COMP-304
Winter2023

Lecture 4

Objectives:

- ❑ Apply Externalizing of Resources
  - ➢ Declare simple resource values - Strings, Integers, Booleans, Colors, Drawables, String Arrays, XML files, etc.
- ❑ Classify Layout Managers.
- ❑ Apply layout classes and simple UI controls in Android apps:
  - ▪ TextView
  - ▪ EditText
  - ▪ Button
- ❑ Utilize Recycler View in Android apps.
- ❑ Utilize data binding in Android apps.

Review of Lecture 3

- **Android Activities**:
  - **Activity:** a task in an Android application
  - Extends class Activity
  - Life Cycle:
    - onCreate
    - onStart
    - onRestart
    - onResume
    - onPause
    - onStop
    - onDestroy

- **Intents**
  - Used to call other activities and built-in apps
  - **startActivity** method
    - Source activity
    - Started activity
  - **Intent** objects
    - the action to be performed
    - the data to be acted upon
  - Passing information to other activities
    - putExtra
    - getExtras

Review of Lecture 3

```
val editText =
findViewById<EditText>(R.id.editText)
val message = editText.text.toString()
val intent = Intent(this,
DisplayMessageActivity::class.java).apply {
  putExtra(EXTRA_MESSAGE, message)
}
startActivity(intent)
```

- ❑ Get the Intent that started this activity and extract the string

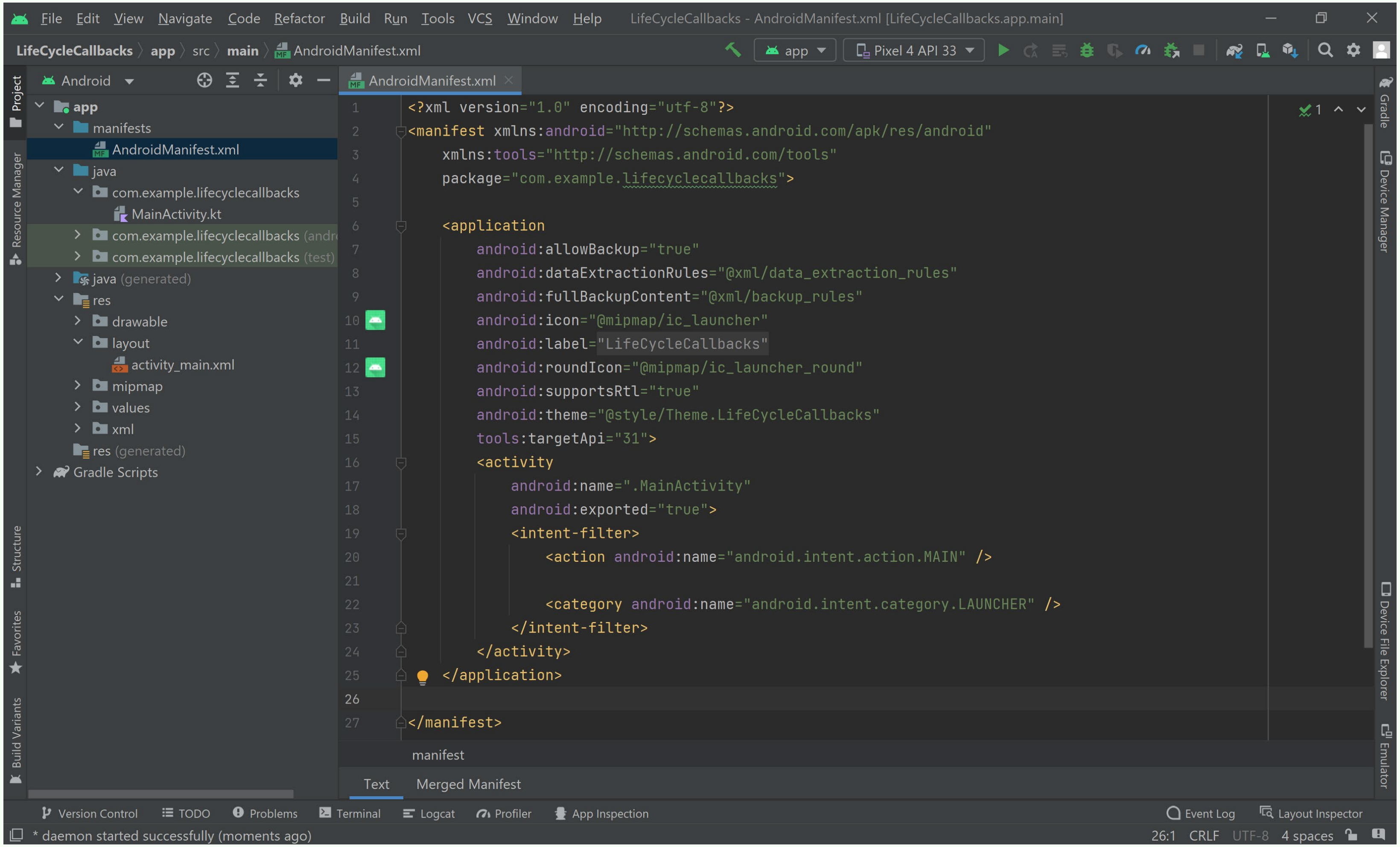
```
 val message =
intent.getStringExtra(EXTRA_MESSAGE
)
```

- ❑ Using **requireActivity** method within a fragment

```
when (checkedId) {
  R.id.radio_sety ->
{Toast.makeText(this@ProgramFragment.requ
ireActivity(),  "Software Engineering
Technology",
Toast.LENGTH_SHORT).show()}
```

- ❑ **Fragments**
  - ➢ A **mini-activity**, derives from Fragment class
  - ➢ Load the UI from xml files
  - ➢ Its life cycle methods are similar to activity life cycle methods:
    - ▪ onAttach, onCreate
    - ▪ onCreateView
    - ▪ onActivityCreate
    - ▪ onStart, onResume, onPause
    - ▪ onDestroyView
    - ▪ onDestroy, onDetach
- ❑ **FragmentManager and FragmentTransaction** classes – manage fragments dynamically

Android Manifest File



```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.example.lifecyclecallbacks">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="LifeCycleCallbacks"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.LifeCycleCallbacks"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

* daemon started successfully (moments ago)

Android Manifest File

- **Android Manifest File** - application configuration information
  - Application’s package name
  - The **components of the app**, which include all **activities**, services, broadcast receivers, and content providers
  - The **permissions**
  - Hardware and software **features**
- The application manifest file is in app/manifests folder
- Some attributes are defined in **build.gradle** file

Android Manifest File

- **Registering Activities**:

```
<activity android:name=".MyActivity" />
```

- Primary Point Activity:

```
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category
   android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

- Using **Features** and **permissions**:
  - **uses-feature** - specifies hardware and software features your application *requires* in order to properly function:

