Mobile Application Development

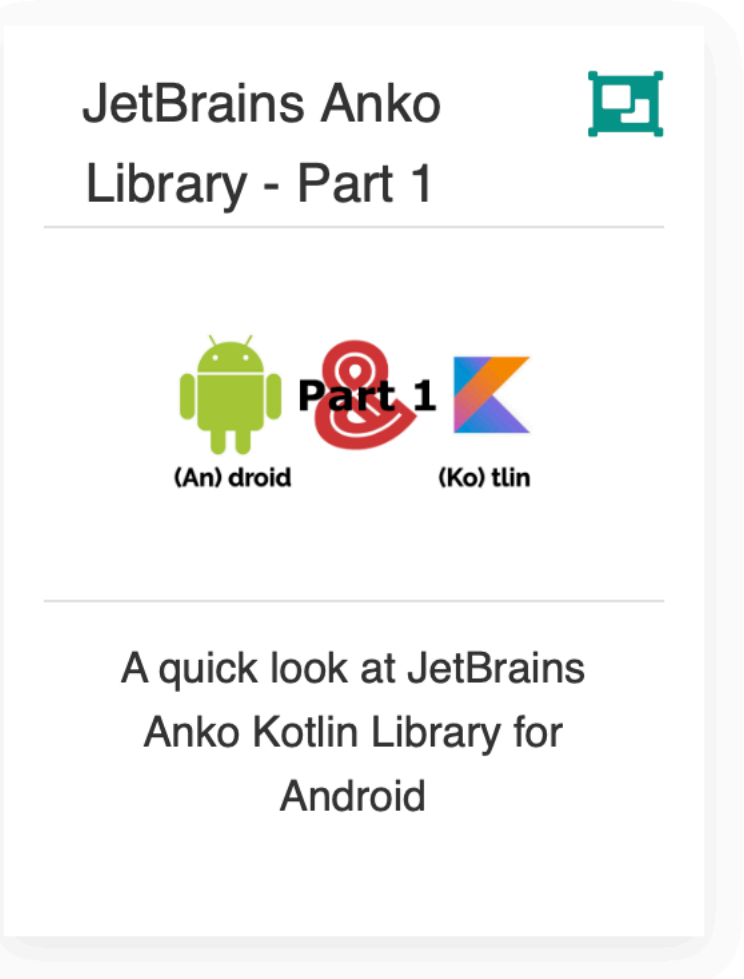
Produced by

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



Agenda

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

Agenda

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

Background

- **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
- The folks at JetBrains, (creators of Kotlin & IntelliJ) have created and continue to maintain **Anko**

Background (https://github.com/Kotlin/anko)

Anko

Anko is a Kotlin library which makes Android application development faster and easier. It makes your code clean and easy to read, and lets you forget about rough edges of the Android SDK for Java.

Anko 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.

Extension Functions

- To understand how **Anko** works, you need to understand Kotlin **Extension Functions**. They allow you to add a function to an existing class without modifying the class.
- For example, say you had a **`Dog`** class:

Dog.kt

```
class Dog(val name: String, val breed: String)
```

Extension Functions

- In another Kotlin file you could add a function to **`Dog`** without modifying the original file:

```
package com.raywenderlich.doggy

fun Dog.bark(): Unit{
  println("woof woof")
}
```

- To create the extension function, after **`fun`** type the class, then a dot, then the name of the extension function.

Extension Functions

- You could test your extension function in another file as follows:

```
//Importing bark extension function :]
import com.raywenderlich.doggy.bark

fun main(args: Array<String>) {
  var myPuppy = Dog("Max", "Pug")
  myPuppy.bark()
}
```

- To use the extension function, you only import bark, and then every Dog object will be able to use the bark() function.

