Who We Are?

Varun Sharma
(SSE Android Team Tokopedia)

Lalit Pratap Singh
(SSE Android Team Tokopedia)

tokopedia

(Technology Company with a mission to democratize commerce through technology)

- 4 million DAU on Android
- 5.4 million sellers
- 90 million users/month
- 70% revenue from Android

Functional Programming

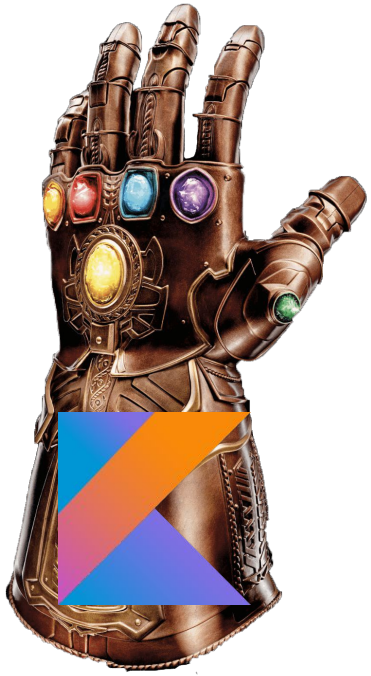
With Kotlin

Treating computation as the evaluation of mathematical functions while avoiding state and mutable data.

Language Support Needed

- ➔ Higher-Order Functions
- ➔ Lambdas/Anonymous Functions
- ➔ Immutable Data

Kotlin offers All of them



Higher Order Function

- ➔ Accepts function as parameter
- ➔ Function as return type
- ➔ Function can be stored in variables and data structures

Function as parameter

```
fun foo(m: String, bar: (m: String) -> Unit) {
bar(m)
}
fun buz(m: String) {
println("another message: $m")
}
fun something() {
foo("hi", ::buz)
}
```

```
OUTPUT
another message: hi
```

Function as return type

```
fun add(a:Int, b:Int): Int{
    return a+b
}
fun getAddFunc(): (Int, Int) -> Int{
    var funcVar = ::add
    return funcVar
}
fun getExecuteAdd(a:Int, b:Int){
    var addFunc = getAddFunc()
    addFunc(a, b)
}
```

- ➔ :: operator used to get Function Reference
- ➔ Function reference is an example of reflection. It returns reference to the function which also implements an interface that represent function type.

Lambdas/Anonymous Functions

- Lambda expression

```
val square = { x: Int -> x * x }
```

- Anonymous Functions

```
val square: (Int)->Int = fun(x) = x * x
```

Immutable Data

➔ All function parameters are final in Kotlin.

Pure Function

A Function is pure function if :

- ➔ Produces the result based only on input.
- ➔ Result independent of external state.
- ➔ No observable side effects.
  - ◆ Does not modify a parameter passed by reference or global variable/object.

Base Method

```
fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
    var results: MutableList<Movie> = mutableListOf()

    do {
        val movie = collection.removeAt(0)
        if (movie.title.contains(query)){
            results.add(movie)
        }
    }
    while (collection.size > 0)

    return results
}
```

From Imperative to Procedural

```kotlin
class FindByTitleKotlin {

    companion object {
        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: MutableList<Movie> = mutableListOf()

            do {
                val movie = collection.removeAt( index: 0)
                if (movie.title.contains(query)){
                    results.add(movie)
                }
            }
            while (collection.size > 0)

            return results
        }

    }
}
```

```kotlin
class FindByTitleKotlin {

    companion object {

        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: MutableList<Movie> = mutableListOf()

            do {
                val movie = collection.removeAt( index: 0)
                addIfMatches(query, movie, results)
            }
            while (collection.size > 0)

            return results
        }

        fun addIfMatches(query: String, movie: Movie, results: MutableList<Movie>){
            if (matches(query, movie)){
                results.add(movie)
            }
        }

        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        fun title(movie: Movie): String = movie.title

        fun String.isInfixOf(query: String) = contains(query)

    }
```

From Procedural to Functional : appearance of a Side Effect

```kotlin
class FindByTitleKotlin {

    companion object {

        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: MutableList<Movie> = mutableListOf()

            do {
                val movie = collection.removeAt(0)
                addIfMatches(query, movie, results)
            }
            while (collection.size > 0)

            return results
        }

        fun addIfMatches(query: String, movie: Movie, results: MutableList<Movie>){
            if (matches(query, movie)){
                results.add(movie)
            }
        }

        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        fun title(movie: Movie): String = movie.title

        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