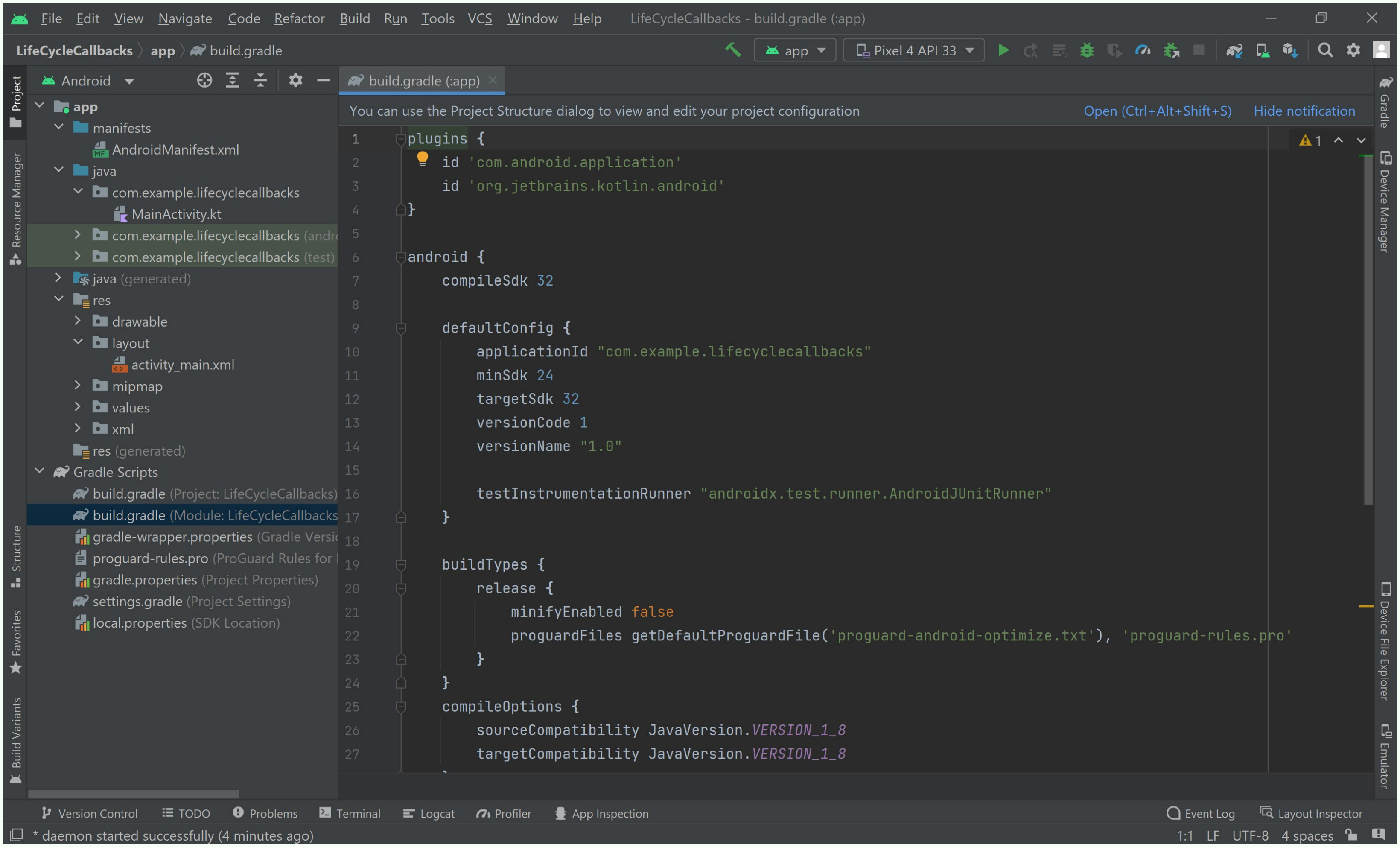
```
<uses-feature android:name="android.hardware.camera" />
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

  - Allows the API to determine as precise a location as possible

Configuring the Gradle Build

- ❑ Each project contains a series of Gradle files used to define your build configuration, consisting of a:
  - **Project-scoped settings.gradle** file that defines modules to be included
  - **Project-scoped build.gradle** file in which the repositories and dependencies for Gradle itself are specified, as well as any repositories and dependencies common to all your modules.
  - **Module-scoped build.gradle** file(s) used **to configure build settings for your application**, including dependencies, minimum and targeted platform versions, your application’s version information, and multiple build types and product flavors.

Configuring the Gradle Build



File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
LifeCycleCallbacks - build.gradle (:app)
LifeCycleCallbacks 〉 app 〉 build.gradle
app
Pixel 4 API 33
Android
build.gradle (:app)
app
manifests
AndroidManifest.xml
java
com.example.lifecyclecallbacks
MainActivity.kt
com.example.lifecyclecallbacks (andr
com.example.lifecyclecallbacks (test)
java (generated)
res
drawable
layout
activity_main.xml
mipmap
values
xml
res (generated)
Gradle Scripts
build.gradle (Project: LifeCycleCallbacks)
build.gradle (Module: LifeCycleCallbacks
gradle-wrapper.properties (Gradle Versi
proguard-rules.pro (ProGuard Rules for
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
Project
Resource Manager
Structure
Favorites
Build Variants
Gradle
Device Manager
Device File Explorer
Emulator
Open (Ctrl+Alt+Shift+S)
Hide notification
Version Control
TODO
Problems
Terminal
Logcat
Profiler
App Inspection
Event Log
Layout Inspector
1:1 LF UTF-8 4 spaces

You can use the Project Structure dialog to view and edit your project configuration

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

android {
    compileSdk 32

    defaultConfig {
        applicationId "com.example.lifecyclecallbacks"
        minSdk 24
        targetSdk 32
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
```

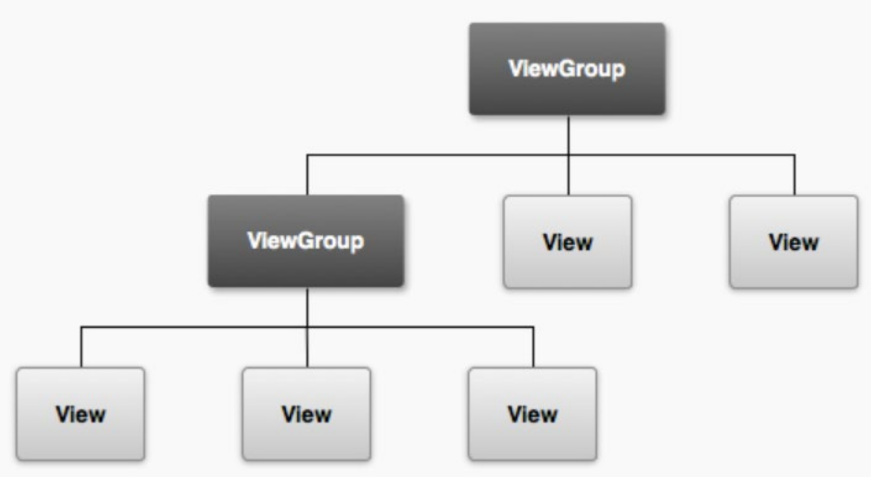
* daemon started successfully (4 minutes ago)

Android UI Toolkits

- ❑ Two toolkits for building native UIs in Android:
  - ➢ **Android View** system - resources are defined in XML
  - ➢ **Jetpack Compose** – uses a declarative API to describe the UI programmatically
- ❑ Compose is compatible with all your existing code: you can call Compose code from Views and Views from Compose
- ❑ We will learn Android View System first

Android Views

- android.**view** package contains a number of interfaces and classes related to drawing on the screen.
- android.view.**View** class is the basic user interface building block within Android.
  - It represents a rectangular portion of the screen.
- The View class serves as the **base class** for nearly all the user interface controls and layouts within the Android SDK.
- ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup object



Creating Layouts Using XML Resources

- Layouts and user interface controls can be **defined as application resources** or **created programmatically** at runtime
- Android provides a simple way to create layout files in XML as resources provided in the **/res/layout** project directory
  - This is the most common way
- Use Layouts to Create **Device-Independent User Interfaces**
  - A defining feature of the layout classes is their **ability to scale and adapt to a range of screen sizes, resolutions, and orient**ations.

