Recap from last week

• Last week:
  • Android history & software stack
  • *(Gradle)*
  • Hello World

• This week:
  • Kotlin
  • Activity lifecycle
  • Android UI concepts
Application priorities, process states

• Applications have limited control over their own lifecycles
• Recall: applications run in separate processes, each one in a separate ART virtual machine instance
• Memory and process management is handled by the runtime and kernel
  • Runtime may kill some services in the background
  • Being aware of application states & priorities is critical
Process lifecycle, state hierarchy

Android uses this hierarchy when freeing up resources:

1. Foreground process (active process)
   - Few simultaneously, killed as last resort

2. Visible process
   - Activity that is visible to the user on-screen but not in the foreground.
     - E.g., if the foreground Activity is displayed as a dialog that allows the previous Activity to be seen behind it.

3. Service process
   - such as background network data upload or download

4. Cached process
   - *First to be killed, killing shouldn’ts affect UX*

Let’s look at the default empty activity

```java
package com.example.myapplication

import ...

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

Apps move through states during the lifecycle.

Understanding the lifecycle is crucial, to prevent:

- Apps crashing if the app’s normal flow is interrupted
- Using valuable resources when the user is not actively using your app
- Loss of user’s progress when they leave your app and return to it later
- Loss of progress or crashes when screen switches between landscape and portrait orientation

[Diagram showing the activity lifecycle]

Logging in Android

• Traditional `System.out.println` is generally not used on Android (output may not be visible, depending on device)

• Instead, use the frameworks logging mechanisms:
  • `android.util.Log` class and its methods
  • For example `Log.v(tag: String, message: String)`
    • Tag: app name, view name, usually a constant per class
    • Message: the actual log message
  • `Log.v(..), Log.i(..), Log.w(..), Log.e(..)`

• Log messages appear in the LogCat component of the Android Studio interface

https://developer.android.com/studio/debug/am-logcat
Demo of LogCat

- Let’s override another lifecycle method and log from there (onDestroy)
- What happens when device is rotated?
UI Creation in Android

• Three approaches:
  • Declaring in XML
  • Visually
    • Android Studio Layout Editor
  • Programmatically

• All based on same set of structured, pre-built UI components from Android framework

• View inflation

• Android View Hierarchy
Viewgroup and View

• A hierarchy of **Viewgroup** and **View** objects
• **View** – something the user can see/interact with
• **Viewgroup** – an invisible container for structuring other view & viewgroup objects

https://developer.android.com/guide/topics/ui/declaring-layout
View items (Widgets)

• EditText
• TextView
• Button
• Switch
• Checkbox
• RadioButton
• Spinner (dropdown list)
Figure 3.4 Class diagram of the Android View API, showing the root View class and specializations from there. Note that ViewGroup classes such as layouts are also a type of View.

ViewPager items

- Contain other view elements
- Invisible
- Define the positioning
- Support nesting
ViewGroup examples

- **Layouts**
  - **LinearLayout**: single row (column) of items
  - **ConstraintLayout**: flat hierarchy
  - **FrameLayout**: each child a layer
  - **TableLayout**
    - similar to HTML tables
    - consist of TableRow objects
  - **GridLayout**: index-based row and column layout
ViewGroup examples (2)

- ListView
- GridView
- ScrollView
- Toolbar
- AdapterView – contents are handled by *adapter*
XML attributes

• View objects let you specify various XML attributes
  • `textSize` for `TextView`
  • `hint` for `EditText`

• While there are lots of object-specific attributes, some are shared:
  • `ID`
  • `layout_width`, `layout_height`
    • `MATCH_PARENT`: Expand the View to fill the space of parent container (Layout)
    • `WRAP_CONTENT`: Expand the View just enough to fit the contents of the View (e.g. the text contained in `TextView`)
  • Fixed: e.g. 10dp (density-independent pixel) (https://developer.android.com/training/multiscreen/screendensities)
Attributes example with LinearLayout

• **orientation** ➔ horizontal, vertical
• **gravity**: “Specifies how an object should position its content, on both the X and Y axes, within its own bounds. “

• For children of a LinearLayout:
  • **android:layout_weight** – specify how to divide remaining space
Linear Layout Example

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:background="#FF0000"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:background="#00FF00"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:background="#0000FF"/>

</LinearLayout>
```
Nested LinearLayouts

