DROIDCONKE. 2022

Building For The Masses With Kotlin

Generics & High Order Functions

BRIAN ODHIAMBO
@mambo_bryan

VICTOR OYANDO
@mawinda_vic

02

"Thrill-seeking adventurer"

Victor Oyando

Software Developer

Brian Odhiambo

Software Developer

"Art-seeking eccentric"

So why this talk?

- project based culture
- multiple apps, multiple lists
- constrained developer time

DEFINITIONS

What are Generics?

I've heard of them somewhere...

05

Generics

Idea to allow type to be a parameter in method classes or interface

```
data class Person(val name: String = "droidcon")

data class Luggage(val quantity: Int = 100)

// Example
class Matatu<T>()

// without type inference
val mat3 : Matatu<Person> = Matatu<Person>()

// with type Inference
val mat3moto = Matatu<Luggage>()
```

DEFINITIONS

What about high order functions?

what are those...

High Order Functions

Any function that can take a function as a parameter or return a function

```
// Example
class Matatu<T>(){
    val items = mutableListOf<T>()
    fun board(block : () -> List<T>){
        items.addAll(block.invoke())
    }
}

// 1. loading humans
val mat3 : Matatu<Person> = Matatu()
mat3.board{
    listOf(Person(), Person())
}

// 2. loading luggage
val mat3moto = Matatu<Luggage>()
mat3moto.board{
    listOf(Luggage(), Luggage())
}
```

USE-CASES

What are the common uses?

How do developers use them

09

Safe Api Call

The saviour of network request

```
data class NetworkResult<T>(
    val isSuccessful: Boolean,
    val message: String,
    val data: T? = null
)

fun <T> safeApiCall(block: () -> T): NetworkResult<T> {
    return try {
        val result = block.invoke()
        NetworkResult(isSuccessful = false, message = "success", data = result)
    } catch (e: Exception) {
        NetworkResult(
           isSuccessful = false,
           message = e.message ?: "error",
           data = null)
    }
}

val error = safeApiCall { "2a".toInt() }
println(error)
// NetworkResult(isSuccessful=false, message=For input string: "2a", data=null)

val success = safeApiCall { "20".toInt() }
println(success)
// NetworkResult(isSuccessful=false, message=success, data=20)
```

USE-CASES

So how do we use them?

Oh boy, What sorcery have we come up with

Lazy Adapter

One Adapter to rule them all

```
val adapter = LazyAdapter<Luggage, LayoutLuggageBinding>()

adapter.onCreate { LayoutLuggageBinding.inflate(it.inflater(), it, false) }
adapter.onBind { item -> this.apply {tvLuggageCount.text = item.count.toString()} }
adapter.onBind { item: Luggage, selected: Boolean -> ... }
adapter.onBind { item: Luggage, selected: Boolean, position: Int -> ... }
adapter.onItemClicked { }
adapter.onItemLongClicked { false }
adapter.onSwipedRight { }
adapter.onSwipedLeft { }
...

// in future populate the list
adapter.submitList(listOf())

// set adapter to the recyclerview
recyclerview.setAdapter(adapter)
```

Here

// Argh adapters

```
class PersonAdapter : ListAdapter<PersonViewHolder, Person>(...)
class LuggageAdapter : ListAdapter<PersonViewHolder, Person>(...)

// you can't do this
class MatatuAdapter<T> : ListAdapter<Matatu<T>, MatatuViewHolder>(...)

// so you have to do this
class MatatuWithPersonAdapter : ListAdapter<Matatu<Person>, ...>(...)
class MatatuWithLuggageAdapter : ListAdapter<Matatu<Luggage>, ...>(...)
```

Here

Argh adapters

```
class PersonAdapter : ListAdapter<PersonViewHolder, Person>(...)
class LuggageAdapter : ListAdapter<PersonViewHolder, Person>(...)

// you can't do this
class MatatuAdapter<T> : ListAdapter<Matatu<T>, MatatuViewHolder>(...)

// so you have to do this
class MatatuWithPersonAdapter : ListAdapter<Matatu<Person>, ...>(...)
class MatatuWithLuggageAdapter : ListAdapter<Matatu<Luggage>, ...>(...)
```

There :

```
val adapter = LazyAdapter<T, ItemTbinding>()
```

Lazy Compare

Extend me please

```
// create an abstract class
abstract class LazyCompare {
    open fun areItemsSame(newItem: Any?): Boolean = this == newItem
    open fun areContentsSame(newItem: Any?): Boolean = this == newItem
}

// without custom comparison
data class Person(val name: String = "koko") : LazyCompare()

// with custom comparison
data class Luggage(val quantity: Int = 100) : LazyCompare(){
    override fun areItemsSame(newItem: Any?): Boolean {
      return when (newItem) {
         !is Luggage -> false
        else -> this.quantity == newItem.quantity
      }
    }
    ...
}

// Now the Person & Luggage class have become Lazy comparable objects
```