Layout classes

- ❑ LinearLayout
- ❑ AbsoluteLayout
- ❑ TableLayout
- ❑ RelativeLayout
- ❑ ConstraintLayout
- ❑ FrameLayout
- ❑ ScrollView

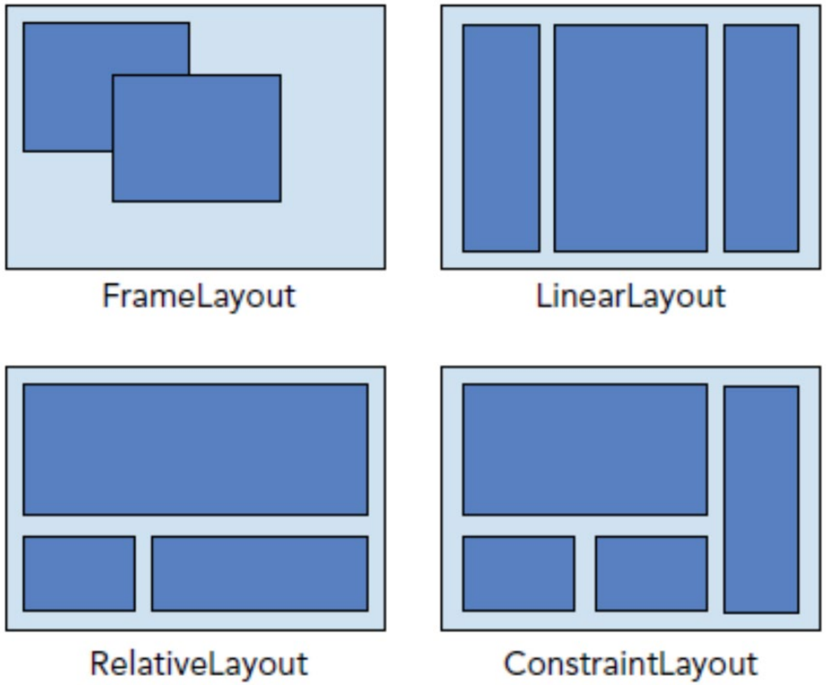


FIGURE 5-1

- ❑ AbsoluteLayout may be used to specify the exact x/y coordinate locations of each control on the screen instead, but this is **not easily portable across many screen resolutions**

Creating Layouts Using XML Resources

- You can configure almost any ViewGroup or View (or View subclass) attribute using the XML layout resource files
- **LinearLayout** is is one of the simplest layout classes. It aligns a sequence of child Views either **vertically** or **horizontally**:

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=
“http://schemas.android.com/apk/res/android”
android:orientation=”vertical”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent” >
<TextView
android:layout_width=”fill_parent”
android:layout_height=”wrap_content”
android:text=”@string/hello” />
</LinearLayout>
```

ViewGroup Attributes

- This example of a LinearLayout sets the size of the screen, containing one TextView that is set to its full **height** and the **width** of the LinearLayout (and therefore the screen):

```
<LinearLayout xmlns:android=
“http://schemas.android.com/apk/res/android”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”>
<TextView
android:id=”@+id/tv_name”
android:layout_height=”fill_parent”
android:layout_width=”fill_parent” />
</LinearLayout>
```

ViewGroup Attributes

- ❑ Here is an example of a **Button** object with some margins set via XML used in a layout resource file:

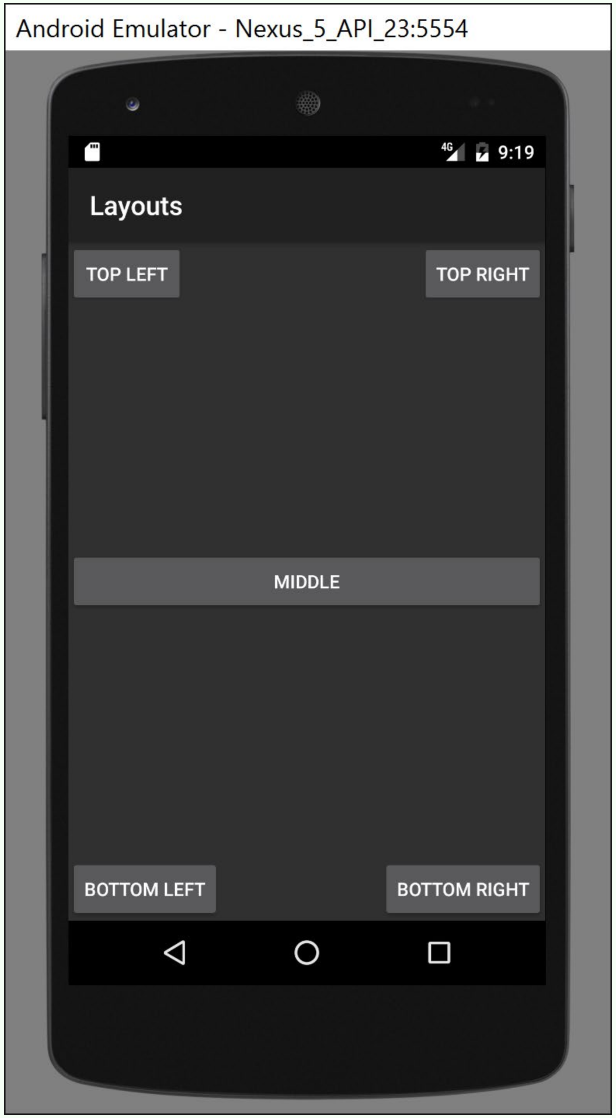
```
<Button
android:id=”@+id/btn_enter”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:text=”Press Me”
android:layout_marginRight=”20px”
android:layout_marginTop=”60px” />
```

Using **RelativeLayout**

- ❑ The **RelativeLayout** defines the position of each element within the layout **in terms of its parent and the other Views**.
- ❑ For instance, you can set a child View to be positioned “above” or “below” or “to the left of ” or “to the right of ” another View, referred to by its unique identifier.
- ❑ You can also align child View objects relative to one another or the parent layout edges.

Using **RelativeLayout**

- ❑ Combining RelativeLayout attributes can simplify creating interesting user interfaces without resorting to multiple layout groups to achieve a desired effect.
- ❑ The picture shows how each of the button controls is relative to each other



Using **RelativeLayout**

- Here’s an example of an XML layout resource with a RelativeLayout and two child View objects, a Button object aligned relative to its parent, and an ImageView aligned and positioned relative to the Button (and the parent):

