Kotlin Coroutine - Concurrency Made Simple for Android

Trung

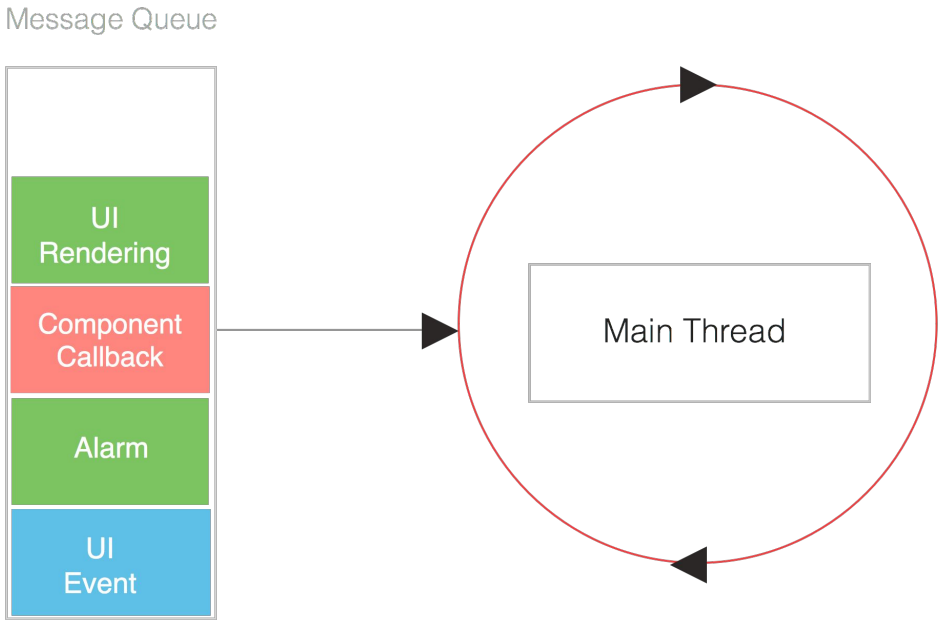
Growth Session #24 - May 16-17 2019

Intro

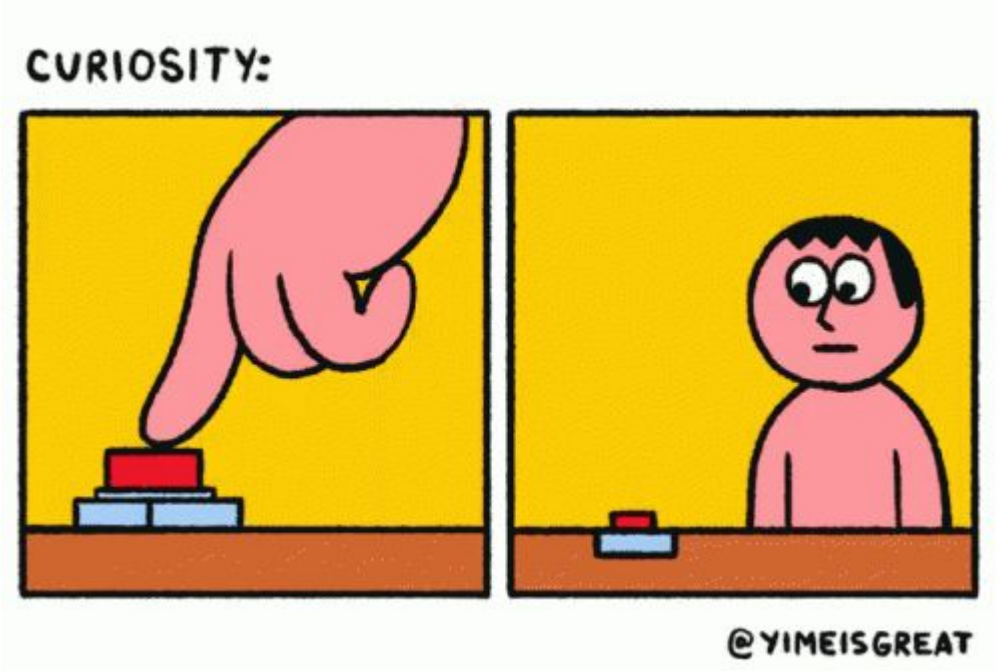
- Android system starts a new Linux process for the application with a single thread of execution.
- Default: All components of the same application run in the same process and thread (called the "main" thread).

