UNIVERSITY OF PLYMOUTH

COMP2000: Software engineering 2

Android Development – Introduction

Outline

- Overview of mobile apps
- Getting started
- Application Fundamentals
- App components
- User interfaces

Mobile apps

- by definition, are software applications that run on portable devices such as smartphones and tablets. These apps are available for download through device-specific portals like Google Play Store and are installed onto users' devices.
- There are three types of mobile applications you can pick from:
- Native (this type will be considered in this module)
- Web
- Hybrid

Any idea what native app means?

- Menti.com

Examples of Native apps

- Menti.com

Native Apps

https://magenest.com/en/native-app-example/

- **Native Apps –** Native mobile apps are programmed in the recommended language for a specific operating system. For example, Android native applications are programmed using Java.
- **Web Apps –** Mobile web applications are websites that look and feel like native applications. However, they have limited access to the device's features and require permission before interacting with them.
- Written in HTML 5 most of the time, they require your device's native browser to run. A good example of this is Google Maps.
- **Hybrid apps-** These applications are available through the app store and allow users to interact with the device's features. However, they rely on HTML being rendered in a browser that's embedded within the app. A good example of hybrid applications is the Netflix app, which uses the same code to run on both desktop and mobile devices.

Native Android apps

- Apps written specifically for the Android Platform.

Component of Android

- Android is made up of several necessary and dependent parts, including the
- following:
- A Compatibility Definition Document (CDD) and Compatibility Test Suite (CTS) that describe the capabilities required for a device to support the software stack.

- A Linux operating system kernel that provides a low-level interface with the hardware, memory management, and process control, all optimized for mobile and embedded devices.

- Open-source libraries for application development, including SQLite, WebKit, OpenGL, and a media manager.

- A run time used to execute and host Android applications, including the Dalvik Virtual Machine (VM) and the core libraries that provide Android-specific functionality. The run time is designed to be small and efficient for use on mobile devices.

- An application framework that agnostically exposes system services to the application layer, including the window manager and location manager, databases, telephony, and sensors.
- A user interface framework used to host and launch applications.
- A set of core pre-installed applications.
- A software development kit (SDK) used to create applications, including the related tools, plug-ins, and documentation.
- Application program interface APIs provides access to all the underlying services, features, and hardware

Getting Started

Download Android SDK
Download the latest version of Android studio:

https://developer.android.com/studio

IntelliJ and Android studio are installed in the **web development lab smb104, 108 and 109**

Application Fundamentals

- Android apps can be written using Kotlin, Java, and C++ languages.

- The Android SDK tools compile your code along with any data and resource files into an APK or an Android App Bundle.

- Each Android app lives in its own security sandbox, protected by the following Android security features:

- The Android operating system is a multi-user Linux system in which each app is a different user.

- By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app).

- Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.

- By default, every app runs in its own Linux process.

- The Android system starts the process when any of the app's components need to be executed, and then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

Important files

app > java > com.example.myfirstapp > MainActivity

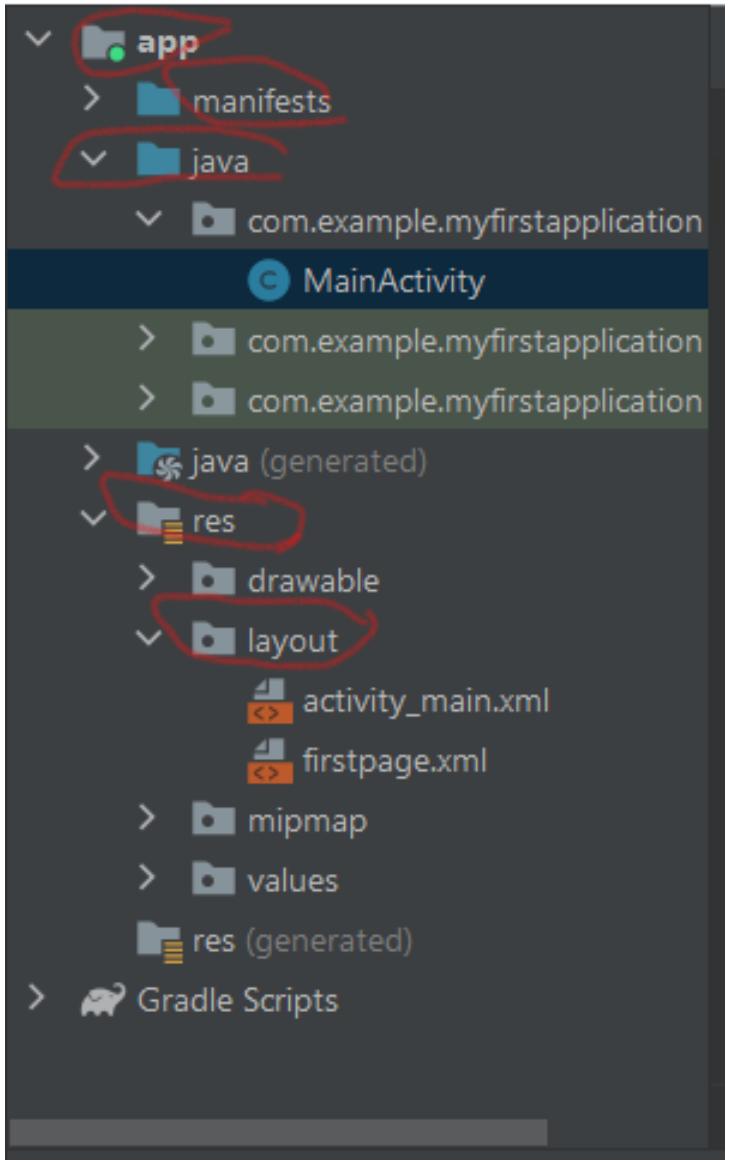
This is the main activity. It's the entry point for your app. When you build and run your app, the system launches an instance of this Activity and loads its layout.

