Coroutine Nedir?

Coroutine (Co+ Routines= Cooperation + Routines yani fonksiyonlar demek) eşzamanlılık tasarım desenidir. Asenkron olarak kodun çalışmasını sağlamak için Kotlin diline 1.3 versiyonu ile birlikte girmiştir. Uzun süren işlemlerde ana thread'i bloklamamak için kullanılır. Ana thread bloklandığı zaman uygulamamız ANR (Application Not Responding) durumuna geçer.

Temel Özellikleri Şunlardır:

1. Lightweight: Suspension özelliği sayesinde bir thread içerisinde onlarca coroutine çalıştırılabilir. Suspension özelliği ile bu thread coroutine'leri eş zamanlı çalıştırabilir. Suspension ile ilgili detay bilgisi aşağıda anlatılacaktır.

2. Fewer Memory Leak: Coroutine bir scope içerisinde çalıştığında, scope iptal edildiğinde bu scope'a bağlı tüm coroutine'ler de iptal edilmektedir. Memory leak olmaması için kullanıcı bir ekrandan başka bir ekrana geçtiği zaman **Activity** sınıfının destroy metodu içerisinde scope iptal edilmelidir.

3. Jetpack Entegrasyonu: Bir çok Jetpack kütüphanesi coroutine'leri desteklemektedir.

Coroutines kütüphanesini Android projesine eklemek için şu kütüphane eklenmelidir:

`implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:0.22.5"`. Ayrıca gradle.properties dosyasına şu satır eklenmelidir: `kotlin.coroutines=enable`

Örnek:

```
sealed class Result<out R> {
    data class Success<out T>(val data: T) : Result<T>()
    data class Error(val exception: Exception) : Result<Nothing>()
}

class LoginRepository(private val responseParser: LoginResponseParser) {
    private const val loginUrl = "https://example.com/login"

    // Function that makes the network request, blocking the current thread
    fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {
        val url = URL(loginUrl)
        (url.openConnection() as? HttpURLConnection)?.run {
            requestMethod = "POST"
            setRequestProperty("Content-Type", "application/json; utf-8")
            setRequestProperty("Accept", "application/json")
            doOutput = true
            outputStream.write(jsonBody.toByteArray())
            return Result.Success(responseParser.parse(inputStream))
        }
        return Result.Error(Exception("Cannot open HttpURLConnection"))
    }
}
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {
        val jsonBody = "{ username: \"$username\", token: \"$token\"}"
        loginRepository.makeLoginRequest(jsonBody)
    }
}
```

Bu örnekte main thread makeLoginRequest fonksiyonu tamamlanıncaya kadar bloklanır, bu da uygulamamızın sunucudan cevap gelince kadar yanıt vermemesine sebep olur. Coroutine kullanarak bu sorunu şu şekilde çözebiliriz:

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {
        // Create a new coroutine to move the execution off the UI thread
```

```
            viewModelScope.launch(Dispatchers.IO) {
                val jsonBody = "{ username: \"$username\", token: \"$token\"}"
                loginRepository.makeLoginRequest(jsonBody)
            }
            println("Bu satır yukarıdaki kodun bitmesini beklemeden direk ana thread
i&ccedil;erisinde &ccedil;alışır")
        }
    }
```

viewModelScope: ViewModel KTX eklentileri ile birlikte gelmiş olan ön tanımlı bir coroutine scope'tur. Her coroutine bir scope içerisinde çalışmak ZORUNDADIR.

launch: Bir fonksiyondur. Yeni bir coroutine oluşturup, parametresinde belirtilen ilgili dispatcher içerisine body kısmındaki kodları gönderir.

Dispatchers.IO: I/O işemlerini gerçekleştirmek için reserve edilmiş bir thread içerisinde yeni oluşturulan coroutine'in çalışması gerektiğini ifade eder. Yani launch fonksiyonunun oluşturduğu yeni coroutine I/O işlemlerini sağlayan thread içerisinde çalışacaktır.

Launch fonksiyonu içerisinde yer alan kodlar farklı bir thread içerisinde çalışmaya başlarken login fonksiyonu içerisinde yer alan diğer kodlar ise ana thread içerisinde çalışmaya devam eder.

makeLoginRequest fonksiyonu çağıran başka bir fonksiyon bu fonksiyonun bir coroutine içerisinde çalıştırmasını gerektediğini bilmesi gerekmektedir. Bunun için suspended keyword'ü makeLoginRequest fonksiyonunda kullanılmalıdır. Bu sayede bu fonksiyonu çağırdığımızda bir scope içerisine eklememiz gerektiğini anlarız. Yukarıdaki örnekte bir başka problem de launch, Dispatchers.IO dispatcher'ında coroutine oluşturduğu için, network isteğinin cevabı geldiğinde uygulamamızın UI kısmında cevabı göremeyiz. Yani başarılı bir giriş yapıldı mı yoksa hata mı oluştu şeklinde cevabı kullanıcıya göstermemiz gerekmektedir.

makeLoginRequest ve login fonksiyonunu aşağıdaki gibi güncellersek sorunumuz çözülmüş olur:

```
class LoginRepository(...) {
    ...
    suspend fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {

        // Move the execution of the coroutine to the I/O dispatcher
        return withContext(Dispatchers.IO) {
            // Blocking network request code
        }
    }
}
```

LoginViewModel

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {

        // Create a new coroutine on the UI thread
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"

            // Make the network call and suspend execution until it finishes
            val result = loginRepository.makeLoginRequest(jsonBody)

            // Display result of the network request to the user
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

Çalışma mantığı şu şekildedir:

1. Uygulama ana thread içerisinde yer alan View katmanından login() fonksiyonunu çağırır.

2. launch Dispatchers.IO gibi bir parametre almadığı için ana thread içerisinde yeni bir coroutine oluşturur. Bu sayede network cevabı ana thread içerisinde alınmış olur.

3. Yeni oluşan coroutine içerisinde, **loginRepository.makeLoginRequest**() fonksiyonunda bulunan **withContext** blok içerisinde yer alan kodlar çalışmasını bitirene kadar, coroutine durdurulur (suspending). Coroutine suspending olduğunda, ilgili thread içerisinde yer alan diğer coroutine'ler bloklanmadan çalıştırılmaya devam eder.

4. **withContext** tamamlandığı zaman coroutine ana thread içerisinde çalışmasını tekrar başlatır (resume). Ayrıca network'ten gelen değeri de tutar.

Özetle launch fonksiyonu callback gibi görev yapmaktadır. Yani launch kod bloğundan sonra gelen kodlar ana thread içerisinde çalışmaya devam eder, launch içerisindeki kodlar ise çalışmasını tamamladıktan sonra ana thread'e döner.

Exception Handling İşlemi

```
class LoginViewModel(
     private val loginRepository: LoginRepository
): ViewModel() {

     fun makeLoginRequest(username: String, token: String) {
         viewModelScope.launch {
             val jsonBody = "{ username: \"$username\", token: \"$token\"}"
             val result = try {
                 loginRepository.makeLoginRequest(jsonBody)
             } catch(e: Exception) {
                 Result.Error(Exception("Network request failed"))
             }
             when (result) {
                 is Result.Success<LoginResponse> -> // Happy path
                 else -> // Show error in UI
             }
         }
     }
}
```

Suspended Fonksiyonlar

Coroutine temel yapıtaşı olan suspending fonksiyonlardır. Bu fonksiyonlar durdurulup, yeniden başlatılabilen fonksiyonlardır. suspended anahtar kelimesi ile tanımlanır. Uzun süren bir işlem başlatıp, threadi bloklamadan sonucu bekleyebilir. Suspended fonksiyonlar bir coroutine içerisinde çalışmak zorundadır.

```
suspend fun backgroundTask(param: Int): Int {
      // long running operation
}
```

Derleyici, bu kodu aşağıdaki haline dönüştürür:

```
fun backgroundTask(param: Int, callback: Continuation<int>): Int {
    // long running operation
}
```

Continuation tanımı:

```
interface Continuation<in T> {
   public val context: CoroutineContext
   public fun resumeWith(value: Result<T>)
}
```

Her suspending fonksiyon derleyici tarafından bir state machine'e dönüştürülür. Bu state machine'de state'ler suspend caller'ları temsil eder. Bir suspension call'dan önce, sonraki state derleyici tarafından üretilen bir sınıfın field'inde tutulur. Suspending fonksiyon normal bir fonksiyonu çağırabilir; fakat normal fonksiyon suspending bir fonksiyonu çağıramaz.

Coroutine Builder Listesi

- **runBlocking:** Yeni bir coroutine başlatır ve bu coroutine tamamlanana kadar ana thread'i block'lar.

- launch: Yeni bir coroutine başlatır ve **Job** nesnesi dönderir; fakat bu nesne içerisinde dönüş değeri yer almaz. Artık GlobalScope.launch şeklinde kullanmak gerekiyor.

- async: Yeni bir coroutine başlatır ve Deferred<t> nesnesi dönderir. Bu nesnede dönüş değeri ve exception da tutulur. Thread'i bloklamadan await keyword'ü ile dönüş değerini alabiliriz. </t>

Bir coroutine iptal etmek için **cancel()** metodu kullanılır. Bu metod **Job** nesnesinde tanımlanmıştır. Ayrıca join metodu sayesinde coroutine çalışmasını iptal edilen **Job** tamamlanana kadar beklemeye alabiliriz. **cancelAndJoin()** metodu bu iki işlemi yapmayı sağlar.

Örnek:

```
fab.setOnClickListener {
     GlobalScope.launch { doWorld()}
}

suspend fun doWorld(){
     delay(1000L)
     println(&amp;quot;World&amp;quot;)
}
```

Mantıksal olarak async tıpkı launch gibi ayrı bir coroutine başlatır ve diğer coroutine'lerle birlikte eş zamanlı çalışır.