The “Main/UI Thread”

It is in charge of dispatching events to the appropriate user interface widgets, including drawing events.



Touching a button



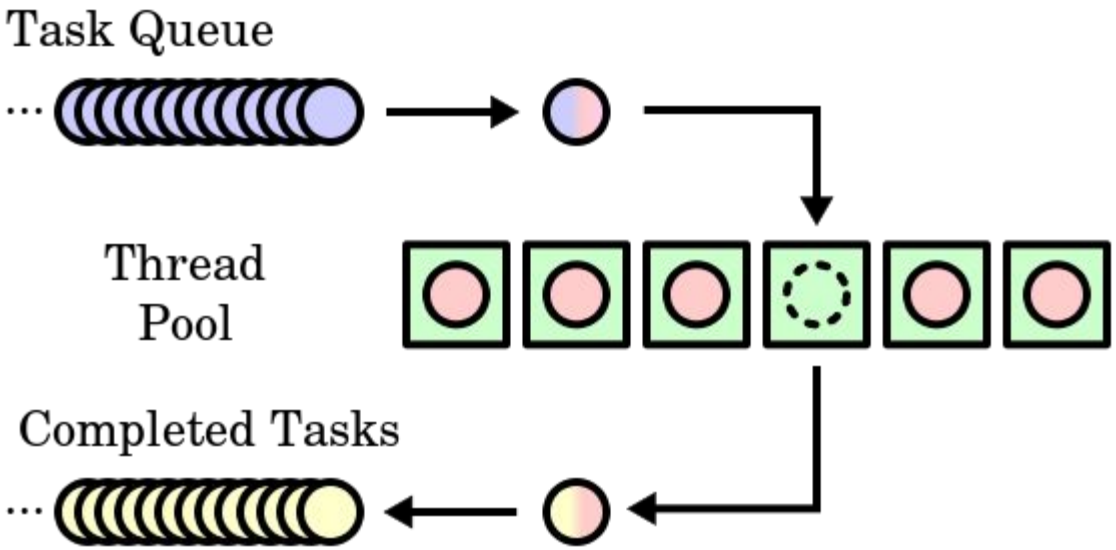
Defining a task that run on a different Thread

```
fun onClick(v: View) {
    Thread(Runnable {
        // a potentially time consuming task
        val bitmap = processBitMap("image.png")
        imageView.post {
            imageView.setImageBitmap(bitmap)
        }
    }).start()
}
```

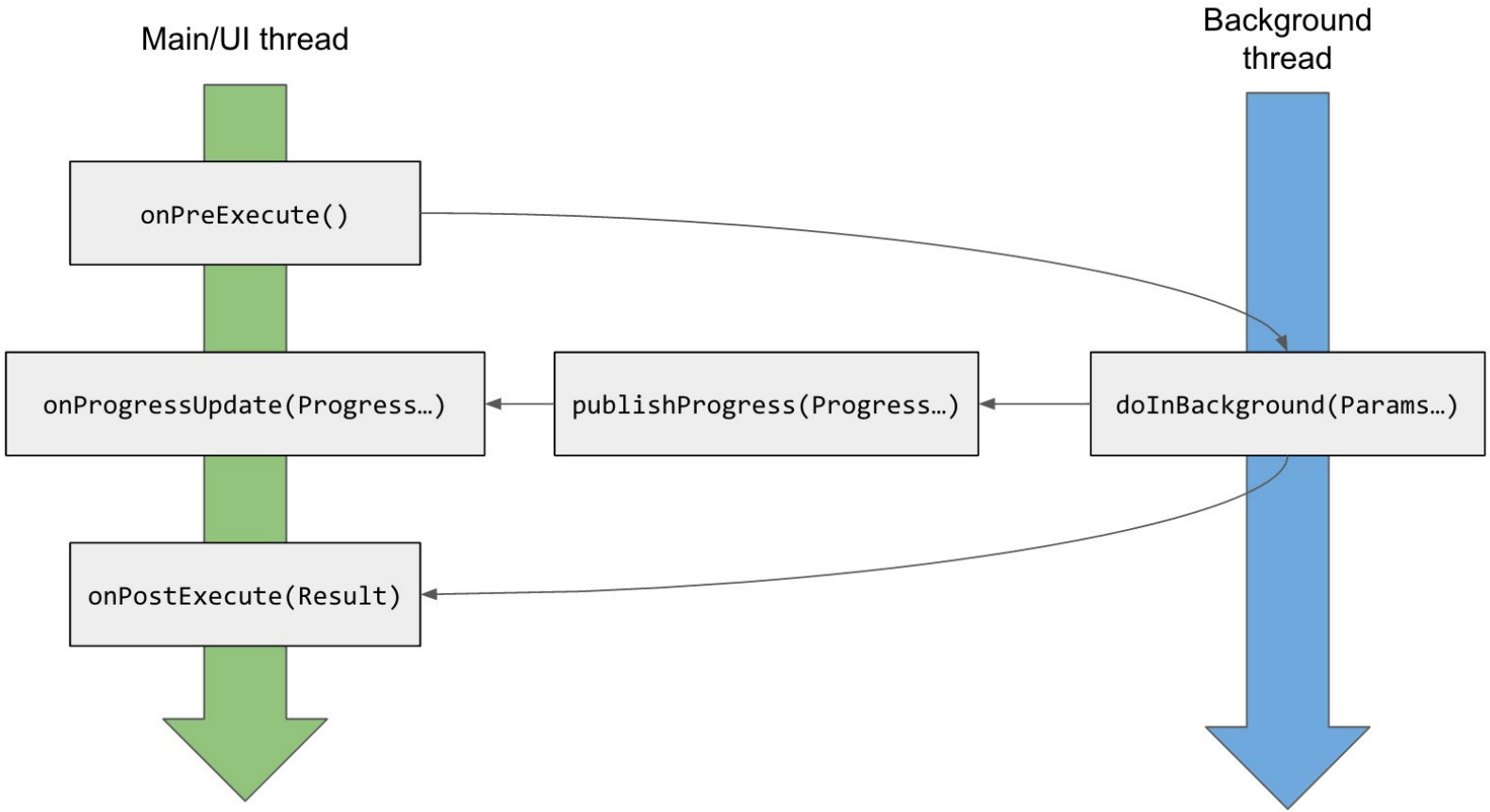
Why wouldn’t we create and manage Thread by ourselves