app > res > layout > activity_main.xml

This XML file defines the layout for the activity's user interface (UI). It contains a TextView element with the text "Hello, World!“

app > manifests > AndroidManifest.xml

The manifest file describes the fundamental characteristics of the app and defines each of its components.

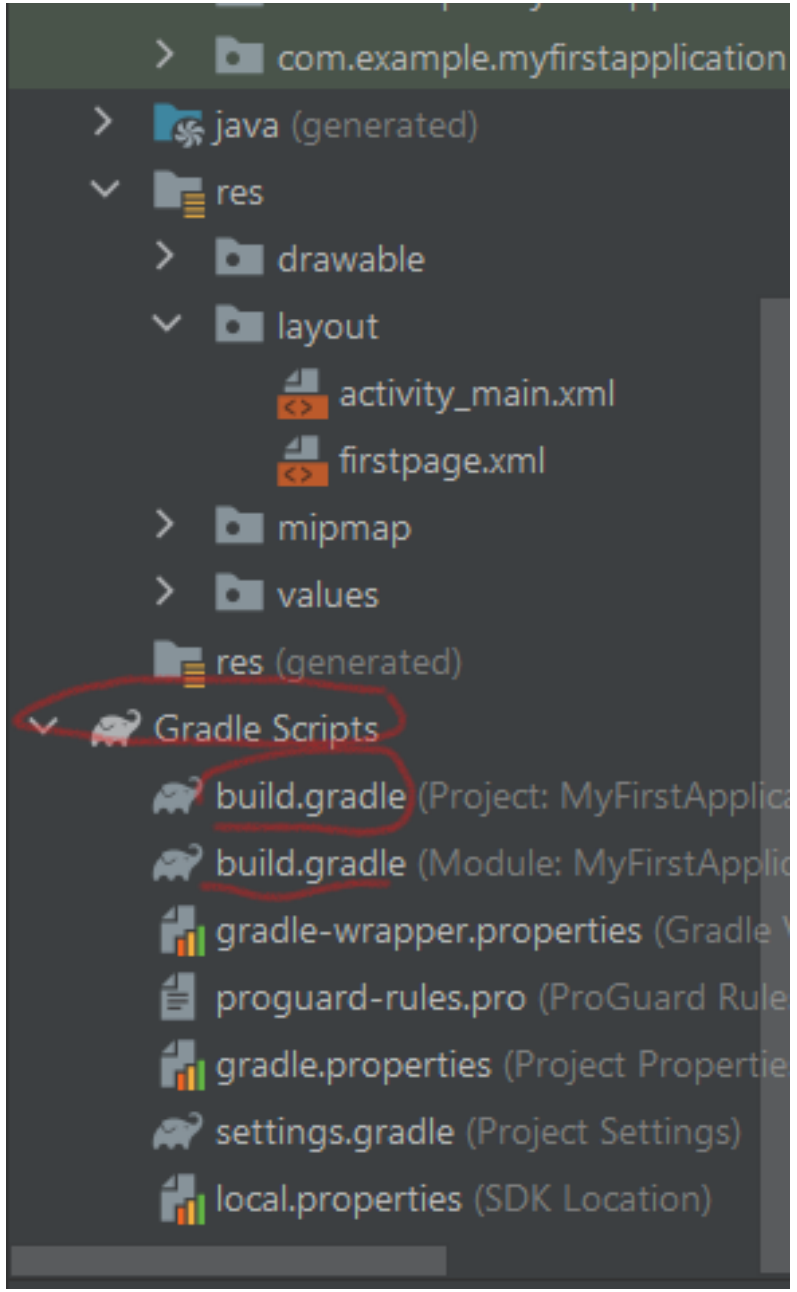


Gradle Scripts > build.gradle

There are two files with this name: one for the project, "Project: My_First_App," and one for the app module, "Module: My_First_App.app.“

Each module has its own build.gradle file, but this project currently has just one module.

Use each module's build.gradle file to control how the Gradle plugin builds your app.



App components

- App components are the essential building blocks of an Android app.

- There are four different types of app components:

- Activities
- Services
- Broadcast receivers
- Content providers

- An ***activity*** is the entry point for interacting with the user. It represents a single screen with a user interface.

- A ***service*** is a general-purpose entry point for keeping an app running in the background for all kinds of reasons.

- A ***broadcast receiver*** is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements.

- A ***content provider*** manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access.

- Three of the four component types: activities, services, and broadcast receivers—are activated by an asynchronous message called an *intent*.

- Intents bind individual components to each other at runtime.

- An intent is created with an Intent object (*will explain this later*).

The manifest file

- Before the Android system can start an app component, the system must know that the component exists by reading the app's *manifest file*, *AndroidManifest.xml.*
- Your app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as the following:

- Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- Declares the minimum API Level required by the app, based on which APIs the app uses.
- Declares hardware and software features used or required by the app, such as a camera, Bluetooth services, or a multitouch screen.
- Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.

How to Declare components?

- The primary task of the manifest is to inform the system about the app's components. For example, a manifest file can declare an activity as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:icon="@drawable/app_icon.png" ... >
        <activity android:name="com.example.project.ExampleActivity"
                  android:label="@string/example_label" ... >
        </activity>
        ...
    </application>
</manifest>
```

In the <application> element, the android:icon attribute points to resources for an icon that identifies the app.

In the <activity> element, the android:name attribute specifies the fully qualified class name of the Activity subclass and the android:label attribute specifies a string to use as the user-visible label for the activity.

You must declare all app components using the following elements:

- <activity> elements for activities.
- <service> elements for services.
- <receiver> elements for broadcast receivers.
- <provider> elements for content providers.

the Android Security model allows an app to do pretty much anything it likes. But with pretty much everything the app wants to do, e.g. to make network connections, you must ask for permission to do this in its manifest.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.INTERNET"/>
```

Introduction to Activity

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.

Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the main activity, which is the first screen to appear when the user launches the app.

- To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element. For example:

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

Activity-lifecycle concepts

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks:

- onCreate(): You must implement this callback, which fires when the system creates your activity. Your implementation should initialize the essential components of your activity.

- onStart(): This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.

- onResume(): The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input.

- onPause(): The system calls on Pause()when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or Recents button.

- onStop(): The system calls on Stop()when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity.

- onRestart(): The system invokes this callback when an activity in the Stopped state is about to restart.onRestart()restores the state of the activity from the time that it was stopped.

- onDestroy(): The system invokes this callback before an activity is destroyed. This callback is the final one that the activity receives.

The system invokes each of these callbacks as an activity enters a new state.