```kotlin
class FindByTitleKotlin {

    companion object {

        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: MutableList<Movie> = mutableListOf()

            do {
                val movie = collection.removeAt(0)
                addIfMatches(query, movie, results)
            }
            while (collection.size > 0)

            return results
        }

        // TODO Side effect (results, because modified outside its local environment)
        fun addIfMatches(query: String, movie: Movie, results: MutableList<Movie>){
            if (matches(query, movie)){
                results.add(movie)
            }
        }

        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        fun title(movie: Movie): String = movie.title

        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

Moving up the Side Effect

```kotlin
class FindByTitleKotlin {

    companion object {

        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: MutableList<Movie> = mutableListOf()

            do {
                val movie = collection.removeAt(0)
                addIfMatches(query, movie, results)
            }
            while (collection.size > 0)

            return results
        }

        // TODO Side effect (results, because modified outside its local environment)
        fun addIfMatches(query: String, movie: Movie, results: MutableList<Movie>){
            if (matches(query, movie)){
                results.add(movie)
            }
        }

        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        fun title(movie: Movie): String = movie.title

        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

```kotlin
ass FindByTitleKotlin {

  companion object {

      // findByTitle :: (String, [Movie]) -> [Movie]
      fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
          var results: MutableList<Movie> = mutableListOf()

          // Use of functions as variables (predicate and add) : functions as first class citizen

          // matches :: (String, Film) -> Boolean
          val predicate = ::matches

          // TODO side effect moved up (still on results)
          // add :: (Film) -> Boolean
          val add = fun (movie: Movie) = results.add(movie)

          for (movie: Movie in collection){
              val fn = addIf(predicate, query, movie, add)
              fn(movie)
          }

          return results
      }

      // addIfMatches :: ((String, Movie) -> Boolean, String, Movie, [Movie] -> (Boolean)) -> (Movie) -> (Boolean)
      fun addIf(predicate: (String, Movie) -> Boolean, query: String, movie: Movie, add: (Movie) -> (Boolean)): (Movie) -> (Boolean){
          if (predicate(query, movie)){
              return add
          }
          return fun (movie: Movie) = false
      }

      // matches :: (String, Film) -> Boolean
      fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

      // title :: (Film) -> String
      fun title(movie: Movie): String = movie.title

      // isInfixOf :: (String, String) -> Boolean
      fun String.isInfixOf(query: String) = contains(query)

  }
```

Removing the Side Effect

```kotlin
class FindByTitleKotlin {

    companion object {

        // findByTitle :: (String, [Movie]) -> [Movie]
        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: MutableList<Movie> = mutableListOf()

            // Use of functions as variables (predicate and add) : functions as first class citizen

            // matches :: (String, Film) -> Boolean
            val predicate = ::matches

            // TODO side effect moved up (still on results)
            // add :: (Film) -> Boolean
            val add = fun (movie: Movie) = results.add(movie)

            for (movie: Movie in collection){
                val fn = addIf(predicate, query, movie, add)
                fn(movie)
            }

            return results
        }

        /*addIfMatches :: ((String, Movie) ->
                        Boolean, String, Movie, [Movie] -> (Boolean)) -> (Movie) -> (Boolean)*/
        fun addIf(predicate: (String, Movie) -> Boolean,
                  query: String, movie: Movie,
                  add: (Movie) -> (Boolean)): (Movie) -> (Boolean){
            if (predicate(query, movie)){
                return add
            }
            return fun (movie: Movie) = false
        }

        // matches :: (String, Film) -> Boolean
        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        // title :: (Film) -> String
        fun title(movie: Movie): String = movie.title

        // isInfixOf :: (String, String) -> Boolean
        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

```kotlin
class FindByTitleKotlin {

    companion object {

        // findByTitle :: (String, [Movie]) -> [Movie]
        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: List<Movie> = listOf()

            // matches :: (String, Film) -> Boolean
            val predicate = ::matches

            // add :: (Film) -> Boolean
            val add = fun (movie: Movie, movies: List<Movie>) = movies.plus(movie)

            for (movie: Movie in collection){
                val fn = addIf(predicate, query, movie, add)
                results = fn(movie, results)
            }

            return results
        }

        /*addIfMatches :: ((String, Movie) ->
                        Boolean, String, Movie, [Movie] -> (Boolean)) -> (Movie) -> (Boolean)*/
        fun addIf(predicate: (String, Movie) -> Boolean,
                  query: String, movie: Movie,
                  add: (Movie, List<Movie>) -> (List<Movie>)): (Movie, List<Movie>) -> (List<Movie>){
            if (predicate(query, movie)){
                return add
            }
            return fun (movie: Movie, movies: List<Movie>) = listOf<Movie>()
        }

        // matches :: (String, Film) -> Boolean
        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        // title :: (Film) -> String
        fun title(movie: Movie): String = movie.title

        // isInfixOf :: (String, String) -> Boolean
        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

Currying

