
Java to Kotlin
SohHyang Lee

List

1. Java와의 차이점
2. Nullable Check
3. AsyncTask -> Coroutine 변경

1. Java와 차이점

```
mSummaries.get(item)?.setText(result[item]);
```

```
mSummaries[item]?.text = result[item]
```

- 표현 방식의 차이

  소스 코드의 길이가 줄어들고 가독성이 높아짐

  코드 끝에 ; 를 사용 안해도 됨

- gradle 의 환경설정이 필요

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.0'
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.preference:preference:1.1.1'
    implementation project(':WorkpathLib')
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.multidex:multidex:2.0.0'
    implementation 'com.google.android.material:material:1.0.0'
}
```

- kotlin library를 이용해 다음과 같이 import 하면 findViewById 사용 필요 없이 바로 사용 가능

2. Nullable Check

!! 는 사용하지 말자

- !!의 무분별한 사용으로 실질적인 null 체크를 하지 못하는 것을 방지

Null Available

- 다음과 같은 방법으로 Null Check 진행

- *Null available*
- *Safe call ?. / Elvis Operator ?:*
- *Run*
- *Let*

* - *Lateinit / Lazy properties*

null이 아니게 보장하는 방법

1) Null 이 사용되지 않는 경우
- onCreate, onResume에서 반드시 초기화가 되는 변수라 null 일 수가 없다
- lateinit은 var, lazy properties는 val 을 초기화 해줄 때 사용

-*Lateinit*

-*Lazy properties*

2) Null 이 사용되는 경우

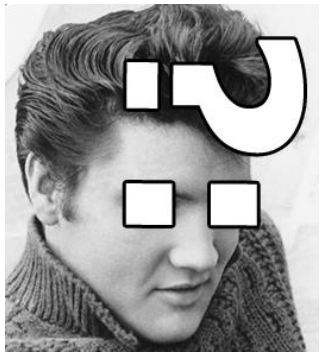
- 다음과 같은 방법으로 Null Check 진행

-*Null available : ?*

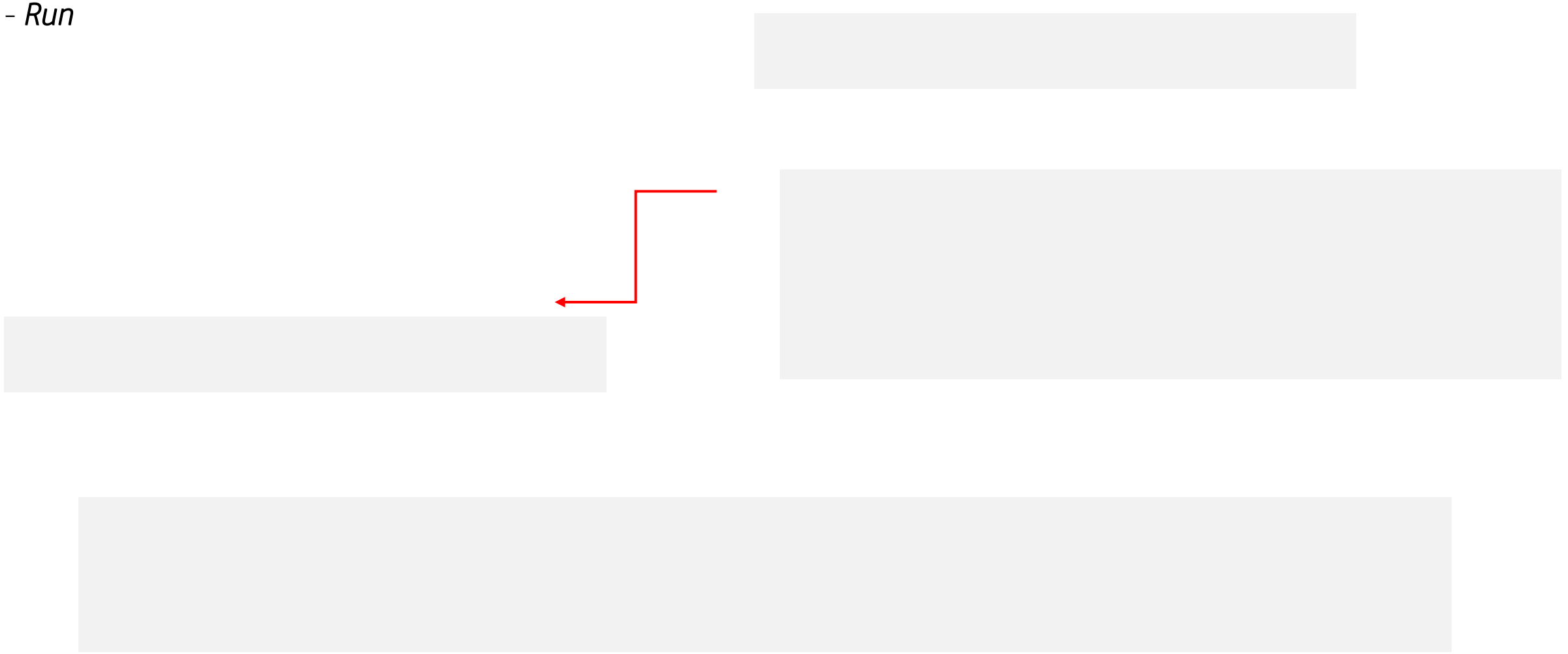
-*Safe call ?.*