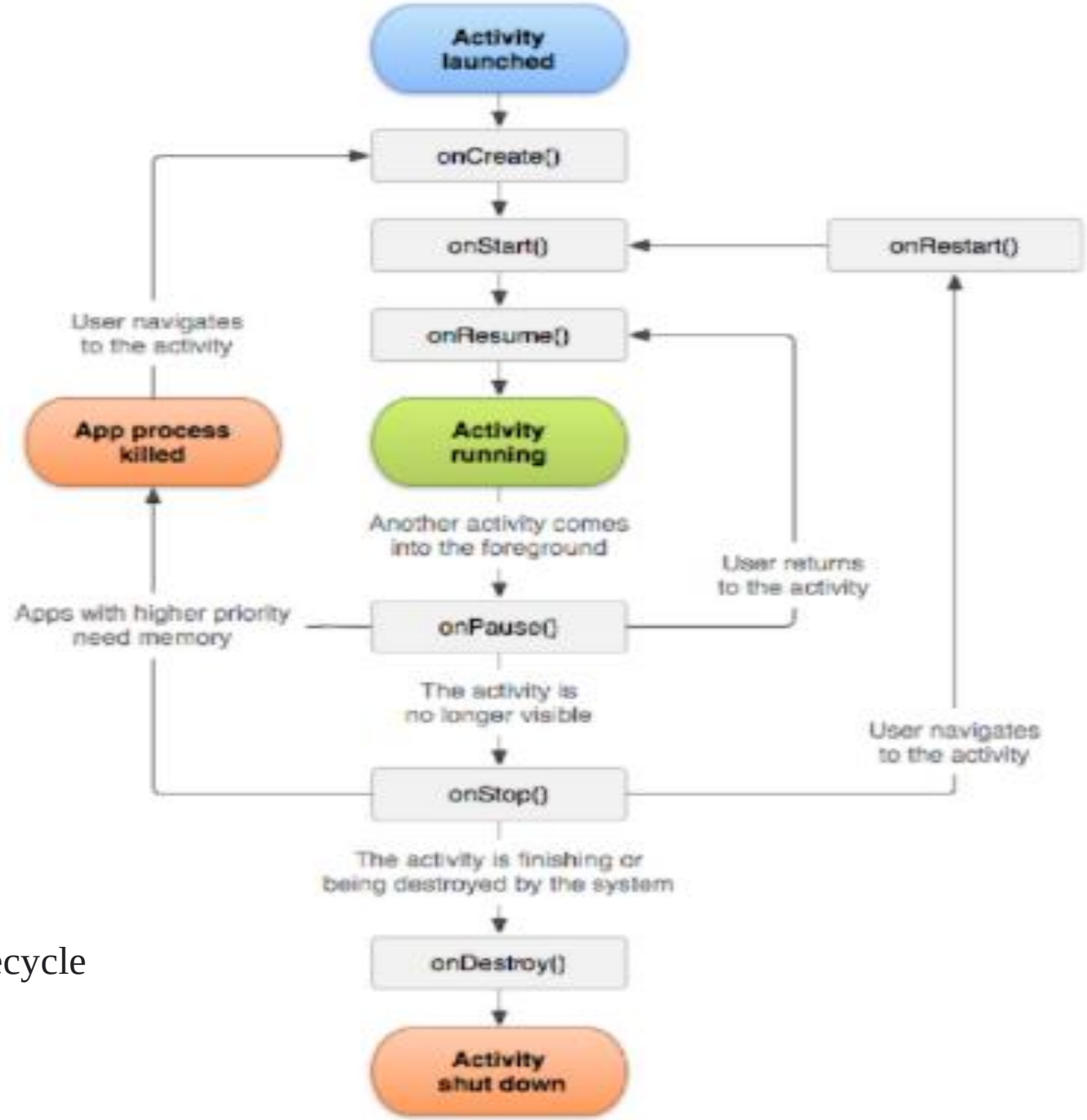


Figure 1. A simplified illustration of the activity lifecycle (Android Developer website).

Android User Interface Fundamentals

- All visual components in Android descend from the View class and are referred to generically as *Views*. You'll often see Views referred to as *controls* or *widgets*— terms you're probably familiar with if you've previously done any GUI development.

- The ViewGroup class is an extension of View designed to contain multiple Views. View Groups are used most commonly to manage the layout of child Views, but they can also be used to create atomic reusable components.

- View Groups that perform the former function are generally referred to as *layouts*.

- The user interface (UI) for an Android app is built as a hierarchy of *layouts* and *widgets*.

- The layouts are ViewGroup objects, containers that control how their child views are positioned on the screen. Widgets are View objects, UI components such as buttons and text boxes.

