Application development for Android

Interface, *layouts* and *views*

Project creation

- Create an Android project
  - Specify the template "Empty views Activity" as the initial screen
  - API: 24

Activity

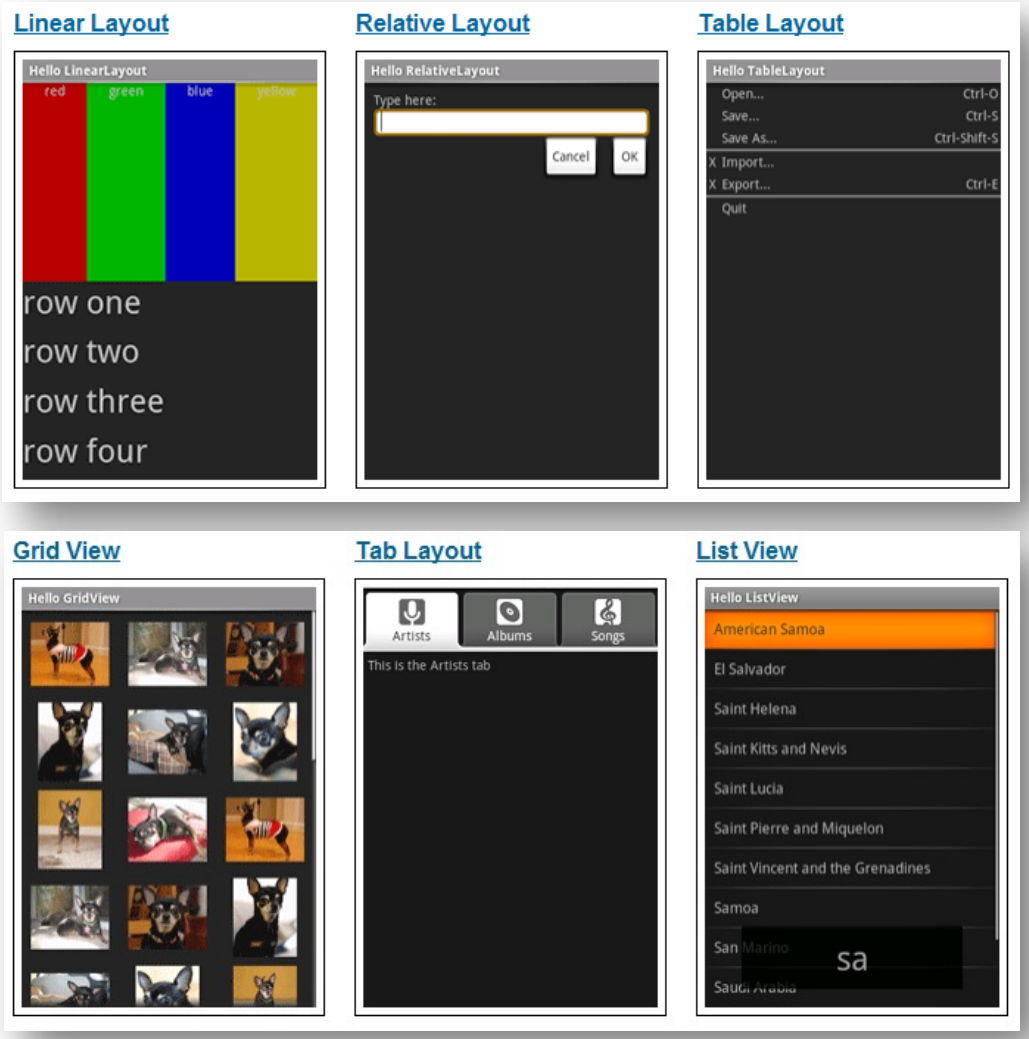
- When the project is created with the aforementioned settings, an activity is automatically created

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Interface

- The application interface is made up of a hierarchy of `View` objects: interaction objects and layout objects
- The *layout* objects help to structure and arrange various elements on the screen
- There are various *layout* objects (`ViewGroup` object extensions) which allow different ways of organizing elements
    - `FrameLayout`
    - `LinearLayout`
    - `RelativeLayout`
    - `TableLayout`
    - `GridLayout`
    - ~~`AbsoluteLayout`~~
    - …
    - `ConstraintLayout` *(Android Jetpack)*

