Coroutines

Asynchrone Workflows linear entwickelt

Was sind Coroutinen?

- Suspendierbare, ”unterbrechbare” Funktionen
- Erweiterung von Kotlin in der Version 1.1
  - Neues Keyword “suspend”
  - Coroutines sind aktuell “Experimental”
- Alternative zu Threads
  - 100’000 Coroutinen sind kein Problem
- Werden vom Compiler zu Statemachines kompiliert

Unterbrechbare Funktionen?

- Keine normalen Funktionen
  → Aufruf nur mit speziellen Library-Funktionen möglich

```
val singleValue: Deferred<Int> = async { 1 }
val result: Int = runBlocking { singleValue.await() }
result == 1
```

- Multiple Resultate

```
val manyValues: Sequence<Int> = buildSequence {
    var v = 0
    while(true) {
        yield(v)
        v += 1
    }
}
```

Coroutine Builders

- *'runBlocking'* und *'buildSequence'* sind Coroutine Builder
- Definieren die Art der Coroutine, z.B. 'buildSequence' generiert eine Lazy Sequence
  - 'yield' ist Teil des 'buildSequence' Builders
- Weitere Builder existieren
  z.B. 'launch', 'async', etc.

Unendliche Sequenz

```
val infiniteValues = buildSequence {
    var v = 0
    while(true) {
        yield(v)
        v += 1
    }
}
```

```
infiniteValues.forEach { value ->
    println("Currently $value")
}
```

Alternative zu Callbacks

```
fun needCallback(input: Int, resultCB: (String) -> Unit) {
    val result = input.toString() // time consuming...
    resultCB(result)
}
```

```
suspend fun noNeed(input: Int) = suspendCoroutine<String>{ continuation ->
    val result = input.toString() // time consuming...
    continuation.resume(result)
}
```

Coroutine als Sequenz von Funktionen

```
suspend fun sequence() {
    val a = async {
        Thread.sleep(1000)
        6
    }.await()

    val b = async {
        Thread.sleep(2000)
        7
    }.await()

    println("Answer: ${a * b}")
}
```

```
fun part1(then: (Int) -> Unit) {
  Thread.sleep(1000)
  then(6)
}
fun part2(then: (Int) -> Unit) {
  Thread.sleep(2000)
  then(7)
}
fun sequence() {
  part1 { a ->
    part2 { b ->
      println(
        "The answer is ${a + b}"
      )
    }
  }
}
```

Continuation passing style - CPS

Keyword “suspend” impliziert einen versteckten Parameter – die Continuation:

```
suspend fun sequence() { … }
```

```
fun sequence(c: Continuation<Unit>) {
    …
    UI_ThreadPool.submit(c)
}
```

Coroutine Context

- Thread Auswahl

```
launch(UI) {
    sequence()
}
```

→ Führt die Coroutine auf dem UI Thread aus

- Zugriff auf “Coroutine-Lokale” Variablen

```
class AuthUser(val name: String) :
AbstractCoroutineContextElement(AuthUser) {
    companion object Key : CoroutineContext.Key<AuthUser>
}
…
async(UI + AuthUser("me")) {
    val user = coroutineContext[AuthUser]?.name
}
```

Vorteile von Coroutinen

- Benötigen wenig Resourcen
- Sehen aus wie Funktionen, sind aber Statemachines
  → Wenig Overhead
- Erlaubt asynchrones, imperatives Programmieren
  → jegliche Konstrukte funktionieren wie gewohnt:
  - Try-catch, try-with-resources
  - Loops
  - Etc.
- Alternative zu Threads

Nachteile von Coroutinen

- Experimenteller Status in Kotlin 1.1 → Noch nicht fix in der Sprache
- Code sieht linear aus, ist aber nicht-linear

Beispiel: Login Prozess

```
Handler().postDelayed({
    object : AsyncTask<Void, Void, Void>() {
        public override fun doInBackground(vararg voids: Void): Void? {
        …
        }
    }.doInBackground()
}, 1000)
```

```
launch(UI) {
    delay(1000, TimeUnit.MILLISECONDS)
    …
}
```

Beispiel: Login blockieren

```
suspend fun verifyPassword(password: String, userEmail: String) : Boolean
{
    val passwordValid = password == "123456"
    if(!passwordValid) {
        delay(10, TimeUnit.SECONDS)
    }
    return passwordValid
}
```

Aufgabe

Login Asynchron Validieren

Aufgabe: Asynchron Usernamen validieren

- Tag der Aufgabe auschecken

```
> git checkout step_08_coroutine_exercise
```

- LoginPresenter#doLogin” um asynchrone Validierung des Usernamen erweitern:
  - Eine suspendierbare Funktion ‘validateUsername’ erstellen und darin mit einem ‘delay’ eine langsame Validierung simulieren

RxJava oder Coroutinen?

RxJava

- Erlaubt einfachere Restrukturierung
- Viele vorgefertigte Operationen
- Hohes Abstraktionlevel

Coroutine

- Linearer Code
- Einfache Logik
- Fehlerbehandlung wie normaler Code
- Eher Low-Level Abstraktion

Was meint Ihr dazu?

Quellen

- http://github.com/Kotlin/kotlinx.coroutines/blob/master/core/kotlinx-coroutines-core/README.md
- https://de.wikipedia.org/wiki/Continuation-Passing_Style
- http://akarnokd.blogspot.ch/2017/09/rxjava-vs-kotlin-coroutines-quick-look.html