Layout

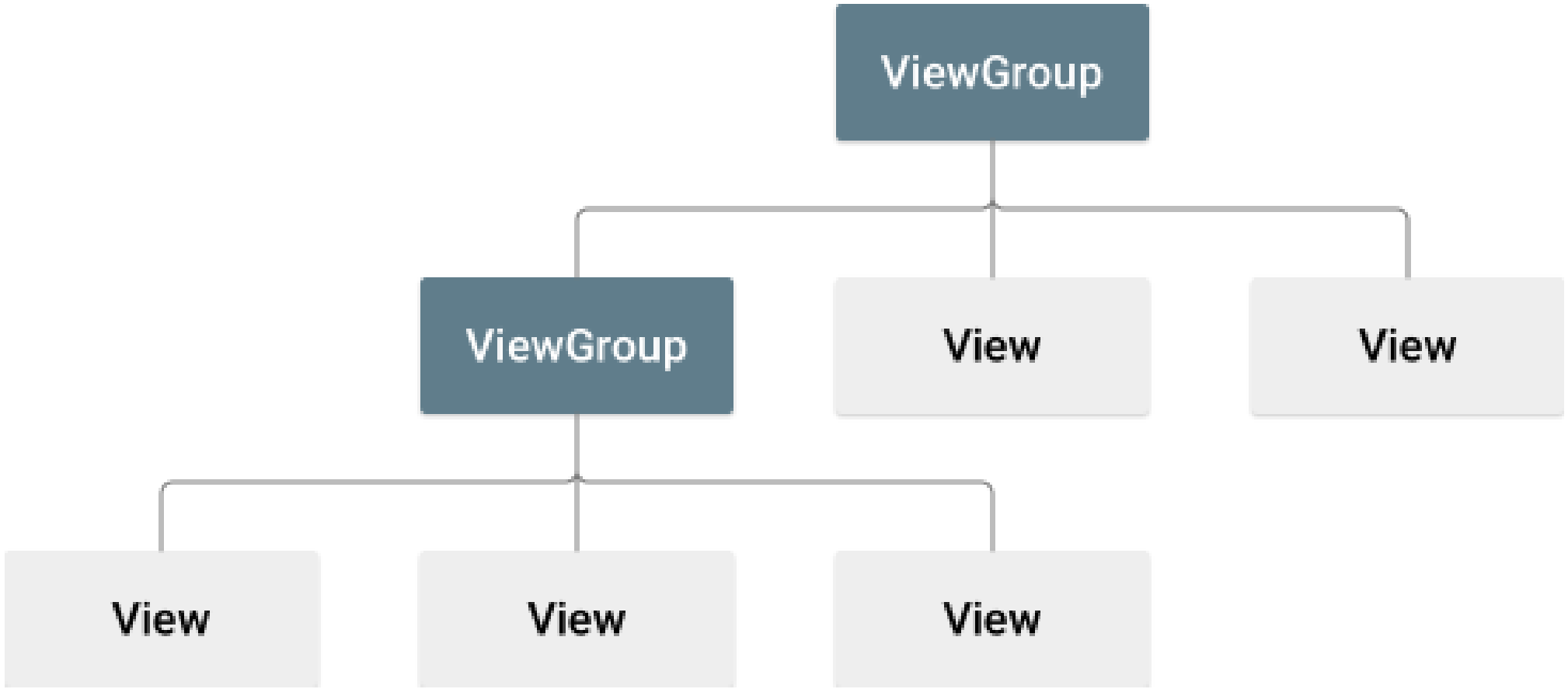