Layout types



Interface

- Objects can be created in two ways:
  - Programmatically
    - Creating `View` objects or their derivatives
    - Forming a hierarchy of *views* and assigning it to the activity
      - `setContentView(view: View)`
  - Built from XML files
    - The application layout is defined through a XML file
    - When associated with the activity, information is automatically read from the file, and objects are created and configured one by one through an 'inflate' operation
      - `setContentView(layoutResID: Int)`

Layout XML files

- *The* XML files include the description of all objects that make up the *layout*
  - Saved in the *resources*: `res/layout`
  - Edited in XML or using the available editor
  - The *tags* XML allow to define the objects to be created, as well as the attributes relating to their initial configuration

- To each object is assigned an ID, which enables its identification
  - For each of these IDs, Android Studio generates and manages a constant in the '`R`' class
  - These constants allow access to the elements that they represent

Layout XML files

- Example of a file

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

    <TextView
        android:id="@+id/tvMsg"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</LinearLayout>
```

Layout XML files

- To obtain references to objects created during the inflate operation and to use them within *Kotlin* classes, the `findViewById` method must be used by passing the assigned ID (e. g., `R.id.<ID>`) as a parameter

- For example:

```
val tvMsg : TextView = findViewById(R.id.tvMsg)

tvMsg.text = "DEIS-AMOV"
```

View Binding

- With *Kotlin* + *Android Jetpack* the task of obtaining references to *views* is made easier through the *View Binding* functionality
  - Include in the `build.gradle.kts (Module :app)` file:

```
android {
    ...
    buildFeatures {
        viewBinding = true
    }
}
```

View Binding

- Inside the Kotlin class where it is intended to be used, do the following:

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        binding.tvMsg.text = "DEIS-AMOV"
    }
}
```

Alignment Attributes

- The elements included in the interface can be configured using different attributes that will be reflected in their alignment
  - In the context of *layout*
    - Mandatory
      - `android:layout_width`
      - `android:layout_height`
        - These attributes can take on the values: `wrap_content`, `match_parent` or a specific value for height and/or width in the format *<value><unit>*
          - The accepted units are as follows: *px (pixels), dp (density-independent pixels), sp (scaled pixels based on preferred font size), in (inches), mm (millimeters)*
    - Other: `layout_weight`, `layout_margin`, …
      - When defining weights for graphic elements, the dimension to be considered when dividing the weights (width or height) must be set to `0dp`
  - Internal to the element
    - Ex: `gravity`, `padding`, …

Application development for Android

Event processing

Event processing

- Assuming the definition of a button through the following layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"  android:layout_height="match_parent"
    android:orientation="vertical" >
    <Button
        android:id="@+id/btnOk" android:text="OK"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"  />
</LinearLayout>
```

Event processing

- The events generated on different elements can be processed using *listeners*
- Example for a button…
  - Get reference for button

    ```
    val btn : Button = findViewById(R.id.btnOK)
    ```

  - Use the method

    ```
    btn.setOnClickListener(ObjectOnClickListener)
    ```

  - The *`ObjectOnClickListener`* can be implemented using from:
    - Interface implementation in the class itself (*this*)
    - Implementation of the interface within the scope of a class created for this purpose
    - Instantiation of an object through an anonymous class
    - Inline creation and implementation of an object from an anonymous class – *Lambda function*

Button processing

- The buttons also allow click processing using the `onClick` attribute in the XML component description
  - Property not present in early versions of *Android*
  - Define the name of the method to be executed in the attribute
  - The method must follow the following template

```
fun methodName(v : View)
{
    ...
}
```

Button style

- The style of the buttons can be changed
  - For example, defining suggestive images for each of the buttons adapted to each situation
    - When pressed, when it has focus, when it is disabled
- Images can be included in the buttons by changing the button `drawable` attribute
  - `android:drawable[Left|Top|Bottom|Right|…]`
  - There is also a button specialization, called `ImageButton`, more adapted to configuring buttons with images only
- See also…
  - https://developer.android.com/develop/ui/views/components/button

Exercise 1

- Implement a basic calculator that allows the operations +, -, *, /, …

Exercise (consolidation)

- Change the developed calculator so that it displays an advanced mode, with more functions, when the device is rotated