Coroutines in Kotlin

کروتین از واژه‌ی cooperative routines تشـــکیل شـــده اســـت. یعنی روتین‌هایی که با یکدیگر همکاری می‌کنند. این بدین معناسـت که هنگامی که یک روتین در حال اجراسـت سـایر روتین‌ها با suspend کردن خودشان ارتباط‌شان با cpu ، memory و هر ریسورس دیگری را قطع می‌کنند تا از بلاک کردن آن‌ها جلوگیری شود.

کوروتین ها یک مکانیسـم lightweight و موثر در concurrency هسـتند که می توانند در Kotlin برای نوشتن کدهای ناهمزمان و غیر مسدود کننده استفاده شوند. آن‌ها در مقایسه با مدل‌های threading سنتی مزایای زیادی دارند، از جمله:

- Simplicity: فهم کروتین در مقایسـه با threading API ها آسـان تر اسـت و برای توسـعه دهندگان با هر سطحی از دانش قابل دسترس تر است.

- Efficiency: کروتین ها lightweight هســـتند و کم‌ترین میزان overhead را دارند. بنابراین می‌توان تعداد بسـیار زیادی از آن هارا بدون آنکه روی عملکرد سـیسـتم تاثیر گسـترده‌ای بگذارد ایجاد کرد.

- Asynchrony: کروتین‌ها می‌توانند برای نوشـتن کدهای asynchronous و non-blocking اسـتفاده شـوند. آن‌ها می‌توانند فعالیت‌های طولانی مدت(long-running) را بدون بلاک کردن ترد اصلی سیستم انجام دهند.

- Structured concurrency: کـروتـیـن بـه صـــــورت built-in از قـواعـد Structured concurrency پشـــتیبانی می‌کند. این پترن مطمئن می‌شـــود آغاز، اجرا، پایان یا کنســـل شـدن کروتین‌ها در یک زمان قابل پیشـبینی و در یک مسـیر امن صـورت بپذیرد. این امر باعث می‌شـود تا کروتین‌ها دچار ریسـورس leak نشـوند و از شـر ارورهای رایج در اعمال concurrency در امان بمانند.

- Exception handling: کروتین بـه صـــورت built-in از قـابلیـت Exception handling پشتیبانی می کند. این بدین معناست که به ارورها اجازه می‌دهد تا در سلسله مراتب به سـمت بالا حرکت کنند تا به handler مناسـب خودشـان برسـند. این کار نوشـتن کدی را آسان‌تر می‌کند که در برابر خطاها مقاوم باشد.

چگونه یک فانکشن را به کروتین تبدیل می‌کنیم ؟

به راحتی! با استفاده از کلمه‌ی کلیدی SUSPEND

```
suspend fun backgroundTask(param: Int): Int {

// long running operation

}
```

Under-hood conversion of Suspend by the compiler

قبل از آنکه وارد جزییات کامپایلر شویم اجازه دهید تا با اساس کار کروتین آشنا شویم. یک کروتین در واقع یک lightweight thread اســـت که می‌تواند در یک نقطه‌ی خاص، بدون بلاک کردن ترد، suspend و resume شـود. در واقع ایده‌ی اصـلی کروتین آن اسـت که کدهای asynchronous را به همان روشی بنویسـیم که کدهای synchronous را می‌نویسـیم. این باعث می‌شـود تا از بسـیاری از خطاها در امان باشیم و کدهای async خوانا و قابل درک باشند.

هنگامی یک کروتین suspend می‌شـود به این معناسـت که کروتین اجرای خودش را متوقف کرده و کنترل را به caller خودش بازمی‌گرداند. کروتین به دلایل مختلفی می‌تواند suspend شود:

- انتظار برای آنکه فرایندهای i/o تکمیل شوند
- انتظار برای آنکه یک تایمر به پایان برسد
- انتظار برای آنکه یک کروتین دیگر اجرایش را کامل کند

هنگامی که یک فانکشـــن را با کلمه‌ی کلیدی suspend مارک می‌کنید کامپایلر آن فانکشـــن را به فرم زیر تبدیل می‌کند:

