KOTLIN COROUTINES TIPS & TRICKS

ING. JAN KALÁB, MASTERAPP

KOTLIN COROUTINES 101

KOTLIN

- JVM/Java kompatibilní programovací jazyk.
- *Java na steroidech!*

COROUTINES

- Způsob, jak psát paralelní kód
- Lehká vlákna
- Kooperativní multitasking
- `launch`, `await`, ...

```
1 suspend fun answer() = async {
2   delay(1000) // Heavy calculation, seriously
3   42
4 }
5
6 suspend fun main() {
7         println(answer().await())
8 }
```

SUSPEND MAIN

```
1 fun main(/*args: Array<String>*/) = runBlocking {
2 }
3
4 suspend fun main(/*args: Array<String>*/) {
5 }
```

SUSPEND FUN VS. BUILDER

`Suspend fun` dokud to jde. Až to nejde (`override fun onCreate()`), tak builder.

SUSPEND FUN VS. BUILDER

`Suspend fun` dokud to jde. Až to nejde (`override fun onCreate()`), tak builder.

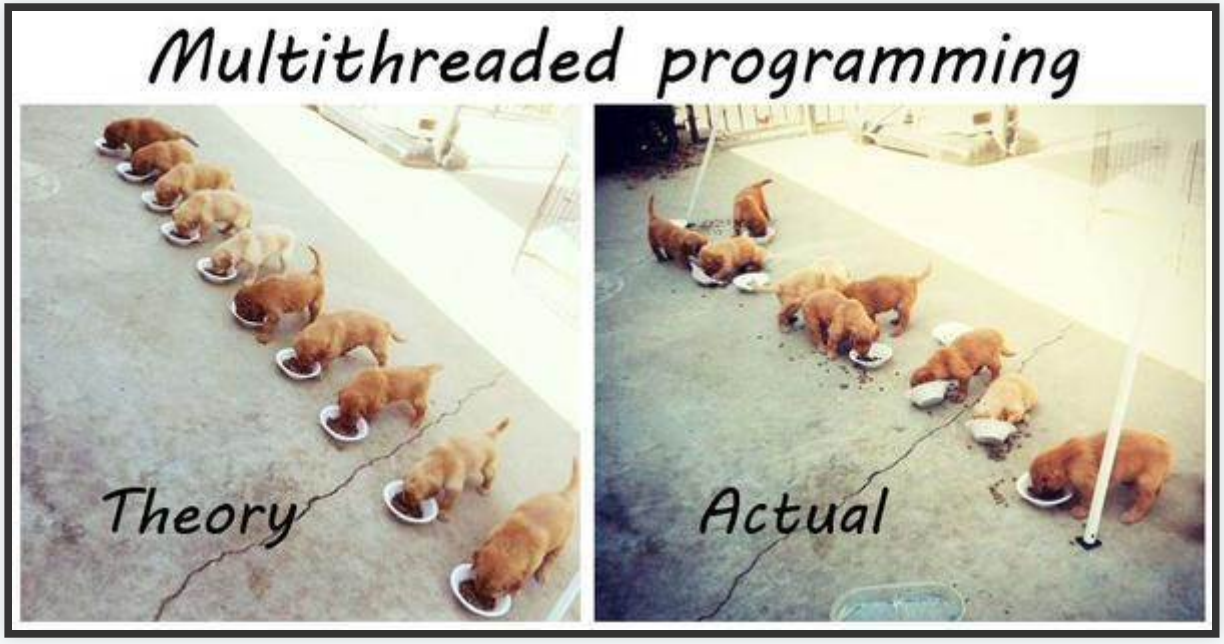
Nebo když jste si **opravdu** jistí, že se má něco provádět paralelně.

SUSPEND FUN VS. BUILDER

`Suspend fun` dokud to jde. Až to nejde (`override fun onCreate()`), tak builder.

Nebo když jste si **opravdu** jistí, že se má něco provádět paralelně.

Nebo když prostě víte co děláte…


Multithreaded programming
Theory
Actual

DISPATCHERS

Všechno `Default` (CPU), co nejméně si odskakovat do `Main` (UI) a `IO`.

```
1 launch(/*Dispatchers.Default*/) {
2   val data = withContext(Dispatchers.IO) {
3     // Read something from disc or internet
4   }
5   val processed = process(data) // Back on Default dispatcher
6   withContext(Dispatchers.Main) {
7     // Show result in UI
8   }
9 }
```

`Default`
: počet jader (nejméně 2)

`IO`
: počet jader (nejméně 64)

- `Default` a `IO` sdílejí pool, nedochází ke context switchům.
- Knihovny (OkHttp) často mají vlastní pooly, netřeba používat `IO`
- `SharedPreferences` při čtení sahá na disk! (~ 80 ms)

CALLBACKS

```
suspend fun OkHttpClient.coEnqueue(request: Request) =
  suspendCancellableCoroutine<Response> { cont ->
    newCall(request).apply {
      cont.invokeOnCancellation { cancel() }
      enqueue(object : Callback {
        override fun onResponse(call: Call, response: Response
          cont.resume(response)
        override fun onFailure(call: Call, e: IOException) =
          cont.resumeWithException(e)
      })
    }
  }
```

```
val data = try {
  http.coEnqueue(request).body.string()
} catch (ioe: IOException) {
  null
}
```

SCOPE

Cokoliv má životní cyklus (`Closeable`), mělo by mít vlastní scope.

```
1 class Foo :
2     Closeable,
3     CoroutineScope by CoroutineScope(Dispatchers.Default) {
4   override fun close() {
5     cancel() // Scope
6   }
7 }
```

SCOPE

Cokoliv má životní cyklus (`Closeable`), mělo by mít vlastní scope.

```
1 class Foo :
2     Closeable,
3     CoroutineScope by CoroutineScope(Dispatchers.Default) {
4   override fun close() {
5     cancel() // Scope
6   }
7 }
```

`GlobalScope` = ZLO!

