Asynchronous programming with Kotlin coroutines

Konrad Kamiński
Allegro.pl

```
fun sendAuditMessage(userId: String) {
  val accountId = getUserAccountId(userId)
  val productCount = getProductCount(accountId)
  sendMessage("User $userId has got $productCount products")
}
```

```
fun getUserAccountId(userId: String): String
fun getProductCount(accountId: String): Int
fun sendMessage(message: String): Unit
```

```
fun sendAuditMessage(userId: String, callback: () -> Unit) {
  getUserAccountId(userId) { accountId ->
    getProductCount(accountId) { productCount ->
      sendMessage("User $userId has got $productCount products") {
        callback.invoke()
      }
    }
  }
}
```

← Callback hell

```
fun getUserAccountId(userId: String, callback: (String) -> Unit): Unit
fun getProductCount(accountId: String, callback: (Int) -> Unit): Unit
fun sendMessage(message: String, callback: () -> Unit): Unit
```

```
fun sendAuditMessage(userId: String)=
  getUserAccountId(userId)
    .thenCompose { accountId ->
      getProductCount(accountId)
    }
    .thenAccept { productCount ->
      sendMessage("User $userId has got $productCount products")
    }
```

Combinators

```
fun getUserAccountId(userId: String): CompletableFuture<String>
fun getProductCount(accountId: String): CompletableFuture<Int>
fun sendMessage(message: String)
```

```
suspend fun sendAuditMessage(userId: String) {
    val accountId = getUserAccountId(userId)
    val productCount = getProductCount(accountId)
    sendMessage("User $userId has got $productCount products")
}
```

```
suspend fun getUserAccountId(userId: String): String
suspend fun getProductCount(accountId: String): Int
suspend fun sendMessage(message: String): Unit
```

```
suspend fun sendAuditMessage(userId: String): Unit

fun sendAuditMessage(userId: String, callback: Continuation<Unit>): Any?

interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}

val COROUTINE_SUSPENDED: Any = Any()
```

```
 suspend fun sendAuditMessage(userId: String) {
0    val accountId = getUserAccountId(userId)
1    val productCount = getProductCount(accountId)
2    sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
```

```
 suspend fun sendAuditMessage(userId: String) {
0    val accountId = getUserAccountId(userId)
1    val productCount = getProductCount(accountId)
2    sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
}
```

```
 suspend fun sendAuditMessage(userId: String) {
0    val accountId = getUserAccountId(userId)
1    val productCount = getProductCount(accountId)
2    sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
```

```
 suspend fun sendAuditMessage(userId: String) {
0   val accountId = getUserAccountId(userId)
1   val productCount = getProductCount(accountId)
2   sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
```

```
 suspend fun sendAuditMessage(userId: String) {
0   val accountId = getUserAccountId(userId)
1   val productCount = getProductCount(accountId)
2   sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
```

```
 suspend fun sendAuditMessage(userId: String) {
0    val accountId = getUserAccountId(userId)
1    val productCount = getProductCount(accountId)
2    sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
```

```
 suspend fun sendAuditMessage(userId: String) {
0    val accountId = getUserAccountId(userId)
1    val productCount = getProductCount(accountId)
2    sendMessage("User $userId has got $productCount products")
3}

class StateMachine(val userId: String,
  val sendAuditMessageCallback: Continuation<Unit>): Continuation<Any?>{
  var label: Int = 0
  var accountId: String? = null
  var productCount: Int? = null}

when (sm.label) {
    0 -> { sm.label = 1; getUserAccountId(userId, sm) }
    1 -> { sm.label = 2; sm.accountId = value as String;
           getProductCount(sm.accountId!!, sm) }
    2 -> { sm.label = 3; sm.productCount = value as Int;
           sendMessage("User ${sm.userId} has got ...", sm) }
    3 -> { sm.sendAuditMessageCallback.resume(Unit) }
```

```kotlin
suspend fun myfun(param: Int): String =
    suspendCoroutine { callback: Continuation<String> ->
      // usually we'll invoke callback methods in a different thread
      // if we want to return a value
      callback.resume("Result of myfun with $param")

      // if we want to throw an exception
      callback.resumeWithException(Exception("Something went wrong"))
}

suspend fun <T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
```

```
launch {
  //suspending functions can be invoked here
}

fun launch(block: suspend CoroutineScope.() -> Unit): Job

interface Job {
   fun cancel(cause: Throwable? = null): Boolean
   suspend fun join()
}

interface CoroutineScope {
   val isActive: Boolean
   val coroutineContext: CoroutineContext
}
```

```
runBlocking {
  //suspending functions can be invoked here
}

fun <T> runBlocking(block: suspend CoroutineScope.() -> T): T
```

```
val result = async {
  //suspending functions can be invoked here
}

fun <T> async(block: suspend CoroutineScope.() -> T): Deferred<T>

interface Deferred<out T>: Job {
    suspend fun await(): T
}
```

```
val completableFuture: CompletableFuture<T> = future {
  //suspending functions can be invoked here
}

suspend fun <T> CompletionStage.await (): T
```

```
val single: Single<T> = rxSingle {
  //suspending functions can be invoked here
}

suspend fun <T> SingleSource.await (): T
```

Things not covered

- `CoroutineContext, launch(UI) {…},`
- `Mutex (lock/unlock),`
- Channels/Actors (`send/receive`),
- Interoperability with Reactor, Guava, Quasar, …
- and many more...

Where to find more information

`github.com/Kotlin/kotlinx.coroutines`

`kotlinlang.org/docs/reference/coroutines.html`

`Roman Elizarov talks`

Thank you

Konrad Kamiński
Allegro.pl

`github.com/konrad-kaminski/spring-kotlin-coroutine`