```
val time = measureTimeMillis {
     val one = GlobalScope.async { doSomethingUsefulOne() }
     val two = GlobalScope.async { doSomethingUsefulTwo() }
     println(&amp;quot;The answer is ${one.await() + two.await()}&amp;quot;)
}
println(&amp;quot;Completed in $time ms&amp;quot;)
```

Bu işlem aşağıdaki işlemden 2 kat daha hızlı çalışır; çünkü async kullanıldığında await kullanana kadar işlemler asenkron başlar.

```
val time = measureTimeMillis {
     val one = doSomethingUsefulOne()
     val two = doSomethingUsefulTwo()
     println(&amp;quot;The answer is ${one + two}&amp;quot;)
}
println(&amp;quot;Completed in $time ms&amp;quot;)
```

Not: Async tanımlanan bir fonksiyon suspend'ed bir fonksiyon DEĞİLDİR. Bundan dolayı her yerden çağırabiliriz. Suspended olan bir fonksiyonu her yerden çağıramayız.

Async fonksiyonun await metodunu **runBlocking** içerisinde çağırabiliriz.

Not: Async fonksiyonları diğer programlama dillerinde çok kullanılmasına rağmen, Kotlin'de kullanılması pek tavsiye edilmez.

CoroutineScope nedir?

Coroutine'ler için bir scope tanımlaması yapar. launch, async gibi coroutine builder'lar da aslında birer coroutine scope'tur. Bir veya daha fazla fonksiyonun async bir şekilde çalışması durumunda oluşabilecek bir exception'da tüm coroutine'lerin iptal edilmesini sağlar.

Android uygulamasında bir activity içerisinde asenkron veri getirme, güncelleme gibi işlemleri yaparken memory leak olmaması için, activity destroy edildiğinde, CoroutineScope kullanılması tavsiye edilir.

Aşağıdaki örnekte **coroutineScope** bir suspended fonksiyondur. Genelde birden fazla suspended fonksiyonları gruplamak için kullanılır. Çünkü bu fonksiyonların birinde bir hata meydana gelince diğer coroutine de iptal edilir.

```
suspend fun concurrentSum(): Int = coroutineScope {
     val one = async { doSomethingUsefulOne() }
     val two = async { doSomethingUsefulTwo() }
     one.await() + two.await()
}
```

Android örneği:

```
class MainActivity : AppCompatActivity() {
     private val scope: MainScope() // MainScope CoroutineScope t&amp;uuml;r&amp;uuml;nde bir factory
```

```
    fonksiyondur. [SupervisorJob] ve [Dispatchers.Main] context elemanlarına sahiptir.
         fun showSomeData() = scope.launch { // ana thread i&ccedil;erisinde başlatılır.
                // ... Burada suspending fonksiyonlarını &ccedil;ağırabiliriz ya da diğer coroutine
    builder&#39;leri kullanabiliriz.
                draw(data) // draw in the main thread
         }

         fun destroy() { // destroys an instance of MyUIClass
             scope.cancel() //bu scope i&ccedil;erisinde başlatılan b&uuml;t&uuml;n
    coroutine&#39;leri iptal eder
             // ... diğer kodlar
         }
    }
```

CoroutineContext ve Dispatchers

Coroutine'ler her zaman bir CoroutineContext türünde nesne içerisinde çalışırlar. CoroutineContext bir dizi **Job** nesnesi ve dispatcher içerir. CoroutineDispatcher hangi coroutine'in hangi thread veya thread'ler tarafından çalışacağına karar verir. CoroutineDispatcher bir coroutine çalışmasını belirli bir thread ile sınırlayabilir, bir thread havuzuna iletebilir veya serbest çalışmasına izin verebilir.

launch ve async coroutine build'ler opsiyonel bir coroutine parametresi alır. Bunlar Dispatchers.Default, Dispatchers.IO, Dispatchers.Unconfined ve newSingleThreadContext'tir.

```
launch { // context of the parent, main runBlocking coroutine
     println(&quot;main runBlocking : I&#39;m working in thread
${Thread.currentThread().name}&quot;)
}
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
     println(&quot;Unconfined : I&#39;m working in thread ${Thread.currentThread().name}&quot;)
}
launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
     println(&quot;Default : I&#39;m working in thread ${Thread.currentThread().name}&quot;)
}
launch(newSingleThreadContext(&quot;MyOwnThread&quot;)) { // will get its own new thread
     println(&quot;newSingleThreadContext: I&#39;m working in thread
${Thread.currentThread().name}&quot;)
}
```

Çıktı:

```
Unconfined : I&#39;m working in thread main
Default : I&#39;m working in thread DefaultDispatcher-worker-1
newSingleThreadContext: I&#39;m working in thread MyOwnThread
main runBlocking : I&#39;m working in thread main
```

Not: runBlocking ve launch ikisi de birer normal fonksiyondur. **runBlocking** yeni bir coroutine çalıştırır ve current thread'i coroutine tamamlanıncaya kadar bloklar. launch ise bloklamadan çalıştırır ve işi bitince **Job** nesnesi dönderir.