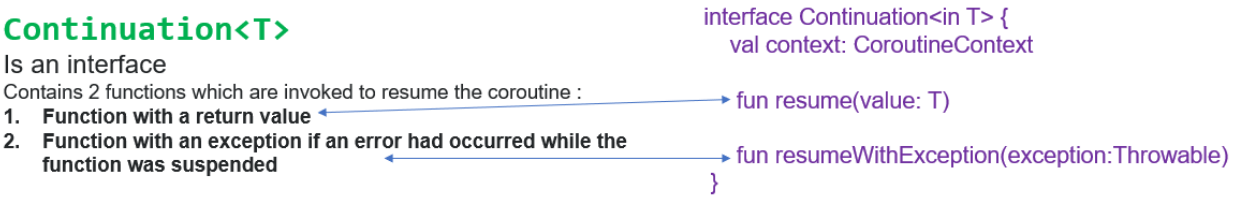
```
fun backgroundTask(param: Int, callback: Continuation<Int>): Int {

// long running operation

}
```

در واقع کامپایلر یک callback از نوع Continuation را به عنوان ورودی تابع در نظر می‌گیرد.

Continuation<T> یک interface است که شامل دوفانکشن است، این فانکشن‌ها هنگامی که یک کروتین resume می‌شود اجرا می‌شوند. یکی از فانکشن‌ها شامل value است و دیگری شامل exception است (برای زمان‌هایی که کروتین در حین suspend شدن با خطا مواجه شده است)



Structured concurrency

یک دیزاین پترن است که به ساده سازی و افزایش قابلیت اطمینان در اجرای فرایند های concurrent کمک می‌کند. این دیزاین پترن اطمینان حاصـــل می‌کند تا تمام فرایندهای concurrent به صـــورت ســاختار یافته و مدیریت شــده اجرا شــوند تا با یکدیگر به تداخل بر نخورند و side effect ای بر جای نگذارند.

این هدف به کمک محدود کردن اجرای فرایند ها در یک ســلســله مراتب خاص (هر فرایند درون یک parent context انجام می شود و تضمین می شود تا قبل از آن که parent اش خاتمه یابد فعالیتش را تمام کند) محقق می‌شود.

Structured concurrency در کاتلین بر اساس یک رابطه‌ی سلسله مراتبی بین کروتین هاست، هر کروتین یک parent دارد. موقعی که یک کروتین جدید start زده می‌شـــود این کروتین به child یک کروتین دیگر تبدیل می‌شود.

این رابطه‌ی سلسله مراتبی باقی است تا زمانی که طول عمر تمام کروتین ها به پایان برسد. لازم به ذکر است هنگامی که یک کروتین parent کنسل می شود تمام child های آن نیز به صورت سلسله مراتبی از پایین ترین قســمت کنســل می‌شــوند این کمک می‌کند تا مطمئن شــویم هیچ leak و یا کروتین معلقی وجود ندارد.

یکی از مزایای مهم Structured concurrency آن است که به کمک محدود کردن اجرای کروتین ها به یک context سلسله مراتبی خاص مطمئن می‌شود تا همه‌ی فرایندها مدیریت شده و ریسورس ها در یک زمان قابل پیشـبینی آزاد شـوند. به کمک این مورد می‌توان بر بسـیاری از مشـکلات فرایندهای concurrent مــانــنــد race condition و deadlock و resource leak غــلــبــه کــرد.

به مثال زیر توجه کنید:

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val job = launch {
        delay(1000)
        println("World!")
    }
    println("Hello, ")
    //job.join() // calling or not calling this line have same o/p
}
// Op => Hello World!
```

جاب در یک کروتین دیگر به نام runBlocking قرار گرفته و خود runBlocking به فانکشـــن main محدود شـده اسـت. در این مثال ابتدا باید job به اتمام برسـد تا runBlocking بسـته شـود و خود runBlocking باید کامل شود تا تابع main بتواند return کند و بسته شود.

این باعث می‌شـود تا مطمئن شـویم هنگامی که تابع main بسـته می‌شـود job و runBlocking به اتمام رسیده است.

Coroutine Cancellation

Cancellation فرایند توقف اجرای یک کروتین اسـت قبل از آن که آن کروتین وظیفه‌ی خود را به پایان برساند. موقعی که یک کروتین کنسل می‌شود اجرایش را از نزدیک ترین cancellation point به پایان می‌رسـاند. یک cancellation point جایی اسـت که کروتین در آن نقطه به دنبال signal توقف می گردد تا در صـورتی که آن را پیدا کرد به اجرا خاتمه دهد(به طور مثال فانکشـن delay می‌تواند نقش یک cancellation point را بازی کند)

یک کروتین به دو روش می‌تواند کنسل شود:

- Explicitly: به کمک فانکشن ()cancel می‌توان یک کروتین را به صورت explicit متوقف کرد. فانکشن ()cancel یک عضو از اینترفیس job ا ست و می‌تواند کروتین را بدون در نظر گرفتن state فعلی آن و به صورت فوری متوقف کند.