```
<?xml version=”1.0” encoding=”utf-8”?>
<RelativeLayout xmlns:android=
“http://schemas.android.com/apk/res/android”
android:id=”@+id/RelativeLayout01”
android:layout_height=”fill_parent”
android:layout_width=”fill_parent”>
<Button
android:id=”@+id/btn_center”
android:text=”Center”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:layout_centerInParent=”true” />
<ImageView
android:id=”@+id/iv_center”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:layout_above=”@id/btn_center”
android:layout_centerHorizontal=”true”
android:src=”@drawable/arrow” />
</RelativeLayout>
```

Using **FrameLayout**

- **FrameLayout** view is designed to display **a stack of child View items**.
- You can add multiple views to this layout, but **each View is drawn from the top-left corner** of the layout.
- Use this to show multiple images within the same region, and the **layout is sized to the largest child View** in the stack.

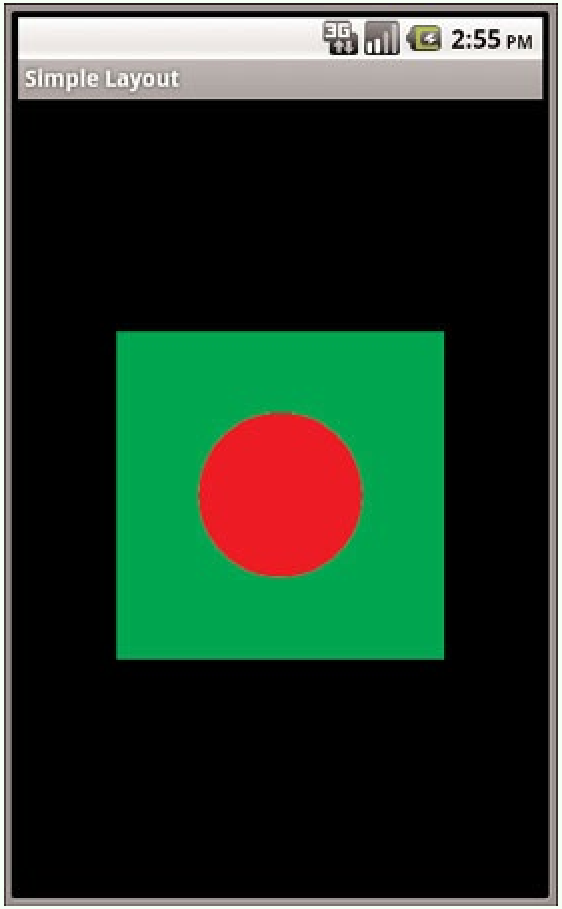
Using **FrameLayout**

- ❑ Here’s an example of an XML layout resource with a FrameLayout and **two child View objects**, both **ImageView** objects.
- ❑ The green rectangle is drawn first and the red oval is drawn on top of it.
- ❑ The green rectangle is larger, so it defines the bounds of the FrameLayout:

```
<FrameLayout xmlns:android=
“http://schemas.android.com/apk/res/android”
android:id=”@+id/fl_center”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:layout_gravity=”center”>
```

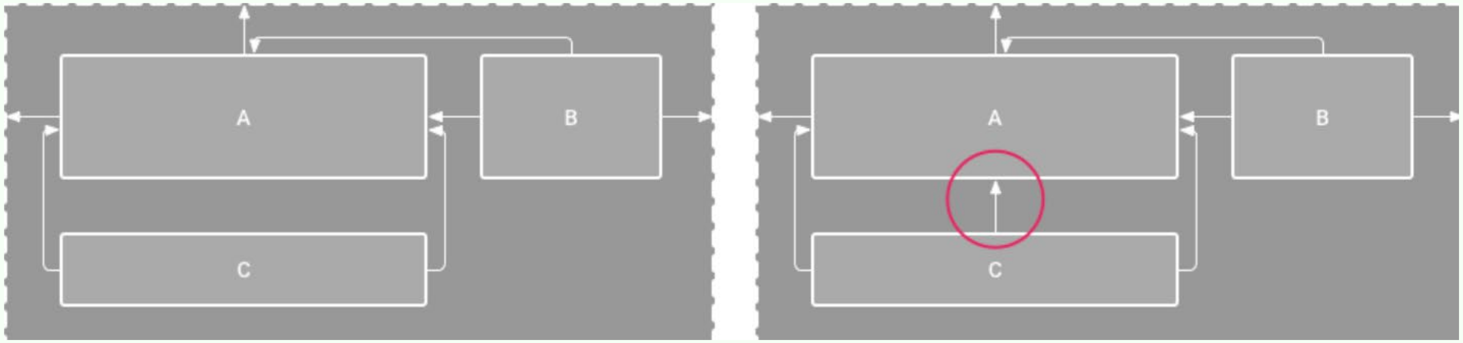
Using **FrameLayout**