Lazy Compare

Extend me please

```
// create an abstract class
abstract class LazyCompare {
    open fun areItemsSame(newItem: Any?): Boolean = this == newItem
    open fun areContentsSame(newItem: Any?): Boolean = this == newItem
}

// without custom comparison
data class Person(val name: String = "koko") : LazyCompare()

// with custom comparison
data class Luggage(val quantity: Int = 100) : LazyCompare(){
    override fun areItemsSame(newItem: Any?): Boolean {
      return when (newItem) {
         !is Luggage -> false
        else -> this.quantity == newItem.quantity
      }
    }
    ...
}

// Now the Person & Luggage class have become Lazy comparable objects
```

Lazy Compare

Extend me please

```
// create an abstract class
abstract class LazyCompare {
    open fun areItemsSame(newItem: Any?): Boolean = this == newItem
    open fun areContentsSame(newItem: Any?): Boolean = this == newItem
}

// without custom comparison
data class Person(val name: String = "koko") : LazyCompare()

// with custom comparison
data class Luggage(val quantity: Int = 100) : LazyCompare(){
    override fun areItemsSame(newItem: Any?): Boolean {
      return when (newItem) {
         !is Luggage -> false
        else -> this.quantity == newItem.quantity
      }
    }
    ...
}

// Now the Person & Luggage class have become Lazy comparable objects
```

Lazy Adapter

Under the hood

```
class LazyAdapter<T : LazyCompare, R : ViewBinding> :
ListAdapter<T, LazyAdapter<T, R>.LazyHolder>(object : DiffUtil.ItemCallback<T>() {
    override fun areItemsTheSame(oldItem: T, newItem: T): Boolean =
        oldItem.areItemsSame(newItem)
    override fun areContentsTheSame(oldItem: T, newItem: T): Boolean =
        oldItem.areContentsSame(newItem)
}) {

  ...

  inner class LazyViewHolder(context: Context, val binding: R?) :
    RecyclerView.ViewHolder(binding?.root ?: View(context)) { ... }

}
```

```
class LazyAdapter<T : LazyCompare, R : ViewBinding> ... {

   private var getViewBinding: ((parent: ViewGroup) -> R)? = null
   fun onCreate(create: (parent: ViewGroup) -> R) = apply {
      getViewBinding = create
   }

   private var executeBindings: (R.(item: T) -> Unit)? = null
   fun onBind(bind: R.(item: T) -> Unit) = apply {
      executeBindings = bind
   }

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): LazyViewHolder {
       val binding = getViewBinding?.invoke(parent)
       return LazyHolder(parent.context, binding)
   }

   override fun onBindViewHolder(holder: LazyViewHolder, position: Int) {
       val item = getItem(position) as T
       holder.bindHolder(item)
   }

   inner class LazyHolder (context: Context, val binding: R?) ... { ... }

}
```

Lazy Holder

How's the view...

```
...

private var executeBindings: (R.(item: T) -> Unit)? = null

fun onBind(bind: R.(item: T) -> Unit) = apply {
    executeBindings = bind
}

...

inner class LazyViewHolder(context: Context, val binding: R?) :
    RecyclerView.ViewHolder(binding?.root ?: View(context)) {

    fun bindHolder(item: T) {
        executeBindings?.let { block -> binding?.block(item) }
    }
}
```

Sample Use

How does it look?

```
// create a layout for your view
item_person.xml or item_luggage.xml

// create adapter with Lazy comparable object and ViewBinding class
val adapter = LazyAdapter<Person, ItemPersonBinding>()
or
val adapter = LazyAdapter<Luggage, ItemLuggageBinding>()

adapter
.onCreate { parent: ViewGroup ->
    ItemPersonBinding.inflate(it.inflater, it, false)
}
.onBind { item: Luggage ->
    this.apply { }
}
...
```

Other Functions

I need more power

```
class LazyAdapter<T : LazyCompare, R : ViewBinding> ... {

    ...
    private var executeBindingWithPosition: (R.(item: T, position: Int) -> Unit)? = null

    @JvmName("onBindWithPosition")
    fun onBind(block: R.(item: T, position: Int) -> Unit) = apply {
        mBindPosition = block
    }

    private var onClick: ((item: T) -> Unit)? = null
    fun onItemClicked(block: ((item: T) -> Unit)? = null) = apply {
        mClicked = block
    }

    ...

}
```

End Result

One generic adapter

```
val adapter = LazyAdapter<Luggage, LayoutLuggageBinding>()

adapter.onCreate { LayoutLuggageBinding.inflate(it.inflater(), it, false) }
.onBind { item -> this.apply {tvLuggageCount.text = item.count.toString()} }
.onBind { item: Luggage, selected: Boolean -> ... }
.onBind { item: Luggage, selected: Boolean, position: Int -> ... }
.onItemClicked { }
.onItemLongClicked { false }
.onSwipedRight { }
.onSwipedLeft { }
...

// in future populate the list
adapter.submitList(listOf())

// set adapter to the recyclerview
recyclerview.setAdapter(adapter)
```

Lessons Learnt?

- Removes redundancies
- Focus more on the UI
- Can be customised to fit any requirement

Advantages

- simple, maintainable, robust, and scalable
- removes redundancies
- saves time
- focus is on the UI

Disadvantages

- runtime overhead
  (can be fixed with inline modifier)

- affects code readability
  (then write descriptive functions)

ASK US ANYTHING

Q & A

If you dare...

@mawindavic

@MamboBryan

@mawinda_vic

@mambo_bryan