```kotlin
class FindByTitleKotlin {

    companion object {

        // findByTitle :: (String, [Movie]) -> [Movie]
        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: List<Movie> = listOf()

            // matches :: (String, Film) -> Boolean
            val predicate = ::matches

            // add :: (Film) -> Boolean
            val add = fun (movie: Movie, movies: List<Movie>) = movies.plus(movie)

            for (movie: Movie in collection){
                val fn = addIf(predicate, query, movie, add)
                results = fn(movie, results)
            }

            return results
        }

        /*addIfMatches :: ((String, Movie) ->
                        Boolean, String, Movie, [Movie] -> (Boolean)) -> (Movie) -> (Boolean)*/
        fun addIf(predicate: (String, Movie) -> Boolean,
                query: String, movie: Movie,
                add: (Movie, List<Movie>) -> (List<Movie>)): (Movie, List<Movie>) -> (List<Movie>){
            if (predicate(query, movie)){
                return add
            }
            return fun (movie: Movie, movies: List<Movie>) = listOf<Movie>()
        }

        // matches :: (String, Film) -> Boolean
        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        // title :: (Film) -> String
        fun title(movie: Movie): String = movie.title

        // isInfixOf :: (String, String) -> Boolean
        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

```kotlin
class FindByTitleKotlin {

    companion object {

        // findByTitle :: (String, [Movie]) -> [Movie]
        fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{
            var results: List<Movie> = listOf()

            // matches :: (String, Film) -> Boolean
            val predicate = ::matches

            // add :: (Film) -> Boolean
            val add = fun (movie: Movie) = fun (movies: List<Movie>) = movies.plus(movie)

            for (movie: Movie in collection){
                val fn = addIf(predicate, query, movie, add)
                results = fn(movie)(results)
            }

            return results
        }

        /*addIfMatches :: ((String, Movie) ->
                        Boolean, String, Movie, [Movie] -> (Boolean)) -> (Movie) -> (Boolean)*/
        fun addIf(predicate: (String, Movie) -> Boolean,
                query: String, movie: Movie,
                add: (Movie) -> (List<Movie>) -> List<Movie>): (Movie) -> (List<Movie>) -> List<Movie>{
            if (predicate(query, movie)){
                return add
            }
            return fun (movie: Movie) = fun (movies: List<Movie>) = listOf<Movie>()
        }

        // matches :: (String, Film) -> Boolean
        fun matches(query: String, movie: Movie): Boolean = title(movie).isInfixOf(query)

        // title :: (Film) -> String
        fun title(movie: Movie): String = movie.title

        // isInfixOf :: (String, String) -> Boolean
        fun String.isInfixOf(query: String) = contains(query)

    }
}
```

Coroutines

New approach to write Async Code in Sequential manner

Context

Consider the `showUserOrders` function

```
fun showUserOrders(username: String, password: String) {
    val user = login(username, password)
    val orders = fetchUserOrders(user.userId)
    showUserOrders(orders)
}

fun login(username: String, password: String) { }

fun fetchUserOrders(userId: Long) { }
```

If we use callbacks then function look like

```
fun showUserOrders(username: String, password: String) {
    login(username, password) {
        user -> fetchUserOrders(user.userId) {
            orders -> showUserOrders(orders)
        }
    }
}

fun login(username: String, password: String, callback: (User) -> Unit)

fun fetchUserOrders(userId: Long, callback: (List<Orders>) -> Unit)
```

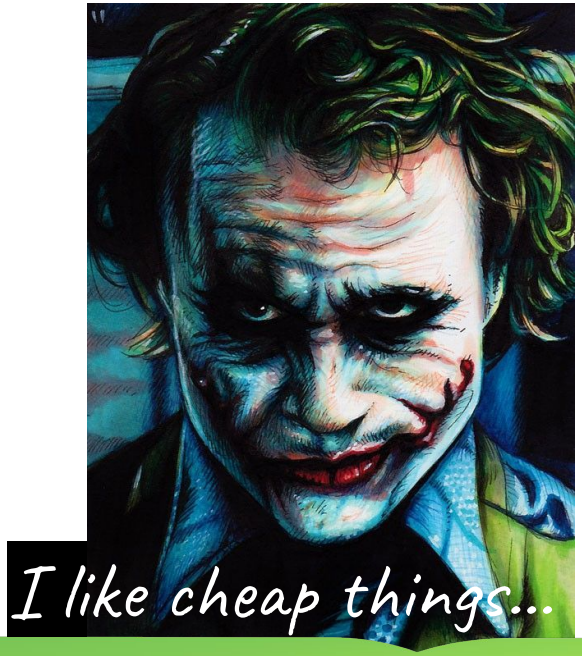
It is difficult to cancel background operations which consequently leads to memory leaks in some of the cases.