- Creating new Thread is not cheap (~4Mb)
- If garbage collection occurs in the middle of an intensive processing loop, it would cause A(pplication) N(ot) R(esponding)

1. ThreadPool



The idea of AsyncTask flow



2. Using AsyncTask

```
class DownloadFilesTask : AsyncTask<URL, vararg Integer, Long> {
     protected fun doInBackground(url: URL): Long {
         var totalSize = 0
         // Heavy calculation with the url!!
         return totalSize
     }

     protected fun onProgressUpdate(progress: vararg Integer) {
         setProgressPercent(progress[0])
     }

     protected fun onPostExecute(result: Long) {
         showDialog("Downloaded " + result + " bytes")
     }
 }

fun main() {
  DownloadFilesTask().execute(url)
}
```

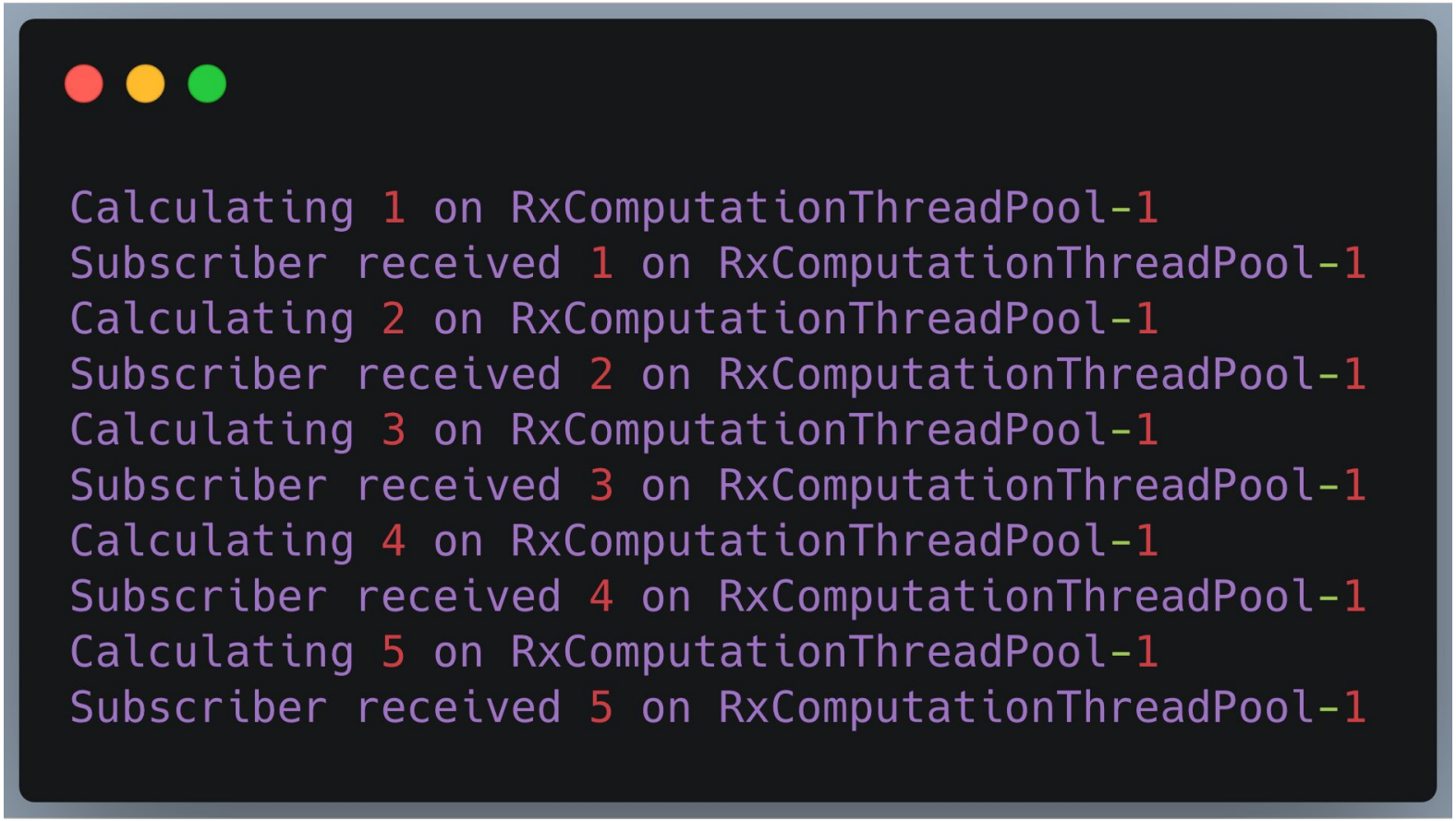
RxJava

- RxJava is a Java VM implementation of Reactive Extension
- Helps you to compose asynchronous and event-based programs by using observable streams.
- Main components:
  - **Observable**: event emission source.
  - **Observer**: receiver of data stream emitted by Observable.
  - **Scheduler**: instrumentation tool to manage threading of this whole pub/sub process.

A simple example

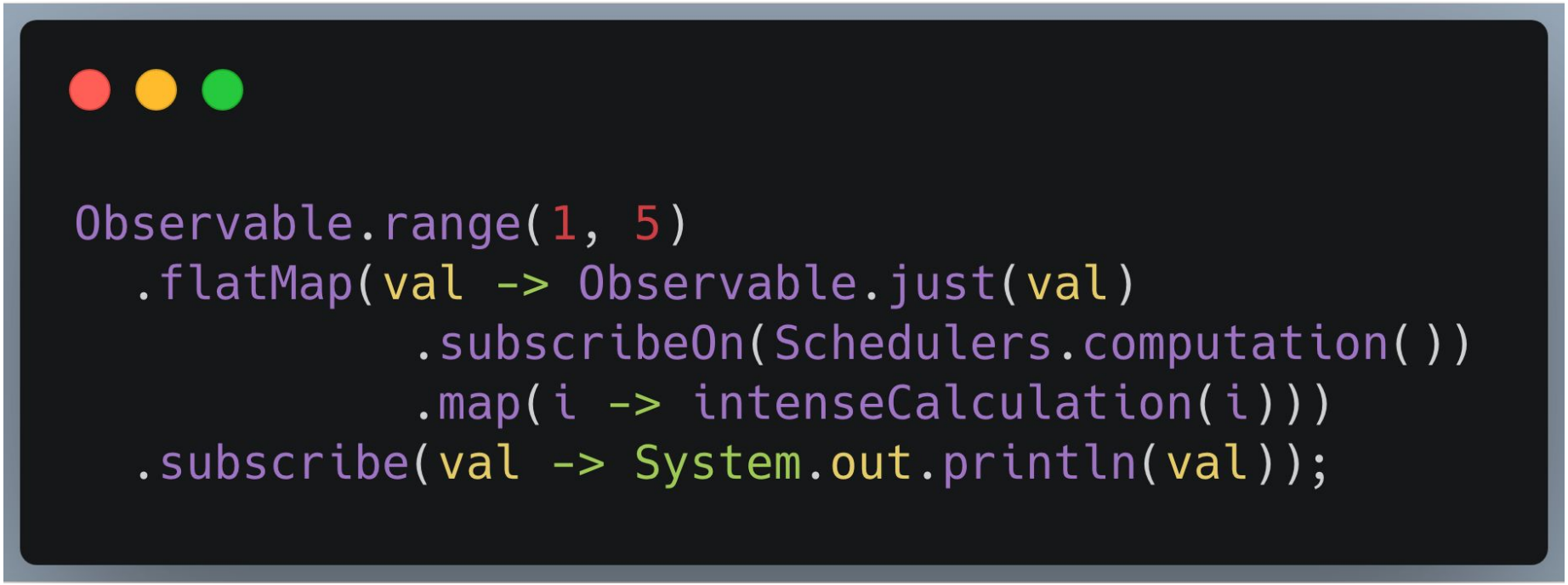
```
Observable.range(1, 5)
    .subscribeOn(Schedulers.computation())
    .map(i -> intenseCalculation(i))
    .subscribe(val -> System.out.println("Subscriber received "
                + val + " on "
                + Thread.currentThread().getName()))
```