```
<ImageView
android:id=”@+id/img1_center”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:src=”@drawable/green_rect”
android:minHeight=”200px”
android:minWidth=”200px” />
<ImageView
android:id=”@+id/img2_center”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:src=”@drawable/red_oval”
android:minHeight=”100px”
android:minWidth=”100px”
android:layout_gravity=”center” />
</FrameLayout>
```



ConstraintLayout

- ConstraintLayout allows you to create large and complex layouts with a flat view hierarchy (no nested view groups).
- It's similar to RelativeLayout in that **all views are laid out according to relationships between sibling views and the parent layout**, but it's **more flexible than RelativeLayout** and easier to use with Android Studio's Layout Editor.



- The editor shows view C below A, but it has no vertical constraint
- View C is now **vertically constrained below view A**

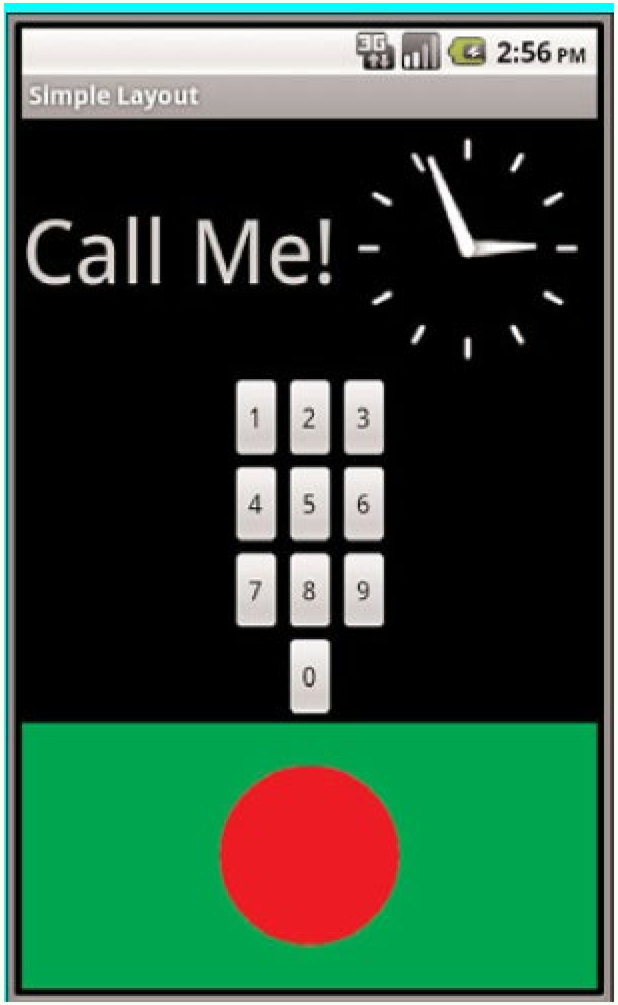
https://developer.android.com/training/constraint-layout/

ConstraintLayout

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
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"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/message_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Using Multiple Layouts on a Screen

- ❑ Combining different layout methods on a single screen can create complex layouts.
- ❑ Because a layout contains View objects and is, itself, a View, it can contain other layouts.
- ❑ The figure on the right demonstrates a combination of layout views used in conjunction to create a more complex and interesting screen

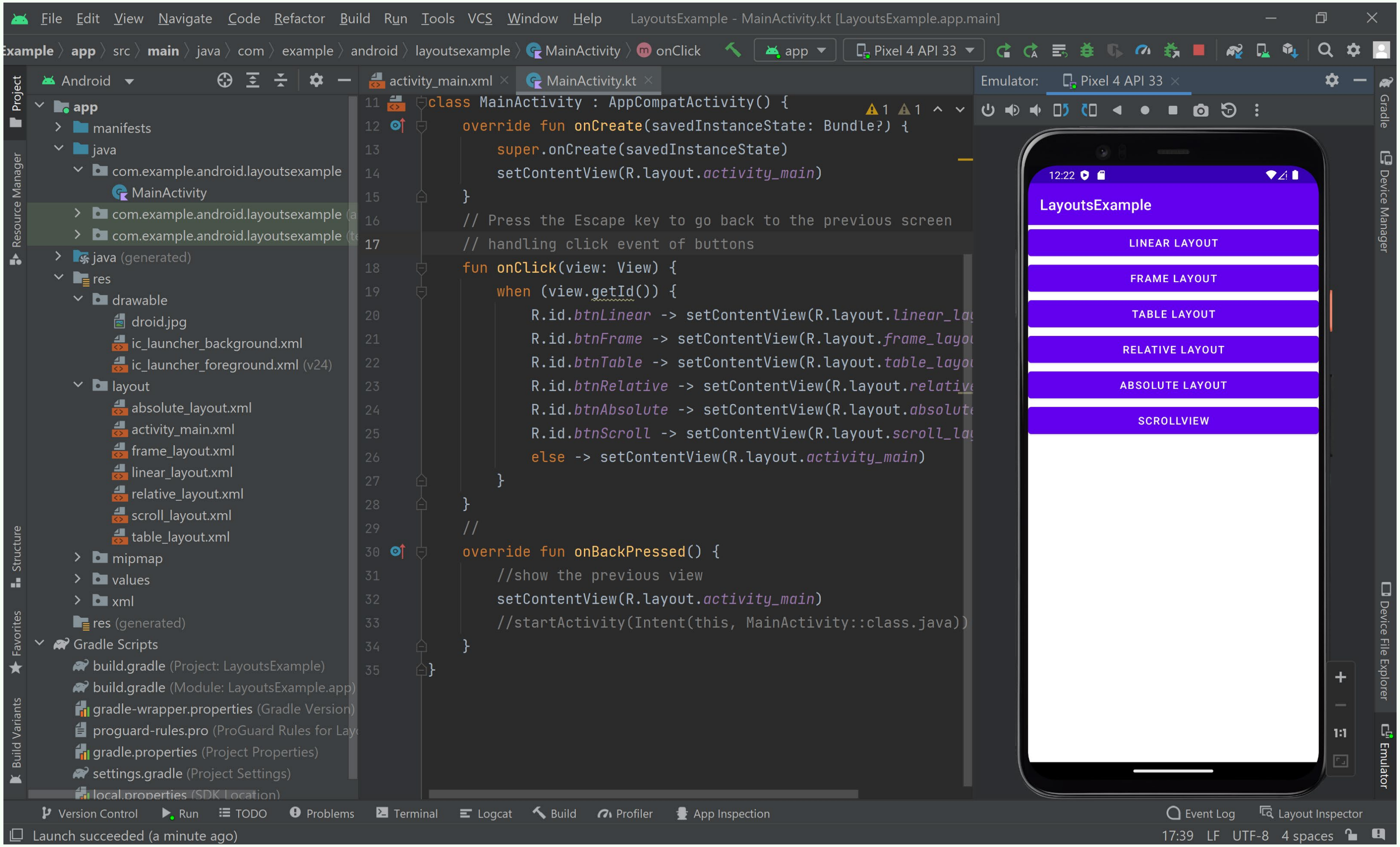


ScrollView

- A ScrollView is a special type of FrameLayout in that it enables users to **scroll through a list of views** that occupy more space than the physical display.
- The ScrollView can **contain only one child** view or ViewGroup, which normally is a LinearLayout.

