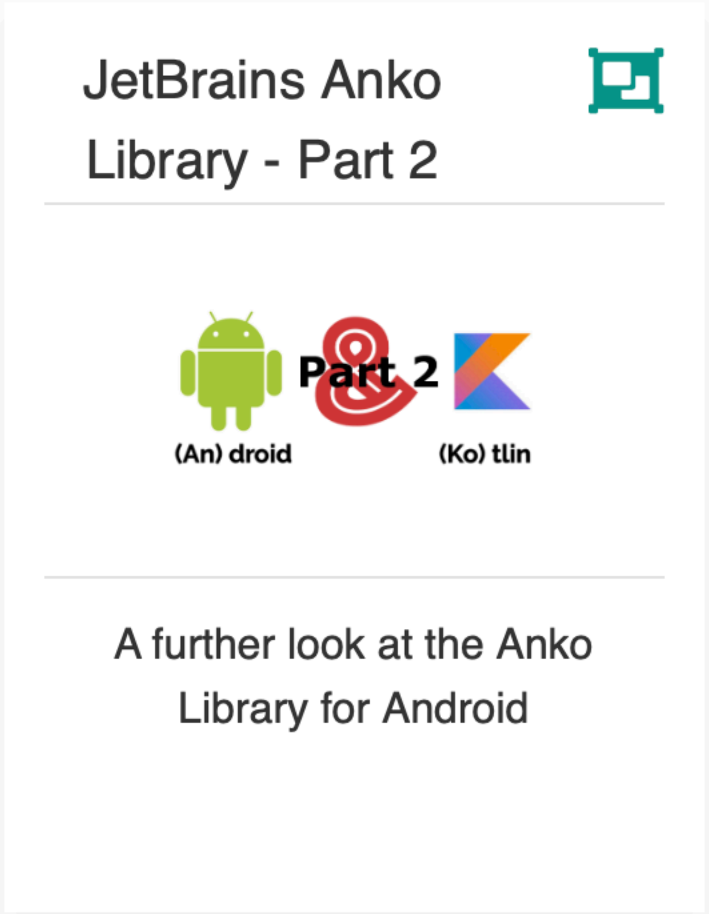
Mobile Application Development

Produced by David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Introducing JetBrains Anko Library



JetBrains Anko Library - Part 2

A further look at the Anko Library for Android

Agenda

- ❑ Background
- ❑ Extension Functions
- ❑ The Anko Library Components
    - Anko Commons
    - Anko Layouts
    - Anko SQLite
    - Anko Coroutines
- ❑ Anko in our Case Study (Donation)

Agenda

- ❑ Background
- ❑ Extension Functions
- ❑ The Anko Library Components
  - ▪ Anko Commons
  - ▪ Anko Layouts
  - ▪ Anko SQLite
  - ▪ Anko Coroutines
- ❑ Anko in our Case Study (Donation)

Background Recap

- ❑ Anko is a library for Android developers that want to achieve more while writing less.
- ❑ It simplifies common tasks that are tedious and generate a lot of boilerplate, making your code a lot easier to read and keeps it concise and clean
- ❑ Created and maintained by JetBrains

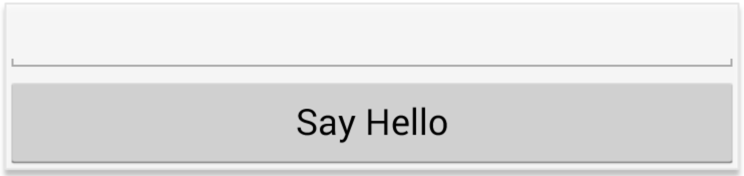
Anko *Layouts*

create type-safe Android layouts

Anko Layouts – What is it?

- Anko Layouts is a DSL for writing dynamic Android layouts. Here is a simple UI written with Anko DSL:

```
verticalLayout {
    val name = editText()
    button("Say Hello") {
        onClick { toast("Hello, ${name.text}!") }
    }
}
```



Anko Layouts – What is it?

- The code previous creates a **`button`** inside a **`LinearLayout`** and attaches an **`OnClickListener`** to that button.

- Moreover, **`onClick`** accepts a *suspend lambda*, so you can write your asynchronous code right inside the listener.

- There is also a plugin for Android Studio that supports previewing Anko DSL layouts.

Anko Layouts – Why? (and why a DSL?)

- By default, UI in Android is written using XML. That is inconvenient in the following ways:
  - It is not type-safe;
  - It is not null-safe;
  - It forces you to write almost the same code for every layout you make;
  - XML is parsed on the device wasting CPU time and battery;
  - Most of all, it allows no code reuse.
- While you can create UI programmatically, it's hardly ever done because it’s messy and hard to maintain. Here's a plain Kotlin version (Java’s is even longer):

Anko Layouts – Why? (and why a DSL?)

```
val act = this
val layout = LinearLayout(act)
layout.orientation = LinearLayout.VERTICAL
val name = EditText(act)
val button = Button(act)
button.text = "Say Hello"
button.setOnClickListener {
    Toast.makeText(act, "Hello, ${name.text}!", Toast.LENGTH_SHORT).show()
}
layout.addView(name)
layout.addView(button)
```

Anko Layouts – Why? (and why a DSL?)