LinearLayout <vertical>

LinearLayout <vertical>

LinearLayout <horizontal>
RelativeLayout

• A Layout where the location for Views can be described:
  • Relative to other Views added ("to the left of X")
  • Relative to the parent RelativeLayout container ("aligned to the container bottom")

• Suggested for use over nested LinearLayouts
  • More complex, deep the nesting of a layout, the longer to inflate
  • Especially with weights

https://developer.android.com/guide/topics/ui/layout/relative
https://developer.android.com/training/improving-layouts/optimizing-layout
RelativeLayout attributes

`layout_alignParentTop`
- If "true", makes the top edge of this view match the top edge of the parent.

`layout_centerVertical`
- If "true", centers this child vertically within its parent.

`layout_below`
- Positions the top edge of this view below the view specified with a resource ID.

`layout_toRightOf`
- Positions the left edge of this view to the right of the view specified with a resource ID.
Example RelativeLayout

- A flat layout!
  -.ImageView
    - android:layout_centerHorizontal="true"
    - android:layout_centerVertical="true"
  - Button
    - android:layout_below="@id/image"
    - android:layout_centerHorizontal="true"
  - SearchView
    - android:layout_alignTop="@id/button"
    - android:layout_toRightOf="@id/button"
  - CheckBox
    - android:layout_alignBottom="@+id/button"
    - android:layout_toRightOf="@+id/button"
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout 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">
    <ImageView
        android:id="@+id/logo"
        android:src="@mipmap/ic_launcher"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true" />
    <Button
        android:id="@+id/button"
        android:text="@string/button_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/logo"
        android:layout_centerHorizontal="true" />
    <SearchView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignTop="@id/button"
        android:layout_toRightOf="@id/button" />
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/checkbox_text"
        android:layout_alignBottom="@+id/button"
        android:layout_toRightOf="@+id/button" />
</RelativeLayout>
ConstraintLayout

• Effectively similar to RelativeLayout
  • All views are laid out according to the *constraint* rules specified
  • More performant
  • More flexible
    • E.g. bias options and negative margins

• Visual design
  • Takes advantage of Layout Editor’s visual tools, thus improving the visual design process
  • (Layout Editor was purpose-built for ConstraintLayout)

• Thus, further helps avoid nesting

https://developer.android.com/training/constraint-layout/
ConstraintLayout

• Recall, every object must have at least 2 constraints: 1 horizontal and 1 vertical
• By default, all constraints also have margins

https://developer.android.com/training/constraint-layout/
Opposing constraints

With two constraints on the same axis, a View is:

• **Centered** when size is set to "wrap_content" or a fixed value

• **Stretched** when size is set to `match_constraint`

https://developer.android.com/training/constraint-layout/
Constraint bias

• The centering behaviour is due to the default bias value of 50%

• Adjust the Bias to get other alignments

```
app:layout_constraintHorizontal_bias="0.25"
```
Layout Editor: Attributes pane

• Easiest way to adjust various attributes, including:
  • Bias
  • Margins
  • Constraints
  • ID
Chaining

A group of views linked bi-directionally

1. Evenly (default)
2. Spread Inside
3. Weighted
   With (1) and (2) and match_constraints + app:layout_constraintHorizontal_weight
4. Packed
   app:layout_constraintHorizontal_chainStyle="packed"

https://developer.android.com/training/constraint-layout/#constrain-chain
Guidelines, Ratios

- **Guidelines**
  - Are not rendered during runtime
  - Allow attaching constraints
  - Help you create a structured design

- **Ratio**
  - Let’s you constrain an object to a given aspect ratio
  - Have to set match_constraint
Programmatic UI creation

• Let’s create a vertical LinearLayout
• Add 2 buttons to it
XML-based approach

- Declarative approach
- Stored in /res/layout
- Straightforward XML vocabulary for view elements, such as widgets and layouts
- Declare the relationship and attributes of components
- During compiling, each XML layout is compiled into a View resource
- View resource is loaded via the `setContentView()` method
Programmatically vs declaratively in XML

```kotlin
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val mLayout = LinearLayout(context = this)
        mLayout.orientation = LinearLayout.HORIZONTAL

        val button1 = Button(context = this)
        button1.text = "First Button"

        val button2 = Button(context = this)
        button2.text = "Second Button"

        mLayout.addView(button1)
        mLayout.addView(button2)

        setContentView(mLayout)
    }
}
```