Anko *Commons*

helpers for intents, dialogs, logging;

Anko Commons

- **Anko Commons** is a "toolbox" for Kotlin Android developers
- The library contains a lot of helpers for the Android SDK, including, but not limited to:
  - Intents
  - Dialogs and toasts
  - Logging
  - Resources and dimensions

Anko Commons - Intents

In general, you have to write a couple of lines to start a new `Activity`. And it requires you to write an additional line for each value you pass as an extra. For example, this is a code for starting an `Activity` with extra `("id", 5)` and a special flag:

```
val intent = Intent(this, SomeOtherActivity::class.java)
intent.putExtra("id", 5)
intent.setFlag(Intent.FLAG_ACTIVITY_SINGLE_TOP)
startActivity(intent)
```

Anko Commons - Intents

Four lines is too much for this. Anko offers you an easier way:

```
startActivity(intentFor<SomeOtherActivity>("id" to 5).singleTop())
```

If you don't need to pass any flags, the solution is even easier:

```
startActivity<SomeOtherActivity>("id" to 5)
```

If you want to put more than one parameter, just split it with comma.

```
startActivity<SomeOtherActivity>(
    "id" to 5,
    "city" to "Denpasar"
)
```

Anko Commons – Useful Intent Callers

Anko has call wrappers for some widely used `Intents`:

Goal	Solution
Make a call	`makeCall(number)` without **tel:**
Send a text	`sendSMS(number, [text])` without **sms:**
Browse the web	`browse(url)`
Share some text	`share(text, [subject])`
Send an email	`email(email, [subject], [text])`

Arguments in square brackets (`[]`) are optional. Methods return true if the intent was sent.

Anko Commons – Dialogs & Toasts

Toasts

Simply shows a Toast message.

```
toast("Hi there!")
toast(R.string.message)
longToast("Wow, such duration")
```

SnackBars

Simply shows a SnackBar message.

```
view.snackbar("Hi there!")
view.snackbar(R.string.message)
view.longSnackbar("Wow, such duration")
view.snackbar("Action, reaction", "Click me!") { doStuff() }
```

Anko Commons – Dialogs & Toasts

Alerts

A small DSL for showing alert dialogs.

```
alert("Hi, I'm Roy", "Have you tried turning it off and on again?") {
    yesButton { toast("Oh…") }
    noButton {}
}.show()
```

The code above will show the default Android alert dialog. If you want to switch to the appcompat implementation, use the `Appcompat` dialog factory:

```
alert(Appcompat, "Some text message").show()
```

Anko Commons – Dialogs & Toasts

Selectors

`selector()` shows an alert dialog with a list of text items:

```
val countries = listOf("Russia", "USA", "Japan", "Australia")
selector("Where are you from?", countries, { dialogInterface, i ->
    toast("So you're living in ${countries[i]}, right?")
})
```

Progress dialogs

`progressDialog()` creates and shows a progress dialog.

```
val dialog = progressDialog(message = "Please wait a bit…", title = "Fetching data")
```

Anko Commons – Logging with **AnkoLogger**

Android SDK provides `android.util.Log` class with some logging methods. Usage is pretty straightforward though the methods require you to pass a `tag` argument. You can eliminate this with using `AnkoLogger` trait-like interface:

```
class SomeActivity : Activity(), AnkoLogger {
    private fun someMethod() {
        info("London is the capital of Great Britain")
        debug(5) // .toString() method will be executed
        warn(null) // "null" will be printed
    }
}
```

android.util.Log	AnkoLogger
`v()`	`verbose()`
`d()`	`debug()`
`i()`	`info()`
`w()`	`warn()`
`e()`	`error()`
`wtf()`	`wtf()`

Each method has two versions: plain and lazy (inlined):

```
info("String " + "concatenation")
info { "String " + "concatenation" }
```

Anko Commons – Resources & Dimensions

Colors

Two simple extension functions to make the code more readable.

Function	Result
`0xff0000.opaque`	non-transparent red
`0x99.gray.opaque`	non-transparent #999999 gray

Dimensions

You can specify dimension values in **dip** (density-independent pixels) or in **sp** (scale-independent pixels): `dip(dipValue)` or `sp(spValue)`. Note that the `textSize` property already accepts **sp** (`textSize = 16f`). Use `px2dip` and `px2sp` to convert backwards.

References

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


Thanks.