Coroutine Basics

코루틴

1. 왜 사용하는지, 왜 필요한지.
2. scope, suspending functions, jobs, context etc.
3. Exception Handling

1. 코루틴이란

쓰레드는 resource intensive 하다. 사용하고, 정리하기 힘들다.

코루틴은 lightweight thread 이고, thread pool 을 사용한다. 사용할만큼 최대한 사용해도 오버헤드가 거의 발생하지 않는다.

코루틴 = Unit of execution

스레드를 재사용한다.

async code, 콜백과 동기화를 간소화한다.

syntax 가 간단하다.

pause, resume 을 어느때나 할 수 있고, 몇 개의 스레드를 사용하건 상관 없다.

2. 기본 개념들

scope : 코루틴을 형성하고 실행시킨다. 라이프사이클 이벤트를 제공한다.

context : 스코프는 코루틴이 돌아가는 context 를 제공한다.

suspending functions : 코루틴 안에서 돌아가는 함수이다. (functions that can be suspended)

Job : 코루틴의 핸들이다. 코루틴의 라이프사이클에 접근할 때 job 을 만들고, 이를 통해 코루틴을 캔슬하는 것과 같은 작업이 가능하다.

Deferred : 코루틴의 미래의 결과이다. 프로그램에게 결과를 기다리라고 알려줄 수 있다.

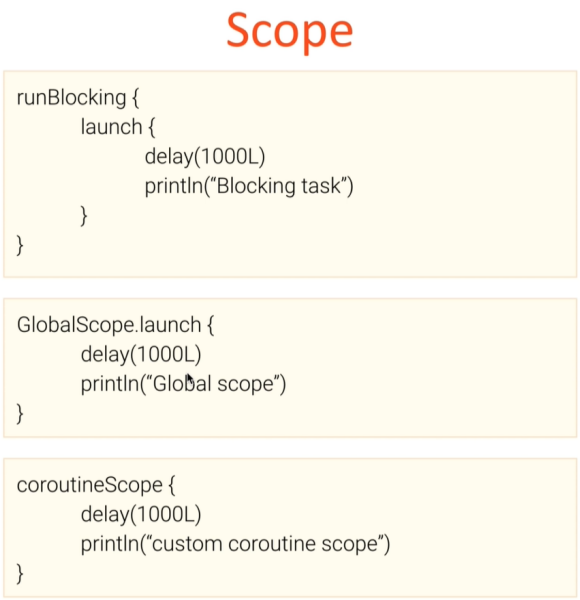
Dispatcher : 코루틴이 돌아가는 스래드들을 매니징한다. 스레드를 할당하고 실행하는 역할.

Error Handling : 어느 에러든 핸들할 수 있게 해준다.

- lifecycle method 를 제공한다.

Scope

- start, stop coroutine 이 가능함.
  - GlobalScope.launch{} : 모든 프래그램이 종료되기 전까지 코루틴이 종료되지 않을 것이다. 코루틴을 간단하게 만들 수 있지만 흔히 사용되지는 않는다.
  - runBlocking : 코드의 실행을 멈추고, 코루틴을 실행할 때 사용한다. 스코프를 시작할 때 많이 사용할 것이다.
  - coroutineScope {} : 코루틴의 자식들이 모두 완료 될때까지 유효하다.




```
import kotlinx.coroutines.*

fun main() {

    println("Program execution will now block")
    //다른 UI 스레드가 없기 때문에 Main 에서 한다.
    runBlocking {
        launch {
            delay(1000L)
            println("Task From runBlocking")
```

```
        }

        GlobalScope.launch {
            delay(500L)
            println("Task From Global Scope")
        }

        coroutineScope{
            launch {
                delay(1500L)
                println("Task From Coroutine Scope")
            }
        }
    }

    println("Program execution will now continue")
}
```

Context

- Context 와 Scope 는 비슷하다.
- set of data that relates to the coroutine.
- 모든 코루틴은 연관된 context 가 있다. 코루틴이 생성되면 그에 상응하는 context 를 제공한다.
- Create, Stop ... 하는 것은 Scope 이고, Context 는 코루틴과 연관된 데이터의 셋.
- Context 의 중요한 요소들
  - Dispatcher - 코루틴이 돌아가는 스레드
  - Job - 코루틴 핸들

```
import kotlinx.coroutines.CoroutineName
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        launch(CoroutineName("myCoroutine")){
            println("This is run from
                      ${this.coroutineContext[CoroutineName.Key]}")
        }
    }
}
```

Suspending Functions

- 코루틴 안에서 돌아갈 수 있는 함수이다.
- Callback 을 쉽게 만든다.
- main 프로그램에서 변수에 접근할 수 있다.

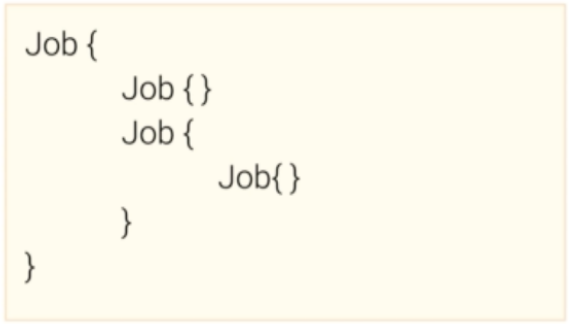
```
suspend fun sayHello(){
  //...
}

GlobalScope.launch{
  sayHello()
}
```