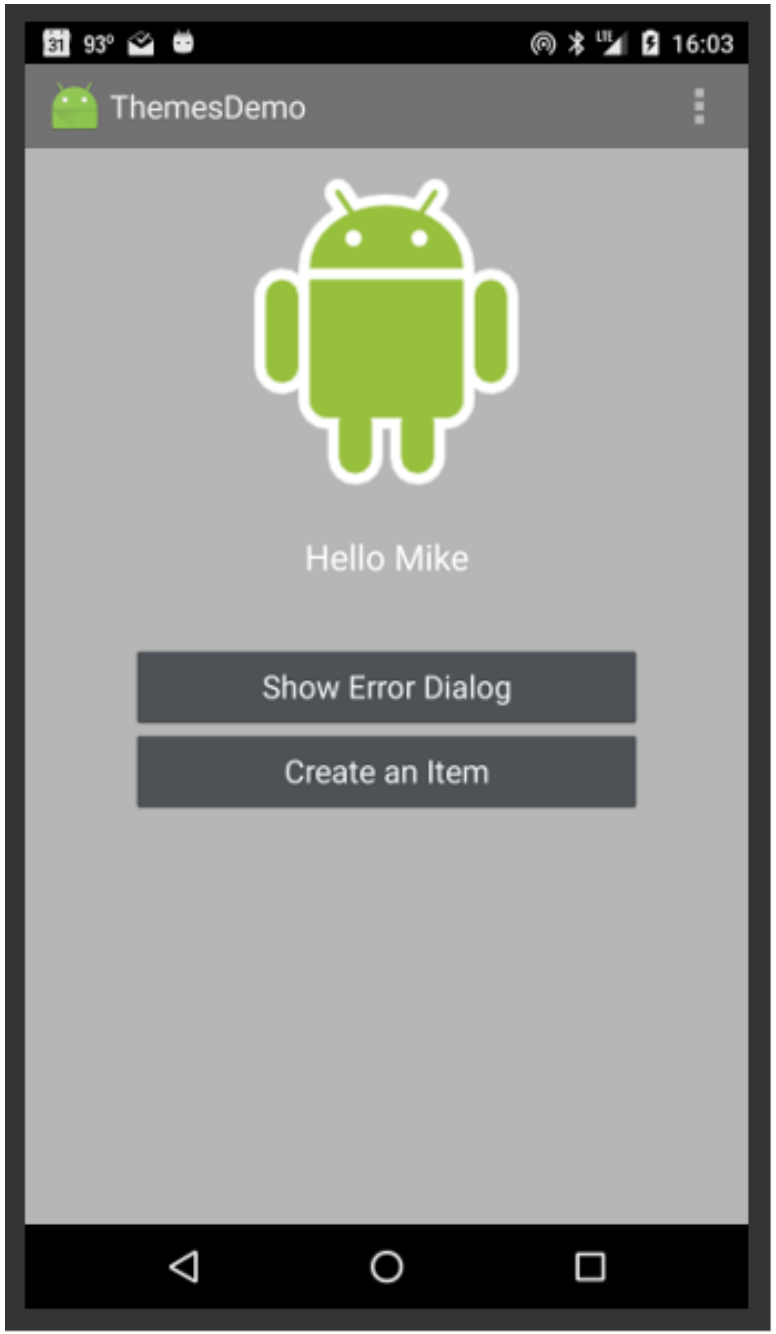