```
if (mAlertDialog != null) {
      dismiss();
   } else {
      mAlertDialog = null;
   }
}
```

-*Elvis ?:*



- *Run*



- 여러 값을 계산하고 지역 변수의 범위를 지정할 때 사용
- switch -> when 으로 변경
- aynsc를 주체로 한 return, code generator가 자동생성

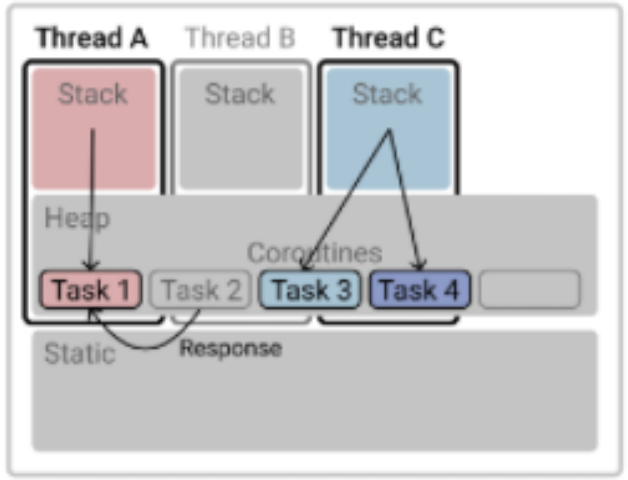
- *Let*

- 지정된 값이 null이 아닌 경우에 코드를 실행해야 하는 경우
- 단일 지역 변수의 범위를 제한 하는 경우
- Nullable 객체를 다른 Non Nullable 객체로 변환해서 사용하려는 경우 (null 경우에는 아예 이 부분이 실행되지 않음)

3. AsyncTask is deprecated in Android

Need to change coroutine in Kotlin

- Thread/AsyncTask/Rx background 작업을 대신 할 수 있는 경량스레드
- Thread 작업 단위 : Thread / Coroutine 작업 단위 : Object
- 하나의 Thread에 여러 개의 Coroutine들이 동시에 실행 될 수 있다.
- 한 Thread 안에서Coroutine 작업 1과 작업 2 두 가지가 있을 때 전환에 있어 단일 Thread 위에서 Coroutine object 들의 객체들만 교체하기 때문에 OS 레벨의 Context Switching이 필요 없음 -> 그래서 경량 스레드라고 불림
- Thread A와 Thread C에서의 예처럼 다수의 스레드가 동시에 수행 될 때 Coroutine switch는 context switch가 일어나야 하기 때문에 다수의 Thread를 사용 하는 것보다 단일 Thread 에서 여러 Coroutine Object들을 실행하는 것이 좋음



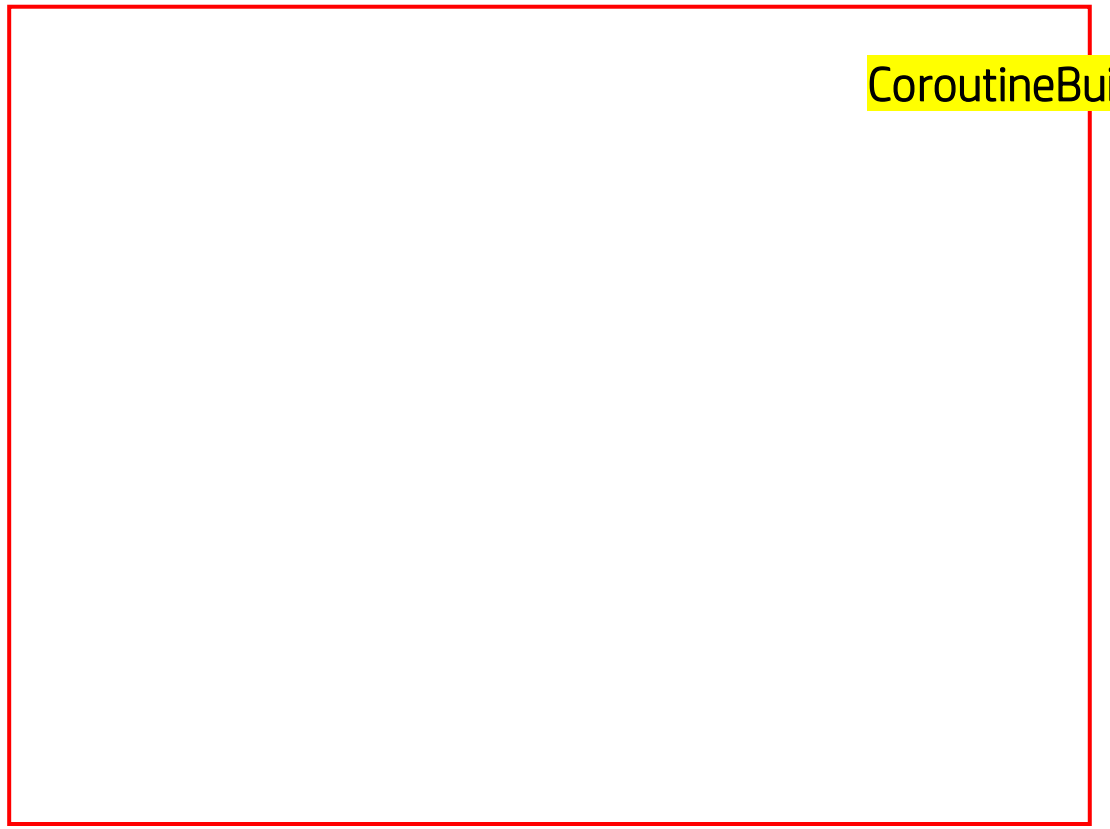
사용법

CoroutineContext

CoroutineDispatcher

CoroutineScope

CoroutineBuilder



Coroutine Scope

CoroutineScope

- launch, async
- launch : job return
- async : deferred return

  반환된 job과 deferred 객체를 이용해 coroutine 제어 가능