Result



```
Calculating 1 on RxComputationThreadPool-1
Subscriber received 1 on RxComputationThreadPool-1
Calculating 2 on RxComputationThreadPool-1
Subscriber received 2 on RxComputationThreadPool-1
Calculating 3 on RxComputationThreadPool-1
Subscriber received 3 on RxComputationThreadPool-1
Calculating 4 on RxComputationThreadPool-1
Subscriber received 4 on RxComputationThreadPool-1
Calculating 5 on RxComputationThreadPool-1
Subscriber received 5 on RxComputationThreadPool-1
```

Make it more like “parallelism”



```
Observable.range(1, 5)
   .flatMap(val -> Observable.just(val)
                .subscribeOn(Schedulers.computation())
                .map(i -> intenseCalculation(i)))
   .subscribe(val -> System.out.println(val));
```

Result

```
Calculating 1 on RxComputationThreadPool-3
Calculating 4 on RxComputationThreadPool-2
Calculating 3 on RxComputationThreadPool-1
Calculating 2 on RxComputationThreadPool-4
Calculating 5 on RxComputationThreadPool-3
```

Kotlin Coroutine

- The idea is suspendable computations, i.e. the idea that a function can suspend its execution at some point and resume later on.
- To the developers, it's like writing non-blocking code is essentially the same as writing blocking code
- Main components:
  - CoroutineContext and Dispatcher: mainly for defining on where the coroutine runs on.
  - The *launch { }* block: kick-off the coroutine.
  - The *suspend { }:* where the execution jobs are suspended.
- Lightweight.

Sample

```
fun postItem() {
    launch {
        val tokenResponse = login(username, pin, type)
        processPost(post)
    }
}

suspend fun login(username: String, pin: String, type: String): Token {
    // makes a request and suspends the coroutine
    return suspendCoroutine { /* ... */ }
}
```

How it works

```
// Dispatchers.Main
suspend fun fetchDocs() {
    // Dispatchers.IO
    val result = get("developer.android.com")
    // Dispatchers.Main
    show(result)
}
// look at this in the next section
suspend fun get(url: String) = withContext(Dispatchers.IO){/*...*/}
```