Figure 1. Illustration of a view hierarchy, which defines a UI layout (Android developer website)

User interface



http://eagle.phys.utk.edu/guidry/android/androidUserInterface.html

- **Views** — Views are the base class for all visual interface elements (commonly known as *controls* or *widgets*). All UI controls, including the layout classes, are derived from View.

- **View Groups** — View Groups are extensions of the View class that can contain multiple child Views. Extend the ViewGroup class to create compound controls made up of interconnected child Views.

- The **ViewGroup** class is also extended to provide the Layout Managers that help you lay out controls within your Activities.

- **Fragments** — Fragments, are used to encapsulate portions of your UI.
- This encapsulation makes Fragments particularly useful when optimizing your UI layouts for different screen sizes and creating reusable UI elements.
-
- **Activities** — Activities, will be described in detail later, represent the window, or screen, being displayed.
- Activities are the Android equivalent of Forms in traditional Windows desktop development.

- To display a UI, you assign a View (usually a layout or Fragment) to an Activity.

- Android provides several common UI controls, widgets, and Layout Managers.
- For most graphical applications, it’s likely that you’ll need to extend and modify these standard Views — or create composite or entirely new Views — to provide your own user experience.

Introducing Layouts

- *Layout Managers* (or simply *layouts*) are extensions of the ViewGroup class and are used to position child Views within your UI.
- Layouts can be nested, letting you create arbitrarily complex UIs using a combination of layouts
- The Android SDK includes a number of layout classes. You can use these, modify them, or create your own to construct the UI for your Views, Fragments, and Activities.
- It's up to you to select and use the right combination of layouts to make your UI aesthetically pleasing, easy to use, and efficient to display.
- The following list includes some of the most commonly used layout classes available in the Android SDK:

- FrameLayout — The simplest of the Layout Managers, the Frame Layout pins each child view within its frame.
- The default position is the top-left corner, though you can use the *gravity* attribute to alter its location. Adding multiple children stacks each new child on top of the one before, with each new View potentially obscuring the previous ones.
- LinearLayout — A Linear Layout aligns each child View in either a vertical or a horizontal line. A vertical layout has a column of Views, whereas a horizontal layout has a row of Views.
- The Linear Layout supports a *weight* attribute for each child View that can control the relative size of each child View within the available space.
- RelativeLayout — One of the most flexible of the native layouts, the Relative Layout lets you define the positions of each child View relative to the others and to the screen boundaries.

