Working with Features

Introduction

We have the assignment of demonstrating and documenting a kotlin specific feature and an android specific feature. We have chosen audio as our android feature and extension functions as our kotlin feature. We will demonstrate a simple app using the MediaPlayer library that can be used to control playback of audio/video files and streams.
Also we will demonstrate a simple program using extension functions showcasing their purpose.

Android - Audio

Layout

To this project we started by creating a new empty project, using Android Studio for this purpose.
In the *activity_main,xml* is used to define our layout for the *audio* app, the xml file can include GUI components such as, which in our case, button component.
The layout only defines the appearance of the app, not how it will function, the functionality of the application we will dive into after.
Android apps are a bunch of files in particular directories, when the app is build all these files gets bundled together, which results in an app that can run on the device.
In *activity_main,xml* we have made use of the layout *LinearLayout* which is a view group that aligns all the childrens in a single direction, either vertically or horizontally.
All the children, as a result of using *LinearLayout*, are stacked one after the other, so for example a vertically list will only have one child per row, no matter how wide they may be, and the horizontal list will be one row high.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:orientation="vertical"
        tools:context=".MainAct">

    <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="play"
            android:text="Play" />

    <Button
```

Above is an example of our *xml* file, where we make use of the *LinearLayout* and also use the *button* component.

The *button* component is set up a way, when it is clicked does execute the particular function assigned to it.

```xml
<Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="play"
        android:text="Play" />

<Button
```

MediaPlayer

In *MainActivity.kt* we build our application to do our desired work, and this class as per default extends `AppCompatActivity()`.

We have created a variable `player` of the type `MediaPlayer` which tells our app that this particular variable is a `MediaPlayer` and this is a framework supported by Android Studio to support functionalities such as controlling playback of audio or video files and streams.

Having Android providing this `MediaPlayer` class, we can access the built in services that `MediaPlayer` supports.

In order to use `MediaPlayer` we first have to call its static method `MediaPlayer.create()` and this method returns an instance of the `MediaPlayer` class.

This method takes two parameters, and the second parameter that this method takes are the of the audio file that you want to get executed, and to do that one first have to create a *raw* folder and place the audio file in it.

Once the `MediaPlayer` object is created we can make use of methods that can play and stop the music.

`player?.start()` and `player?.pause()` are some of the methods we have made use of.

Kotlin - Extension Functions

Extension Functions

Kotlin supports extension functions and extension properties. Extension functions provide the ability to extend a class with new functionality without having to inherit from the class. This is achieved by defining a receiver type - the the class to provide additional functionality to. Together with a function or a property

```
fun recieverType.extensionFunctionName(param: String) {
    //logic
}
```

Syntax for extension function

Extensions do not actually modify classes they extend. By defining an extension, you do not insert new members into a class, but merely make new functions callable with the dot-notation on variables of the receiver type.

Behind the scenes, they get compiled down to regular functions that take the target instance as a parameter. This makes extension functions syntactic sugar, but definitely not just a gimmick. So what problem does it solve exactly?

It binds this, so we don't need to use a qualified this. But most important - it makes code readable and easier to understand.

Especially when it comes to big nested calls.

```
foo(x, y) || foo(x, y)
```

Or in more complex scenarios it shines even more. This example uses an extension function and an extension property.

```
x.foo(y).bar(z).baz || getBaz(bar(foo(x, y)))
```