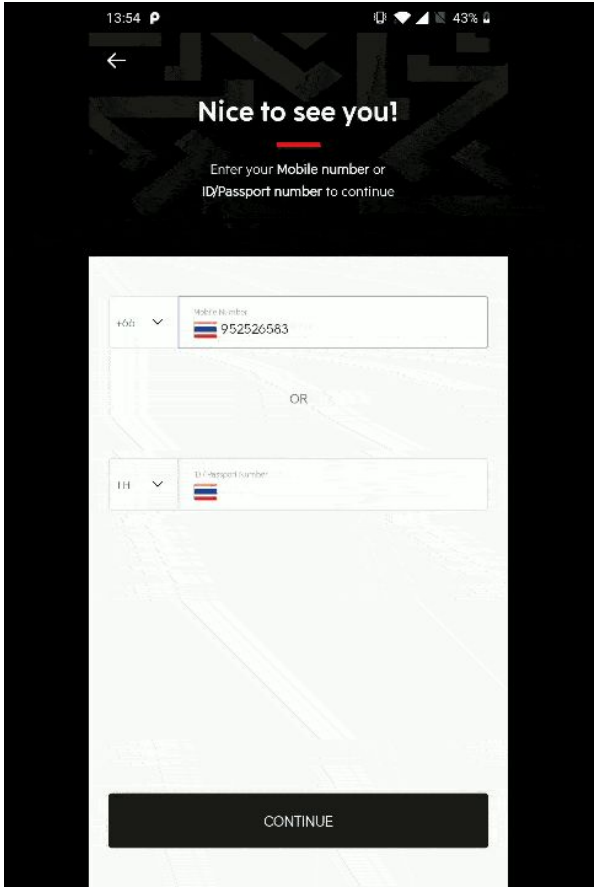
How it works


suspend fun fetchDocs() {
val docs = get("…")
show(docs)
}
suspend
resume
Main Thread
[stack]

From Android-Ktx: LifecycleScope

```
lifecycleScope.launchWhenStarted {
    try {
        // Call some suspend functions.
    } finally {
        // This line might execute after Lifecycle is DESTROYED.
        if (lifecycle.state >= STARTED) {
        // Here, since we've checked, it is safe to run any
        // Fragment transactions.
        }
    }
}
```

In The 1 app - Login flow



The old way

```
mRepository.authenticate(userName, pin, type, new CallBackApi<TokenModel, Throwable>() {
                @Override
                public void onSuccess(TokenModel data) {
                    if (viewCallback == null) return;

                    // store the token, process to the next step
                    nextStep()
                }

                @Override
                public void onError(int httpCode, String errorCode, Object errorObject) {
                    hideProgress();
                    // show error
                }

                @Override
                public void onExpired() {
                    hideProgress();
                    // show error
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
            hideProgress();
            if (viewCallback == null) return;
        }
```

In The 1 App

```
authRepo.requestAccessToken(userName, pin, type)
            .doOnSubscribe { showProgress.onNext(true) }
            .subscribeOn(schedulersProvider.io())
            .observeOn(schedulersProvider.main())
            .pipeErrorTo(errorSubject)
            .subscribe(
                accessTokenResponse -> {
                    showProgress.onNext(false);
                    // do the next step here
                    // getUserProfile()
            })
```

The Coroutine way

```
fun login() {
    showProgress()

    GlobalScope.launch {
        getToken(authRepo, userName, pin, type)
    }
}

suspend fun getToken(authRepo: AuthenticationRepositoryImpl,
                     userName: String,
                     pin: String,
                     type: String) {

        withContext(Dispatchers.IO) {
            // Synchronously execute a Network Request
            val token = authRepo.requestAccessToken(userName, pin, type).blockingFirst()
            if (token != null) {
                keepTheToken(token, pin)
                typeYourPinViewModel.getCurrentUser()

                withContext(Dispatchers.Main) {
                    hideFragmentProgress()
                }
            }
        }
    }
```

Coroutine and Goroutine?

```kotlin
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()

    launch {
        for (i in 1..5) {
            channel.send(i)
        }
        channel.close()
    }

    for (i in channel) {
        println("received: $i")
    }
}
//console output
received: 1
received: 2
received: 3
received: 4
received: 5
```

```go
package main
import (
 "fmt"
)
func myFunc(done chan string) {
   for i := 0; i < 5; i++ {
      fmt.Println(i )
   }
   fmt.Println("finished loop in myFunc")
   done <- "goroutine finished" // send the message into the channel
}
func main() {
   done := make(chan string)   // make the "done" channel
   go myFunc(done)

   msg := <- done              // receive from the channel
   fmt.Println(msg)       // print out the value when receiving
}
```

Achievement and progress

- Seize more insights about threading issues on The 1 application.
- Established improvement solutions.
- The verdict?
- Next step: is pending on when the stable support libs come out from Google to judge on.

Next Steps

Pending on when the stable support libs come out from Google to judge on

The key point here, is understanding the system design on the OS we work on. Tools are just semantic differences, mostly.

Thanks!

Contact Nimble

nimblehq.co

hello@nimblehq.co

Bangkok

399 Interchange 21 Sukhumvit Road, Unit #2402-03, Klong Toei, Wattana, Bangkok 10110, Thailand

Singapore

28C Stanley St, Singapore 068737

Hong Kong

20th Floor, Central Tower

28 Queen's Road, Central, Hong Kong

nimble