```
<ScrollView
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
xmlns:android=”http://schemas.android.com/apk/res/android” >
        <LinearLayout
                android:layout_width=”fill_parent”
                android:layout_height=”wrap_content”
                android:orientation=”vertical” >
                ……..
        </LinearLayout>
</ScrollView>
```

Layout example

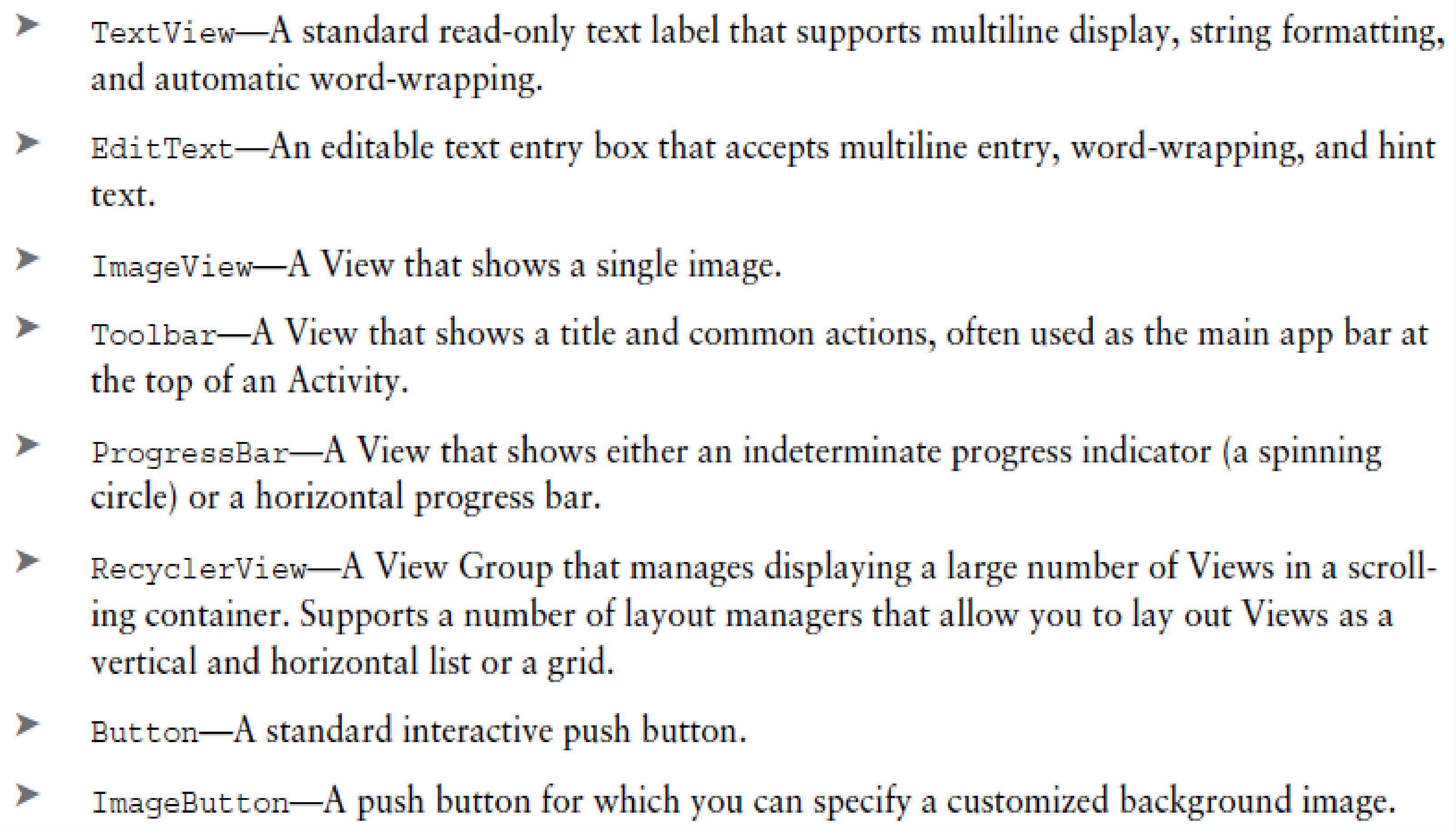

File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
LayoutsExample - MainActivity.kt [LayoutsExample.app.main]
xample 〉 app 〉 src 〉 main 〉 java 〉 com 〉 example 〉 android 〉 layoutsexample 〉 MainActivity 〉 onClick
app
Pixel 4 API 33
Android
app
manifests
java
com.example.android.layoutsexample
MainActivity
com.example.android.layoutsexample (a
com.example.android.layoutsexample (te
java (generated)
res
drawable
droid.jpg
ic_launcher_background.xml
ic_launcher_foreground.xml (v24)
layout
absolute_layout.xml
activity_main.xml
frame_layout.xml
linear_layout.xml
relative_layout.xml
scroll_layout.xml
table_layout.xml
mipmap
values
xml
res (generated)
Gradle Scripts
build.gradle (Project: LayoutsExample)
build.gradle (Module: LayoutsExample.app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for Lay
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
activity_main.xml
MainActivity.kt
Emulator:
Pixel 4 API 33
12:22
LayoutsExample
LINEAR LAYOUT
FRAME LAYOUT
TABLE LAYOUT
RELATIVE LAYOUT
ABSOLUTE LAYOUT
SCROLLVIEW
1:1
Project
Resource Manager
Structure
Favorites
Build Variants
Gradle
Device Manager
Device File Explorer
Emulator
Version Control
Run
TODO
Problems
Terminal
Logcat
Build
Profiler
App Inspection
Event Log
Layout Inspector
17:39 LF UTF-8 4 spaces

```kotlin
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
    // Press the Escape key to go back to the previous screen
    // handling click event of buttons
    fun onClick(view: View) {
        when (view.getId()) {
            R.id.btnLinear -> setContentView(R.layout.linear_la
            R.id.btnFrame -> setContentView(R.layout.frame_layo
            R.id.btnTable -> setContentView(R.layout.table_layo
            R.id.btnRelative -> setContentView(R.layout.relative
            R.id.btnAbsolute -> setContentView(R.layout.absolute
            R.id.btnScroll -> setContentView(R.layout.scroll_la
            else -> setContentView(R.layout.activity_main)
        }
    }
    //
    override fun onBackPressed() {
        //show the previous view
        setContentView(R.layout.activity_main)
        //startActivity(Intent(this, MainActivity::class.java))
    }
}
```

Launch succeeded (a minute ago)

Android Controls

- android.**widget** contains Android controls
  - All controls are typically **derived from the View** class



- TextView—A standard read-only text label that supports multiline display, string formatting, and automatic word-wrapping.
- EditText—An editable text entry box that accepts multiline entry, word-wrapping, and hint text.
- ImageView—A View that shows a single image.
- Toolbar—A View that shows a title and common actions, often used as the main app bar at the top of an Activity.
- ProgressBar—A View that shows either an indeterminate progress indicator (a spinning circle) or a horizontal progress bar.
- RecyclerView—A View Group that manages displaying a large number of Views in a scrolling container. Supports a number of layout managers that allow you to lay out Views as a vertical and horizontal list or a grid.
- Button—A standard interactive push button.
- ImageButton—A push button for which you can specify a customized background image.

Displaying Text to Users with TextView

- ❑ TextView control is used to draw text on the screen.
  - ➢ You primarily use it to **display fixed text** strings or labels
- ❑ It is derived from View and is within the **android.widget** package:
  - ➢ all the standard attributes such as width, height, padding, and visibility can be applied to the object
- ❑ You can set the **android:text** property of the TextView to be either a raw text string in the layout file or a reference to a string resource:

```
<TextView android:id=”@+id/tv_username”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
android:text=”User Name:” />
```

Displaying Text to Users with **TextView**

- ❑ Call findViewById method with the TextView identifier to make a reference to the TextView object:
- ❑ Capture the layout's TextView and set the string as its text:

```
val textView = findViewById<TextView>(R.id.
tv_username).apply { text = message }
```

- ❑ Retrieving the text is done using property access syntax:

```
val str: String = textView.text.toString()
```

Retrieving Data from Users

- Two frequently used controls to handle this type of job are **EditText** controls and **Spinner** controls
- **EditText** handles text input from a user.
  - The EditText class is **derived from TextView**
- This is how to define an EditText control in an XML layout file:

```
<EditText
android:id=”@+id/et_description”
android:layout_height=”wrap_content”
android:hint=”Enter description here”
android:lines=”4”
android:layout_width=”fill_parent” />
```

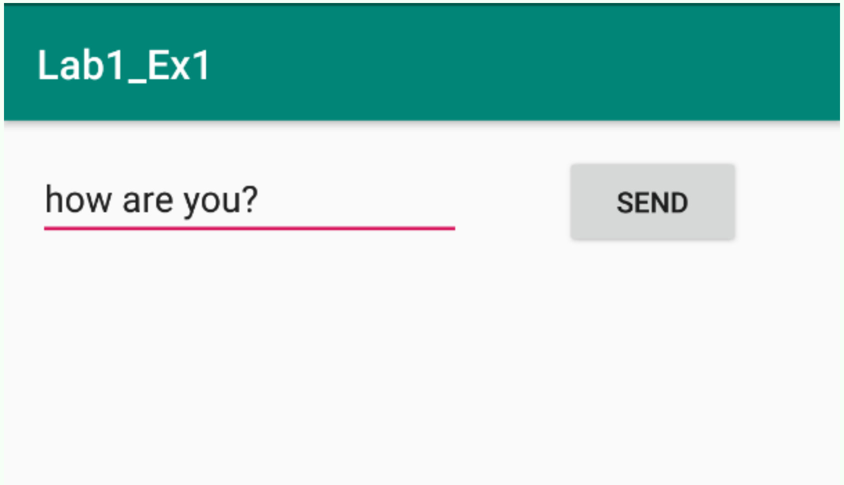
- **hint** attribute gives a hint to the user as to what should be typed in EditText control
- **lines** attribute, which defines how many lines tall the input box is

Retrieving Data from Users

- ❑ EditText object is essentially an editable TextView.
  - ➢ This means that you can read text from it in the same way as you did with TextView: by using the **text** property.

