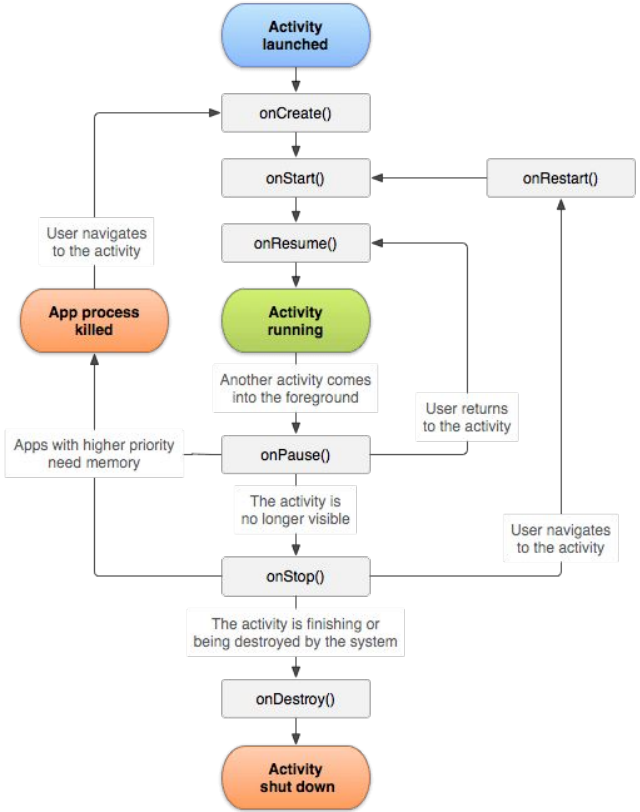
Combination of

- coroutine
- state
- channel

```
144  sealed class CacheAction(val keyword: String)
145  class RetrieveAction(keyword: String, val value: CompletableDeferred<String?>) : CacheAction(keyword)
146  class StoreAction(keyword: String, val value: String) : CacheAction(keyword)
147
148  fun CoroutineScope.cacheActor() = actor<CacheAction> { this: ActorScope<CacheAction>
149      val cache = mutableMapOf<String, String>() //state
150      for (msg in channel) {
151          when (msg) {
152              is RetrieveAction -> msg.value.complete(cache[msg.keyword])
153              is StoreAction -> cache[msg.keyword] = msg.value
154          }
155      }
156  }
157
158  fun CoroutineScope.cache(keywords: ReceiveChannel<String>):  ReceiveChannel<String> = produce { this: Proc
159      for(keyword in keywords) {
160          val cache = cacheActor()
161          val value = CompletableDeferred<String?>()
162          cache.send(RetrieveAction(keyword, value))
163          val retrievedValue = value.await()
164          if( retrievedValue != null) {
165              send(retrievedValue!!)
166          } else {
167              val result = google(keyword)
168              cache.send(StoreAction(keyword, result))
169              send(result)
170          }
171          cache.close()
172      }
173  }
```

Lifecycle

```
4   class LifecycleAwareClass : CoroutineScope { //eg Activity
5
6       //...
7
8       private val job : Job  = Job()
9
10      override val coroutineContext: CoroutineContext
11          get() = job + Dispatchers.Main
12
13      //...
14
15      fun doSomethingImportan() {
16          launch { this: CoroutineScope
17              //important process
18          }
19      }
20
21      //...
22
23      fun onDestroy() { //or similar finalization method
24          //...
25          job.cancel()
26      }
27  }
```



Final Thoughts

- Coroutines are **NOT like threads**
- Force us to **rethink the way we structure** our code
- Intend to **look like sequential code** and hide the complicated stuff
- Resource-wise are **almost free**
- Coroutines are the **cool new thing in the JVM** world

References

- Source Examples
  https://github.com/antonis/CoroutinesExamples

- kotlinlang.org
  https://kotlinlang.org/docs/reference/coroutines-overview.html
- KotlinConf 2018: Exploring Coroutines in Kotlin by Venkat Subramariam
  https://youtu.be/jT2gHPQ4Z1Q
- KotlinConf 2018: Kotlin Coroutines in Practice by Roman Elizarov
  https://youtu.be/a3agLJQ6vt8
- Concurrent Coroutines - Concurrency is not parallelism by Simon Wirtz
  https://kotlinexpertise.com/kotlin-coroutines-concurrency/

Thank you!

Questions?

http://antonis.me/