- GridLayout — Introduced in Android 4.0 (API level 14), the Grid Layout uses a rectangular grid of infinitely thin lines to lay out Views in a series of rows and columns.
- The Grid Layout is incredibly flexible and can be used to greatly simplify layouts and reduce or eliminate the complex nesting often required to construct UIs using the layouts described above.

- Each of these layouts is designed to scale to suit the host device's screen size by avoiding the use of absolute positions or predetermined pixel values.
- This makes them particularly useful when designing applications that work well on a diverse set of Android hardware.

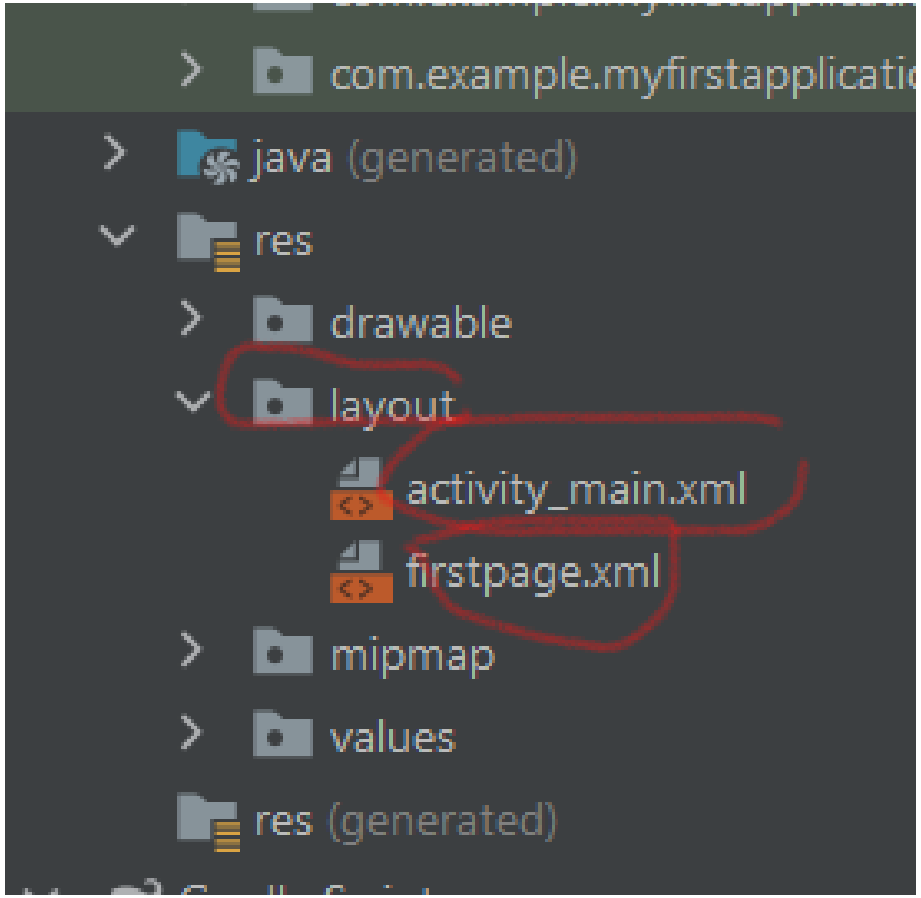
Layout resources

A layout resource defines the architecture for the UI in an Activity or a component of a UI.

File location:
res/layout/*filename*.xml
The filename will be used as the resource ID.

Compiled resource datatype:
Resource pointer to a View (or subclass) resource.

Resource reference:
In Java: R.layout.*filename*
In XML: @[*package*:]layout/*filename*



Defining Layouts

- The preferred way to define a layout is by using XML external resources.
- Each layout XML must contain a single root element. This root node can contain as many nested layouts and Views as necessary to construct an arbitrarily complex UI.
- The following snippet shows a simple layout that places a *TextView* above an *EditText* control using a vertical *LinearLayout*.