```
class MainActivity : AppCompatActivity(), CoroutineScope {

    private val job = SupervisorJob()

    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Main + job

    override fun onDestroy() {
        super.onDestroy()
        coroutineContext.cancelChildren()
    }

    fun loadData() = launch {
        // code
    }
}
```

GlobalScope

- Application의 생명주기를 따라감
- GlobalScope로 실행된 코루틴들은 실행된 scope에 연관되지 않고 독립적으로 동작하기 때문에 scope가 취소 되어도 영향을 받지 않음
- Android 환경에서는 CoroutineScope를 활용해 Android Lifecycle에 맞게 사용하는 걸 권장

Coroutine Dispatchers

Dispatchers.Main

- UI(Main) Thread에서 동작

Dispatchers.Default

- CPU 사용량이 많은 작업에 사용, Main Thread 에서 사용하기엔 긴 작업들에 적합

Dispatchers.IO

- 네트워크 / 디스크(파일) 작업에 사용하는 방식으로 File의 읽기, 쓰기 및 소켓 읽기, 쓰기에 적합

Dispatchers.Unconfined

- caller thread에서 시작되지만 suspend 되었다가 재시작 하면 적절한 thread에 재할당 되어 재시작
- 특정 스레드로 지정되어 처리되어야 하는 경우에는 사용하지 않음


keep reinventing