```
val editText = findViewById<EditText>(R.id.editText)
val message = editText.text.toString()
```

  - ➢ You can also set initial text to draw in the text entry area using the **text** property.



Lab1_Ex1

how are you?

Using Buttons

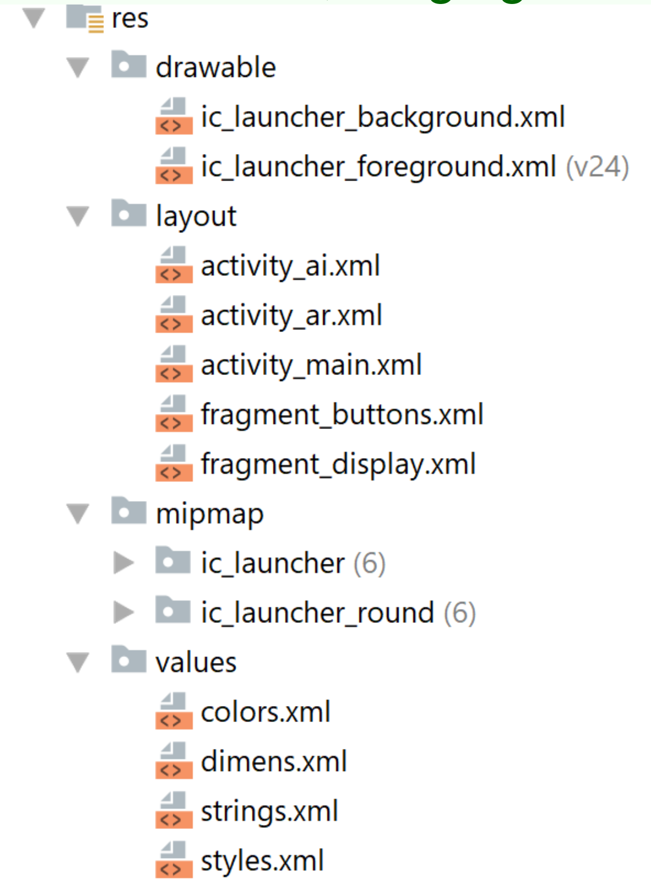
- The android.widget.**Button** class provides a basic button implementation.
- Within the XML layout resources, buttons are specified using the Button element.
- Use basic Button controls for buttons with text such as “Ok,”“Cancel,” or “Submit.”
- The following XML layout resource file shows a typical Button control definition:

```
<Button
android:id=”@+id/basic_button”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content”
Android:onClick=“sendInfo”
android:text=”Send” />
```

- Use **onClick** attribute to handle a click event and create an event handler method with a **View** argument to handle it.

Externalizing Resources

- It’s always good practice to **keep non-code resources, such as images and string constants, external** to your code.
- Android supports the externalization of resources, ranging from simple values such as:
  - **strings** and **colors**
  - **Images** and **animations**
  - **themes**
  - **UI layouts**
- Application resources are stored under the **res** folder in your project hierarchy.
- Each of the available resource types is stored in **subfolders**, grouped by resource type.



```
▼ res
  ▼ drawable
      ic_launcher_background.xml
      ic_launcher_foreground.xml (v24)
  ▼ layout
      activity_ai.xml
      activity_ar.xml
      activity_main.xml
      fragment_buttons.xml
      fragment_display.xml
  ▼ mipmap
    ▶ ic_launcher (6)
    ▶ ic_launcher_round (6)
  ▼ values
      colors.xml
      dimens.xml
      strings.xml
      styles.xml
```

Setting Simple Resource Values

- ❑ You can define resource types by **editing resource XML files manually** or by **using resource editors** available in Android Studio.
  - ➢ Here is a view of /res/values/**strings.xml** file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Using Android Resources</string>
    <string name="display">Demonstrating Font and Color!</string>
    <string name="app_name">Simple Resource Example</string>
    <color name="prettyTextColor">#fa31ff</color>
    <dimen name="textPointSize">14pt</dimen>
    <drawable name="red_rect">#ff0000</drawable>
</resources>
```

Setting Simple Resource Values

- ❑ It is a common practice to **store different types of resources in different files**.
- ❑ For example, store:
  - ➢ the strings in **/res/values/strings.xml**
  - ➢ the prettyTextColor color resource in **/res/values/colors.xml**
  - ➢ the textPointSize dimension resource in **/res/values/dimens.xml**
- ❑ This does not change the names of the resources, nor the code used earlier to access the resources programmatically

Using Resources in Code

❑ The generated R.java file:

```
package test.simpleresources;
public final class R {
    public static final class attr {
    }
    public static final class color {
        public static final int prettyTextColor=0x7f050000;
    }
    public static final class dimen {
        public static final int textPointSize=0x7f060000;
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
        public static final int redDrawable=0x7f020001;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Using Resources in Code

- ❑ Within your application, you access resources in code using the static R class.
- ❑ R is a generated class, created when your project is built, that lets you reference any resource you've included to offer design-time syntax checking.
- ❑ The R class **contains static subclasses** for each of the resources available, such as *R.string* and *R.drawable* subclasses.
- ❑ Each of the subclasses within R exposes its associated resources as variables, with the variable names matching the resource identifiers - for example, *R.string.app_name* or *R.mipmap.ic_launcher*.

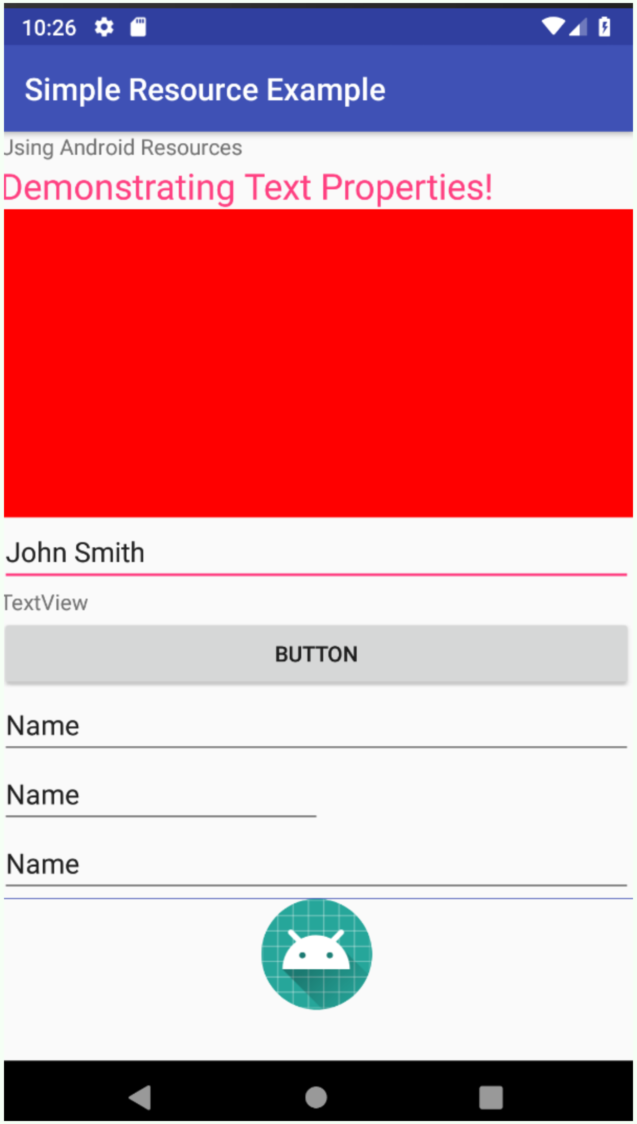
Using Resources Example