MAINSCOPE

Běží na `main` vlákně!

```
1 class MyActivity : Activity(), CoroutineScope by MainScope() {
2   override fun onDestroy() {
3     super.onDestroy()
4     cancel() // Scope
5   }
6 }
```

```
1 class MyFragment : Fragment(), CoroutineScope by MainScope() {
2   override fun onDetach() {
3     super.onDetach()
4     cancel() // Scope
5   }
6 }
```

LIFECYCLESCOPE

Běží na `main` vlákně!

```
dependencies {
  implementation 'androidx.lifecycle:lifecycle-runtime-ktx:+'
}
```

```
class MyFragment : Fragment() { // Activity, Service
  fun foo() {
    // Attach/detach
    lifecycleScope.launch {}

    // viewCreated/destroyed
    viewLifecycleOwner.lifecycleScope.launch {}
  }
}
```

VIEWMODELSCOPE

```
dependencies {
  implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:+'
}
```

```
class MyViewModel : ViewModel() {
  fun foo() {
    viewModelScope.launch {
      // Main
    }
  }
}
```

JOBS, CANCELLATION

EXCEPTION BUBBLING

```
1 launch { // ①
2   launch { // ②
3     // Stuff
4   }
5
6   launch { // ③
7     throw Exception()
8   }
9 }
```

1. Výjimka z ③ probublá do rodiče ①.
2. ① nemá žádného rodiče → cancel.
3. ② se ukončí (`CancellationException`).

`async` si drží výjimku až do `await()`.

```
try {
  launch {
  }
}

launch {
  try {
  }
}
```

CANCELLATION PROPAGATION

```
1  val job = launch {
2    launch {
3      // Stuff
4    }
5    launch {
6      // Stuff
7    }
8  }
9
10 job.cancel()
11 //job.cancelChildren()
12
13 launch(job) {
14   // Nope!
15 }
```

- Cancel rodiče ukončí i jeho potomky.
- Zrušený `Job` už nemůže mít žádné potomky!

SUPERVISORJOB

Výjimka v potomkovi neukončí rodiče.

```
1 val supervisor = SupervisorJob()
2
3 launch(supervisor) {
4   // Stuff
5 }
6
7 launch(supervisor) {
8   throw Exception() // Everything is fine.
9 }
```

Možnost vlastního zpracování výjimek.

ENDLESS LOOP

```
1  launch {
2    while(true) {
3      // Endless, ignores cancellation!
4    }
5  }
6
7  launch {
8    while(isActive) {
9      // Ends with cancellation.
10   }
11 }
```

MULTIPLE PARENTS

```
1 val job1 = Job()
2 val job2 = Job()
3
4 launch(job1 + job2) {
5   // Stuff
6 }
7
8 job2.cancel()
9 job1.isActive // true
```

`cancel` jednoho z rodiču zruší i závislé joby.

ACTOR

Channel s logikou

- Click spamming
- Sensors processing

```
1  val actor = actor<Int>(capacity = Channel.CONFLATED) {
2    consumeEach {
3      delay(1000)
4      println(it)
5    }
6  }
7
8  actor.offer(1)
9  actor.offer(2)
10 actor.offer(3)
```

```kotlin
val actor = actor<Int>(capacity = Channel.CONFLATED) {
  consumeEach {
    delay(1000)
    println(it)
  }
}

actor.offer(1)
actor.offer(2)
actor.offer(3)
```

```
1
3
```

DEBUGGING

```
 1 import kotlinx.coroutines.*
 2
 3 class App : Application() {
 4   init {
 5     System.setProperty(
 6       DEBUG_PROPERTY_NAME,
 7       DEBUG_PROPERTY_VALUE_ON
 8     )
 9   }
10 }
```

```
1 launch(Dispatchers.Main + CoroutineName("foobar")) {
2   println("${Thread.currentThread().name}\tHello")
3 }
```

```
main @foobar#8  Hello
```

KNIHOVNY

RETROFIT 2.6

```
1 interface StuffController {
2   @GET("stuff")
3   suspend fun getStuff(): Stuff
4 }
```

ROOM 2.1

```
implementation 'androidx.room:room-coroutines:2.+'
```

```
1 @Dao
2 interface UsersDao {
3   @Query("SELECT * FROM users")
4   suspend fun getUsers(): List<User>
5 }
```

MOCKK

```
1 coEvery { foo.bar() } returns 42
2 coVerify { foo.bar() }
```