```
val job = launch {
    // Do some work here
}
// Cancel the coroutine explicitly
job.cancel()
```

Automatically: یک کروتین هنگامی که parent اش متوقف می‌شـود می‌تواند به صـورت اتوماتیک متوقف شود.

```
val parentJob = Job(

// Launch a child coroutine in the parentJob
val childJob = launch(parentJob) {
    // Do some work here
}

// Cancel the parent coroutine
parentJob.cancel())
```

موقعی که یک کروتین کنسـل می‌شـود، اسـتیت job آن به cancelling تغییر می‌کند و فرزندانش به صورت recursive از پایین ترین قسمت سلسله مراتب شروع به کنسل شدن می‌کنند. در طی فرایند cancelling هر کروتین فرصت این را دارد تا ریسورس های خودش را clean up کند.

توجه شـود که وقتی یک کروتین کنسـل می‌شـود الزاما به این معنا نیسـت که بلاک آن می‌تواند به طور کامل انجام شـود. تنها زمانی می‌توان از اجرای کامل بلاک مطمئن شـد که کروتین به صـورت نرمال complete شده باشد. (پس چگونه می‌توان فرصت آزاد سازی منابع را به دست آورد؟)

How to handle the Coroutine Cancellation?

هنگامی که یک کروتین کنسـل می‌شـود، این نکته حائز اهمیت اسـت که cancellation را به نحوی مدیریت کنیم تا از هر گونه resource leak و خطاهای دیگر جلوگیری شـود. چندین راه برای این امر وجود دارد.

- try-catch-finally: این بلاک می‌تواند برای مدیریت cancellation استفاده شود. بلاک finally می‌تواند برای آزاد سازی ریسورس‌ها استفاده شود زیرا حتی اگر کروتین سیگنال کنسل را دریافت کرده باشد finally همچنان اجرا می‌شود.

```
val job = launch {
    try {
        // Do some work here
    } catch (e: Exception) {
        // Handle the exception
    } finally {
        // Release any resources here
    }
}
```

- withContext(NonCancellable): این فانکشـن می‌تواند برای اجرای یک بلاک از کد هایی استفاده شود که در coroutine context فعلی قابل کنسل نیستند.

- kotlinx.coroutines.delay(): این فانکشـــن یک فانکشـــن cancellable اســـت و هنگامی که ســـیگنال cancel را در یافت می‌کند CancellationException ای را throws می‌کند. ما با دریافت این exception می‌توانیم فرصت آزاد سازی resource ها را به دست آوریم.