Syntax

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroup
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@[+][package:]id/resource_name"
    android:layout_height=["dimension" | "match_parent" | "wrap_content"]
    android:layout_width=["dimension" | "match_parent" | "wrap_content"]
    [ViewGroup-specific attributes] >
    <View
        android:id="@[+][package:]id/resource_name"
        android:layout_height=["dimension" | "match_parent" | "wrap_content"]
        android:layout_width=["dimension" | "match_parent" | "wrap_content"]
        [View-specific attributes] >
        <requestFocus/>
    </View>
    <ViewGroup >
        <View />
    </ViewGroup>
    <include layout="@layout/layout_resource"/>
</ViewGroup>
```

Note: The root element can be either a **ViewGroup**, a **View**, or a **<merge>** element, but there must be only one root element and it must contain the **xmlns:android** attribute with the **android** namespace as shown.

XML:

```
<TextView android:id="@+id/nameTextbox"/>
```

Java:

```
TextView textView = findViewById(R.id.nameTextbox);
```

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout
      xmlns:android=”http://schemas.android.com/apk/res/android”
       android:orientation=”vertical”
       android:layout_width=”match_parent”
       android:layout_height=”match_parent”>
<TextView
        android:layout_width=”match_parent”
        android:layout_height=”wrap_content”
        android:text=”Enter Text Below”
      />
<EditText
         android:layout_width=”match_parent”
         android:layout_height=”wrap_content”
         android:text=”Text Goes Here!”
      />
</LinearLayout>
```

- When preferred, or required, you can implement layouts in code.
- When assigning Views to layouts in code, it’s important to apply LayoutParameters using the setLayoutParams method, or by passing them in to the addView call:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);

TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);

myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");

int lHeight = LinearLayout.LayoutParams.MATCH_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;

ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight, lWidth));

setContentView(ll);
```

Assigning User Interfaces to Activities

- A new Activity starts with a temptingly empty screen onto which you place your UI.
- To do so, call *setContentView*, passing in the View instance, or layout resource, to display.
- Because empty screens aren't particularly inspiring, you will almost always use *setContentView* to assign an Activity's UI when overriding its *onCreate* handler.
- The setContentView method accepts either a layout's resource ID or a single View instance. This lets you define your UI either in code or using the preferred technique of external layout resources.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
```

- Using layout resources decouples your presentation layer from the application logic, providing the flexibility to change the presentation without changing code.
- This makes it possible to specify different layouts optimized for different hardware configurations, even changing them at run time based on hardware changes (such as screen orientation changes).
- You can obtain a reference to each of the Views within a layout using the *findViewById* method:

```
TextView myTextView = (TextView)findViewById(R.id.myTextView);
```

If you prefer the more *traditional* approach, you can construct the UI in code:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView myTextView = new TextView(this);
        setContentView(myTextView);
        myTextView.setText("Hello, Android");
}
```

- The *setContentView* method accepts a single View instance; as a result, you use layouts to add multiple controls to your Activity.

Example of a Layout

XML file saved at res/layout/main_activity.xml:

```
<?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" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

This application code will load the layout for an Activity, in the onCreate() method:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
}
```

- You need to define variable of the type you want to interact, so for a Button you will declare

  Button button1 = null;
- You need to get the instance of your component (View) in your layout.
- Your layout won’t exist until you execute the line

  setContentView(R.layout.calculator_layout);

  That inflates your view.

Plugging your program into your layout

- For the minute, put this in your Activity's method onCreate().
- Use findViewById() and the R thing.
- **The R thing** is a machine generated class, just a load of static ints, which maps onto your projects folder structure**.** **It won't build if there are errors in any of your XML documents.**
- You need to know the name you gave your component in your layout.
- Be careful using the XML layout to nominate the onClick() method.

Interacting with the many things

The pattern is always the same for sensors, touch events, location events etc.

- There will be a Java interface named *onSomethingOrOtherListener*
- That interface defines methods *onSomethingHashappened*. These methods have to be implemented somewhere.
- A component, say a simple Button will have various methods to set these listeners, like

  myButton.setOnClickListener(l)
- In this case, whatever l might be, it must implement the interface OnClickListener.

```
 button1.setOnClickListener(new OnClickListener()
{
@Override
public void onClick(View view)
{
   //Do something
}
});
```

Start your first Project

- To create your new Android project, follow these steps:
- Install the latest version of Android Studio.
- In the **Welcome to Android Studio** window, click **Start a new Android project**.
- If you have a project already opened, select **File** > **New** > **New Project**.
- In the **Choose your project** window, select **Empty Activity** and click **Next**.
- In the **Configure your project** window, complete the following:
  - Enter "My First App" in the **Name** field.
  - Enter "com.example.myfirstapp" in the **Package name** field.
  - If you'd like to place the project in a different folder, change its Save location.
  - Select either **Java** from the Language drop-down menu.
  - **Leave the other options as they are**.
- Click **Finish**

Resources

- Android developer website: https://developer.android.com/docs

Would you like to share what was clear/not clear in this lecture?

- Menti.com

Thank you