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 اسـتفاده شـود. بلاک finally می‌تواند برای آزاد سازی ریسورس‌ها استفاده شود زیرا حتی اگر کروتین متوقف شده 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 کدهایی اسـتفاده شود که در coroutine context فعلی قابل کنسل نیستند.

kotlinx.coroutines.delay(): این فانکشن یک فانکشن cancellable است و هنگامی که سیگنال cancel را دریافت 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
    }
}
```

.....