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <Button
        android:id="@+id/A"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Press A!" />

    <Button
        android:id="@+id/B"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Press B!" />

</LinearLayout>
```
String resources

• Notice the warnings the IDE shows you

    ```kotlin
    val button1 = Button(context = this)
    button1.text = "First Button"
    ```

• Android provides a system of managing resources like strings, numbers, colors, etc.
  • Open `app > res > values > strings.xml`.
  • Create a string value
  • Use it for your button
Working with string resources

• strings.xml:

```
<string name="hello">Greetings!</string>
<string-array name="planets_array">
    <item>Mercury</item>
    <item>Venus</item>
    <item>Earth</item>
</string-array>
<string name="message">Hello, %1$s.</string>
```

• Kotlin:

```kotlin
val string = getString(R.string.hello)

// String array
val stringArray: Array<String> = resources.getStringArray(R.array.planets_array)

// Formatted string
val text = getString(R.string.message, username)
```

Other resources

• Colour
  • accessed via the R.color class
    ```kotlin
    val color: Int = resources.getColor(R.color.pink)
    ```

• Style, dimension resources
  • Accessed from the R.style, R.dimen classes
  • Define various UI aspects such as margins, action bar configurations

https://developer.android.com/guide/topics/resources/more-resources
Resource file benefits

• Separation of code & concerns
  • Easy to find, update – e.g. no hardcoded strings, colours in code

• Organizing, structuring resources
  • Localization – region, language
  • Screen orientation
  • Device physical configuration – external keyboard, tablet

• Not just strings & layouts!
  • Color schemes, Fonts, Animations
  • Etc

https://developer.android.com/guide/topics/resources/available-resources
Providing resource alternatives

• Adjust to device configuration

• Create a resource subdirectory with the structure:
  •  `<resources_name>`-`<qualifier>`

  •  resource_name
    •  Type of resource
  •  Qualifier
    •  An individual configuration
    •  Can be chained with dashes

• How qualifiers are filtered:
  •  [https://developer.android.com/guide/topics/resources/providing-resources#BestMatch](https://developer.android.com/guide/topics/resources/providing-resources#BestMatch)

  [https://developer.android.com/guide/topics/resources/providing-resources#AlternativeResources](https://developer.android.com/guide/topics/resources/providing-resources#AlternativeResources)
Supporting different screens

• Create additional res/layout directories for alternative layouts
  • Smallest width:
    res/layout/main_activity.xml  # For handsets (smaller than 600dp available width)
    res/layout-sw600dp/main_activity.xml  # For 7” tablets (600dp wide and bigger)
  • Available width:
    res/layout/main_activity.xml  # For handsets (smaller than 600dp available width)
    res/layout-w600dp/main_activity.xml  # For 7” tablets or any screen with 600dp
    # available width (possibly landscape handsets)
  • Orientation:
    res/layout/main_activity.xml  # For handsets
    res/layout-land/main_activity.xml  # For handsets in landscape
    res/layout-sw600dp/main_activity.xml  # For 7” tablets
    res/layout-sw600dp-land/main_activity.xml  # For 7” tablets in landscape

https://developer.android.com/training/multiscreen/screensizes
Localizing applications

• Support different languages with
  • res/values-en
  • res/values-fr
  • res/values-est

• ISO 639 language codes

• Works for any resource – strings, mipmap images, ..

• Good practice means including a default value as well
  • Just “/values”, without qualifiers

https://developer.android.com/training/basics/supporting-devices/languages
https://developer.android.com/guide/topics/resources/localization
Re-cap

• Kotlin 101
  • Null-Safety

• Activity Lifecycle

• Android View Hierarchy
  • Items and Layouts

• UI creation
  • Programmatic + XML based

• Working with resources
Next week

• Navigating between activities
  • Intents & Intent Filters
• Application Components
• Fragments
• Context