- A DSL makes the same logic easy to read, easy to write and there is no runtime overhead. Here it is again:

```
verticalLayout {
    val name = editText()
    button("Say Hello") {
        onClick { toast("Hello, ${name.text}!") }
    }
}
```

- As previously mentioned, **`onClick()`** supports coroutines (accepts suspending lambda) so you can write your asynchronous code without an explicit **`async(UI)`** call.

Anko Layouts – LayoutParams

- The positioning of widgets inside parent containers can be tweaked using **`LayoutParams`**. In XML it looks like this:

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="5dip"
    android:layout_marginTop="10dip"
    android:src="@drawable/something" />
```

- In Anko, you specify **`LayoutParams`** right after a **`View`** description using **`lparams()`**

Anko Layouts – LayoutParams

- ❑ If you specify **`lparams()`**, but omit width and/or height, their default values are both **`wrapContent`**.

```
linearLayout {
    button("Login") {
        textSize = 26f
    }.lparams(width = wrapContent) {
        horizontalMargin = dip(5)
        topMargin = dip(10)
    }
}
```

Anko *SQLite*

helpers for working with Android SQLite;

Anko SQLite

- If you’ve ever tried to parse SQLite query results using Android cursors, you would have had to write lots of boilerplate code just to parse query result rows, and enclose it in countless try..finally blocks to properly close all opened resources.
- You would use **`SQLiteOpenHelper`**, and call **`getReadableDatabase()`** or **`getWritableDatabase()`**
- Anko provides lots of extension functions to simplify working with SQLite databases.

Anko SQLite

- Anko provides a special class **`ManagedSQLiteOpenHelper`** that seamlessly replaces the default one

```
fun getUsers(db: ManagedSQLiteOpenHelper): List<User> = db.use {
    db.select("Users")
            .whereSimple("family_name = ?", "John")
            .doExec()
            .parseList(UserParser)
}
```

- Instead of enclosing your code in lots of try blocks, you can just write **`databasereference.use { }`**

Anko SQLite – Creating and dropping tables

- With Anko you can easily create new tables and drop existing ones. The syntax is straightforward.

```
database.use {
    createTable("Customer", true,
        "id" to INTEGER + PRIMARY_KEY + UNIQUE,
        "name" to TEXT,
        "photo" to BLOB)
}
```

- and

```
dropTable("User", true)
```

Anko SQLite – Inserting Data

- Usually, you need a **`ContentValues`** instance to insert a row into the table, Anko has a simple **`insert`** function

```
db.insert("User",
    "id" to 42,
    "name" to "John",
    "email" to "user@domain.org"
)
```

- or

```
database.use {
    insert("User",
        "id" to 42,
        "name" to "John",
        "email" to "user@domain.org"
    )
}
```

Anko SQLite – Querying Data

- Anko provides a convenient query builder. It may be created with **`db.select(tableName, vararg columns)`**

```
db.select("User", "name")
    .whereArgs("(_id > {userId}) and (name = {userName})",
        "userName" to "John",
        "userId" to 42)
```

- and

```
db.select("User", "email").exec {
        // Doing some stuff with emails
}
```

Anko SQLite – Updating/Deleting Data

```kotlin
update("User", "name" to "Alice")
        .where("_id = {userId}", "userId" to 42)
        .exec()
```

❑ and

```kotlin
val numRowsDeleted = delete("User", "_id = {userID}", "userID" to 37)
```

Anko *Coroutines*

utilities for using Kotlin coroutins.

Anko Coroutines

- Anko Coroutines is based on the kotlinx.coroutines library and provides:
  - **`bg()`** function that executes your code in a common pool.
  - **`asReference()`** function which creates a weak reference wrapper. By default, a coroutine holds references to captured objects until it is finished or canceled. If your asynchronous framework does not support cancellation, the values you use inside the asynchronous block can be leaked. asReference() protects you from this.
  - **`doAsync()`** let us run codes asynchronously or in the background.

Anko Coroutines

```kotlin
fun getData(): Data { ... }
fun showData(data: Data) { ... }

doAsync(UI) {
    val data: Deferred<Data> = bg {
                // Runs in background
                getData()
    }

    // This code is executed on the UI thread
    showData(data.await())
}
```

Summary

- Anko is a Kotlin library which makes Android application development faster and easier
- It consists of several parts:
  - Anko Commons: a lightweight library full of helpers for intents, dialogs, logging and so on;
  - Anko Layouts: a fast and type-safe way to write dynamic Android layouts;
  - Anko SQLite: a query DSL and parser collection for Android SQLite;
  - Anko Coroutines: utilities based on the *kotlinx.coroutines* library.

References

Sources: https://github.com/Kotlin/anko
https://adorahack.com/introduction-to-anko

Thanks.