RxJava solves this problem but it is an overkill to use for a simple scenario of fetching data from the background

Introduction

A coroutine is a computation that can be *paused* or *suspended* and *resumed* at a later time.

- ➔ Coroutine are lightweight thread.
- ➔ Coroutines are cheap
- ➔ Async code will look like Synchronous code



Key Coroutine Concepts

- ➔ `suspend()` function
- ➔ Coroutine Context
- ➔ Coroutine Dispatcher
- ➔ Coroutine Builder

Suspend Function

- Suspend Function is like its backbone.
- A suspending function is simply a function that can be paused and resumed at a later time.
- Suspend fun can only be called from `suspend()` and coroutine.
- The syntax of a suspending function is similar to that of a regular function except for the addition of the `suspend` keyword

```
suspend fun backgroundTask(param: Int): Int {
    // long running operation
}
```

Coroutine Context

- It is an indexed set that maps from a `CoroutineContext.KEY` to a `CorountineContext.ELEMENT.`
- Some of coroutine context elements are:
  - **Job**: An object that represents a background operation and its lifecycle.
  - **CoroutineName**: User specified name for the coroutine.
  - **CoroutineExceptionHandler**: Exception handler used to handle uncaught exceptions in the current context.

Coroutine dispatcher

- ➔ Coroutine dispatchers are continuation interceptors and ***coroutine context element.***
- ➔ Dispatchers ensure that the execution and continuation of a coroutine gets dispatched on the correct thread.
  - ❖ `Dispatchers.Main` to dispatch execution onto the Android main UI thread (for the parent coroutine).
  - ❖ `Dispatchers.IO` to dispatch execution in the background thread (for the child coroutines).

Coroutine uses ***CommonPool*** for ***background Context*** which limit the number of threads running in parallel to 64 or the number of cores (whichever is larger).

```
// represent a pool of shared threads as coroutine dispatcher
val bgDispatcher: CoroutineDispatcher = Dispatchers.I0
```

Coroutine Builders

- Coroutine Builders provide a way to launch a coroutine from a regular function or non suspending scope.
- They are extension functions on CoroutineScope and they inherit the CoroutineContext of the scope they are invoked in.
- CoroutineScope basically controls the lifecycle of a coroutine and should be implemented on entities that have a well defined scope or lifecycle.
- The two most popular coroutine builders are
  - ➔ **launch**
  - ➔ **async**

launch

- We can launch a coroutine using *launch* builder in context with same Coroutine scope
- When we launch a Coroutine using *launch*, it returns a Job object.
- Job objects can be used to cancel or check the status of the coroutine.It is similar to a Subscription/Disposable in RxJava

Coroutine Scope

Coroutine builder

```
GlobalScope.launch { this: CoroutineScope
    val userId : String =getUserId()
    val user : User = getUserById(userId)
    val profileImage : Bitmap = downloadProfileImage(user.profileUrl)
}
```

async

- ➔ *async* is just like *launch*. It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines.
- ➔ Main difference between *async* and *launch* is while *launch* returns a Job, *async* returns a Deferred.
- ➔ Deferred - A light-weight non-blocking future that represents a promise to provide a result later

async Coroutine Builder

```
uiScope.launch { this: CoroutineScope
    val imageOne : Deferred<Bitmap> =
        ioScope.async { this: CoroutineScope
            downloadImage(imageOneUrl)
        }
    processImages(imageOne.await())
}
```

launch + withContext

```
val uiScope = CoroutineScope(Dispatchers.Main)
val bgDispatcher = CoroutineScope(Dispatchers.IO)
fun loadData() = uiScope.launch {
    view.showLoading() // ui thread

    val result1 = withContext(bgDispatcher) { // background thread
        // your blocking call
    }

    val result2 = withContext(bgDispatcher) { // background thread
        // your blocking call
    }

    val result = result1 + result2

    view.showData(result) // ui thread
}
```

Parallel Processing

```
fun performCalculation = uiScope.launch {
        val first = ioScope.async { firstNumber() }
        val second = ioScope.async { secondNumber() }
        val third = ioScope.async { thirdNumber() }

        val result = first.await() + second.await() + third.await()
    }
    println(time)
}
```

Thanks

Q&A?

Reach out to us:
varun.sharma@tokopedia.com
lalit.singh@tokopedia.com