```
val job = launch {
    try {
        // Do some work here
        delay(5000)
    } catch (e: CancellationException) {
        // Handle the cancellation exception
    }
}
```

How to ensure that finally block is also executed even when coroutine is cancelled?

در کاتلین کروتین، یک بلاک finally برای اجرای کدهایی به کار می‌رود که باید صـــرف نظر از آن که exception ای اتفاق افتاده اســـت یا خیر اجرا شـــوند. در مواردی که یک کروتین ســـیگنال کنســـل را دریافت کرده، مهم است تا مطمئن شویم که بلاک finally به طور کامل اجرا می‌شود یا خیر

هنگامی که یک کروتین کنسـل می‌شـود، اجرای آن متوقف می‌شـود و یک cancellation exception به سـمت والد آن پرتاب می‌شـود، این exception در سـلسـله مراتب آن قدر بالا می‌رود تا به root برسد.

هنگامی که یک کروتین کنسل می‌شود، بلاک finally قبل از آن که کروتین به صورت حقیقی متوقف شود (رسیدن exception به root در سلسله مراتب) اجرا می شود. این بلاک برای اعمالی مانند آزاد سـازی منابع، بسـتن ارتباط شـبکه یا دیتابیس و ... کاربرد دارد. اما گاهی از اوقات یک یا برخی از این اعمال ممکن اسـت خودشـان cancellable و زمانبر باشـند در چنین شـرایطی وقوع exception در عملکرد آن‌ها می‌تواند سیستم را در شرایط ناپایداری قرار دهد.

برای آن که مطمئن شویم بلاک finally هنگام کنسل شدن کروتین به طور کامل اجرا می‌شود سه راه حل وجود دارد:

- کال کردن ()yield در بلاک finally: هنگامی که تابع ()yield کال می‌شــود کروتین را suspend می‌کند و منتظر می‌شود تا مجدد resume شود. استفاده از yield منجر می‌شود تا مطمئن شویم بلاک finally در هنگام کنسل شدن کروتین حتما اجرا می‌شود. اما چگونه؟
هنگامی که تابع ()yield کال می‌شـود کروتین را suspend می‌کند و منتظر می‌شـود تا مجدد resume شـود. هنگامی که یک کروتین کنسـل می‌شـود نمی‌تواند مجددا resume شـود، اما کال کردن yield در finally منجر می‌شود تا کروتین مجددا در صف ready coroutine ها قرار بگیرد به این ترتیب finally فرصــت پیدا میکند تا قبل از پایان حقیقی کروتین کارش را به پایان برساند.

- اسـتفاده از withContext(NonCancellable) در بلاک finally: این فانکشـن مطمئن می‌شـود که کد درون بلاک آن حتی در صورت کنسل شدن کروتین، اجرا می‌شود. اما چگونه؟
این تابع یک context جدید می سازد که قابل کنسل شدن نیست. استفاده از آن در شرایطی مناسـب اسـت که می‌خواهیم بدون در نظر گرفتن این که کروتین کنسـل شـده اسـت یا خیر بلاکی از کدهارا اجرا کنیم.

```
fun main() = runBlocking {
    val job = launch {
        try {
            println("Coroutine started")
            delay(500)
            throw RuntimeException("Oops!")
            println("RuntimeException is thrown")
        } finally {
            withContext(NonCancellable) {
                delay(100)
                println("Finally block executed inside withcontext")
            }
            delay(10)
            println("Finally block executed outside")
        }
    }
    delay(100)
    job.cancelAndJoin()
    println("Coroutine cancelled")
}
```

توجه داشـــته بـاشـــید کـه قبـل از اجرای کـد clean up (کـد هـای داخـل بلـاک withContext)، cancelAndJoin را در کوروتین اصـــلی فراخوانی می کنیم. این مهم اســـت زیرا فراخوانی withContext (NonCancellable) در حالی که کوروتین اصلی هنوز در حال اجرا است می‌تواند باعث مســدود شــدن نامحدود کوروتین جدید شــود. با لغو کوروتین اصــلی قبل از اجرای کد clean up ، اطمینان حاصـل می کنیم که کوروتین جدید مسـدود نشـده اسـت و می تواند کار clean up را کامل کند.

- اســـتفاده از فانکشـــن ()delay در بلاک try و دریافت CancellationException: به مثال زیر توجه کنید:

```
fun main() = runBlocking {
    val job = launch {
        try {
            println("Coroutine started")
            repeat(10) { i ->
                delay(100)
                println("Iteration $i")
            }
        } catch (e: CancellationException) {
            println("Coroutine cancelled")
        } finally {
            println("Finally block executed")
        }
    }

    delay(500)
    job.cancel()
    println("Cancellation requested")
    job.join()
}
```

در مثال بالا هنگامی که کروتین کنسل می‌شود فانکشن delay یک exception از نوع CancellationException را پرتاب می‌کند. دریافت این exception می‌تواند فرصت مناسبی را در اختیار بلاک finally قرار دهد تا بتواند تسک‌های clean up را بدون در نظر گرفتن اینکه کروتین به صورت normal به اتمام رسیده یا کنسل شده است انجام دهد.