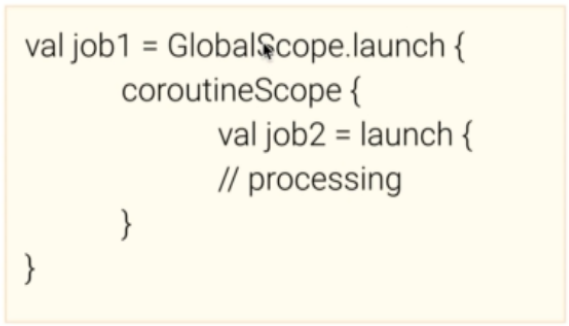
```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

//track number of calls
//메인 어플리케이션의 변수이지만 코루틴에서 접근할 수 있다.
var functionCalls = 0

fun main() {

    GlobalScope.launch {
        completeMessage()
    }
    GlobalScope.launch {
        improveMessage()
    }

    println("Hello, ")

    Thread.sleep(2000L)
    //몇 개의 함수가 이 변수를 접근했는지 확인해본다.
    println("There have been $functionCalls calls so far")
}

//코루틴 간의 동기화의 필요가 없다.
suspend fun completeMessage(){
    delay(500L)
    println("World!")
    functionCalls++ //병렬 처리
}

suspend fun improveMessage(){
    delay(1000L)
```

```
    println("suspend functions are cool!")
    functionCalls++
}
```

Jobs

- 코루틴에 달린 핸들이다.
- `.launch` 콜은 Job 객체를 리턴한다.
- 다른 Job 들의 계층 구조에 속한다.





- lifecycle method 에 job 을 통해 접근할 수 있다.
  - cancel()
  - join()
- 만약 job 을 cancel 하면 해당 job 의 자식들도 cancel 된다.

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
```

```
    runBlocking {
        val job1 = launch {
//            delay(3000L)
            println("Job 1 launched")
            val job2 = launch {
                println("Job 2 launched")
                delay(3000L)

                println("Job 2 is finishing")
            }

            job2.invokeOnCompletion { println("Job 2 Completed") }

            val job3 = launch {
                println("Job 3 launched")
                delay(3000L)
                println("Job 3 is finishing")
            }
        }

        delay(500L)
        job1.invokeOnCompletion {
            println("Job 1 is completed")
        }

        println("Job 1 will be canceled")
        job1.cancel()
    }
}
```

실행 결과 :

```
Job 1 launched
Job 2 launched
Job 3 launched
Job 1 will be canceled
Job 2 Completed
Job 1 is completed
```

Dispatchers

코루틴이 어떤 스레드풀에서 돌아갈지 결정한다. 필요 없다면 만들 필요가 없다.

좀 더 세밀하게 코루틴이 어디서 돌아갈지 설정할 경우 사용한다.

네트워크 통신, 많은 양의 데이터 처리, UI 통신 등과 같은 곳에 사용한다.

- CPU 연산이 많이 소요되는 작업에 사용한다.

```
launch(Dispatchers.Default) {
        // do some CPU intensive processing task here
}
```

- Main 디스패처 : UI driven application 을 업데이트 하는 데 사용된다. 메인 디스패처는 Gradle 에 정의되어야 한다. 안드로이드 앱에서는 자동으로 설정이 된다.
  - 인텔리제이 아이디어 에서는 수동으로 설정해야 한다.
- Default 디스패처 : CPU 연산이 많이 소요되는 작업에 사용한다.
- IO : 네트워크 통신, 파일 쓰기 읽기 등에 사용된다.
- Unconfined 디스패처 : 자신을 부른 디스패처에 자식으로서 또 코루틴을 시작한다.
- newSingleThreadContext("myThread") : 새로운 스레드의 생성을 강제한다.
  - 스레드는 오버헤드가 크기때문에 최대한 사용하지 않는 것이 바람직하다.

async

코루틴을 시작하는 또 다른 방법.

launch 와 비슷하지만, 결과를 반환한다는 점에서 다르다.

result 를 바로 받을 수 없으므로, deferred 의 형태로 반환한다.

- deferred 는 반환된 결과가 있을 것이라는 미래의 약속이다.

결과가 필요한 시점에는 await() 을 호출한다. - blocking call

async

```
suspend fun getRandom() = Random.nextInt(1000)
```

```
val valueDeferred = GlobalScope.async { getRandom() }
... // Do some processing here
val finalValue = valueDeferred.await( )
```

결과가 필요할 때 await() 을 호출하는 것.

withContext

context 를 쉽게 바꿀 수 있도록 해준다. 디스패처 사이에서 자유롭게 바꿀 수 있다.

Main Dispatcher 로부터 이미지를 보여주도록 할 수 있다.

very lightweight

```
launch(Dispatchers.Default) {
        // default context
        withContext(Dispatchers.IO) {
                // IO context
        }
        // back to default context
}
```

Exception Handling

예외 행위는 코루틴 빌더에 의존한다.

launch

- 부모 - 자식 계층구조를 통해서 알린다.
- 예외가 바로 던져질 것이며 바로 job 들이 fail 할 것이다.
- try catch 를 사용하거나, exception handler 를 사용한다.

async

- 예외는 결과가 나올때까지 deferred 된다.
- 만약 결과가 나오지 않으면 예외는 던져지지 않는다.
- try-catch 를 코루틴 안에서 사용하거나, await 를 호출한다.

Exception handler

```
val myHandler = CoroutineExceptionHandler {coroutineContext, throwable ->
        // handle exception
}
```

```
launch(myHandler) {
        // do some task here
        throw IndexOutOfBoundException()
}
```

```
launch(Dispatchers.Default + myHandler) {
        // do some task here
        throw IndexOutOfBoundException()
}
```