```
package com.example.android.usingresourcesexample

import android.graphics.drawable.BitmapDrawable
import android.graphics.drawable.ColorDrawable
import android.os.Bundle
import android.view.View
import android.widget.ImageView
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity


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

Using Resources Example

```
        val myString = resources.getString(R.string.display)
         val myColor = ContextCompat.getColor(this, R.color.niceTextColor)
         val myDimen = resources.getDimension(R.dimen.textPointSize)
         val myDraw = resources.getDrawable(R.drawable.red_rect) as ColorDrawable
         val imgView: ImageView = findViewById<View>(R.id.imageView1) as
        ImageView
         //get the flag image from resources
         val bitmapFlag = resources.getDrawable(R.drawable.flag) as BitmapDrawable
         //display the image on image view
         //imgView.setImageDrawable(myDraw);
         imgView.setImageDrawable(bitmapFlag)
         val flavors = resources.getStringArray(R.array.flavors)
         val tv = findViewById<View>(R.id.txtView) as TextView
         tv.textSize = myDimen
         tv.setTextColor(myColor)
         tv.text = myString
    }
}
```

Using Resources Example



10:26

Simple Resource Example

Jsing Android Resources

Demonstrating Text Properties!

John Smith

TextView

Name

Name

Name

Defining String Arrays

- String arrays, may be added to resource files by editing them manually:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name=”flavors”>
            <item>Vanilla</item>
            <item>Chocolate</item>
            <item>Strawberry</item>
    </string-array>
</resources>
```

- Access the array in your code:

```
val flavors = resources.getStringArray(R.array.flavors)
```

Working with Boolean Resources

- ❑ **Boolean** resources are defined in XML under the /res/values project directory and compiled into the application package at build time
- ❑ Are tagged with the <bool> tag and represent a name-value pair:

```
<resources>
    <bool name=”bOnePlusOneEqualsTwo”>true</bool>
    <bool name=”bAdvancedFeaturesEnabled”>false</bool>
</resources>
```

- ❑ The following code retrieves a boolean resource named bAdvancedFeaturesEnabled:

```
val bAdvancedMode =
    resources.getBoolean(R.bool.bAdvancedFeaturesEnabled)
```

Working with Integer Resources

- **Integer** values are tagged with the <**integer**> tag and represent a name/value pair.

```
<resources>
     <integer name=”numTimesToRepeat”>25</integer>
     <integer name=”startingAgeOfCharacter”>3</integer>
</resources>
```

- The following code accesses your application’s integer resource named numTimesToRepeat:

```
val repTimes =
    resources.getInteger(R.integer.numTimesToRepeat)
```

Working with Colors

- Android applications can store RGB color values, which can then be applied to other screen elements
- The following color formats are supported:
  - #RGB (example, #F00 is 12-bit color, red)
  - #ARGB (example, #8F00 is 12-bit color, red with alpha 50%)
  - #RRGGBB (example, #FF00FF is 24-bit color, magenta)
  - #AARRGGBB (example, #80FF00FF is 24-bit color, magenta with alpha 50%)
- Color values are tagged with the <color> tag and represent a name-value pair:

```
<resources>
        <color name=”background_color”>#006400</color>
        <color name=”text_color”>#FFE4C4</color>
</resources>
```

- The following code retrieves a color resource called text_color:

```
val myColor = ContextCompat.getColor(this, R.color.niceTextColor)
```

Working with Dimensions

- ❑ Many user interface layout controls such as text controls and buttons are drawn to specific dimensions.
  - ➢ These dimensions can be stored as resources.
- ❑ Dimension values always end with a **unit of measurement tag**:

Pixels	Actual screen pixels	**px**	20px
Inches	Physical measurement	**in**	1in
Millimeters	Physical measurement	**mm**	1mm
Points	Common font measurement unit	**pt**	14pt
Screen density Independent Pixels	Pixels relative to 160dpi screen (preferable dimension for screen compatibility). One **dp** is one pixel on a 160 dpi screen. dp = (width in pixels * 160) / screen density	**dp**	1dp
Scale independent Pixels	Best for scalable font display sp preserves a user's font settings	**sp**	14sp

- ❑ Dimension values are tagged with the <dimen> tag and represent a **name/value pair**.

Working with Dimensions

- Here’s an example of a simple dimension resource file /res/values/dimens.xml:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <dimen name=”FourteenPt”>14pt</dimen>
    <dimen name=”OneInch”>1in</dimen>
    <dimen name=”TenMillimeters”>10mm</dimen>
    <dimen name=”TenPixels”>10px</dimen>
</resources>
```

- Dimension resources are simply floating point values.
  - The following code retrieves a dimension resource called textPointSize:

```
val myDimension =
   resources.getDimension(R.dimen.textPointSize)
```

Working with Simple Drawables

- **Simple paintable drawable** resources are defined in XML under the /res/values project directory and compiled into the application package at build time.
- **Paintable drawable resources** use the <drawable> tag and represent a **name-value** pair.
  - Here’s an example of a simple drawable resource file /res/values/drawables.xml:

```
<resources>
     <drawable name=”red_rect”>#F00</drawable>
</resources>
```

- Drawable resources defined with <drawable> are simply **rectangles of a given color**:

```
val myDraw = ContextCompat.getDrawable(this,
    R.drawable.red_rect) as ColorDrawable?
```

Using Image Resources Programmatically

- ❑ Images resources are simply **another kind of Drawable** called a **BitmapDrawable**
- ❑ Use resource ID of the image to set as an attribute on a user interface control:

```
val flagImageView = findViewById<View>(R.id.ImageView01) as
   ImageView
flagImageView.setImageResource(R.drawable.flag)
```

- ❑ You can access the BitmapDrawable object directly:

```
val bitmapFlag = ContextCompat.getDrawable(this, R.drawable.flag) as
    BitmapDrawable?
```

Working with Lists and grids

- The **RecyclerView** (available from the androidx Library) offers **a scrollable View Group specifically designed to efficiently display, and scroll through, a large number of items**.
- The Recycler View can be used in both vertical and horizontal orientations, configured using the **android:orientation** attribute:

```
<androidx.recyclerview.widget.RecyclerView
xmlns:android:"http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/recycler_view"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
[... Layout Manager Attributes ...]
/>
```

RecyclerView

- ❑ The overall container for your user interface is a RecyclerView object that you add to your layout.
- ❑ The RecyclerView fills itself with views provided by a layout manager that you provide.
  - ➢ You can use one of our standard layout managers (such as LinearLayoutManager or GridLayoutManager), or implement your own.
- ❑ The views in the list are represented by **view holder objects**.
  - ➢ These objects are instances of a class you define by extending RecyclerView.**ViewHolder**.
  - ➢ Each view holder is in charge of displaying **a single item with a view**.

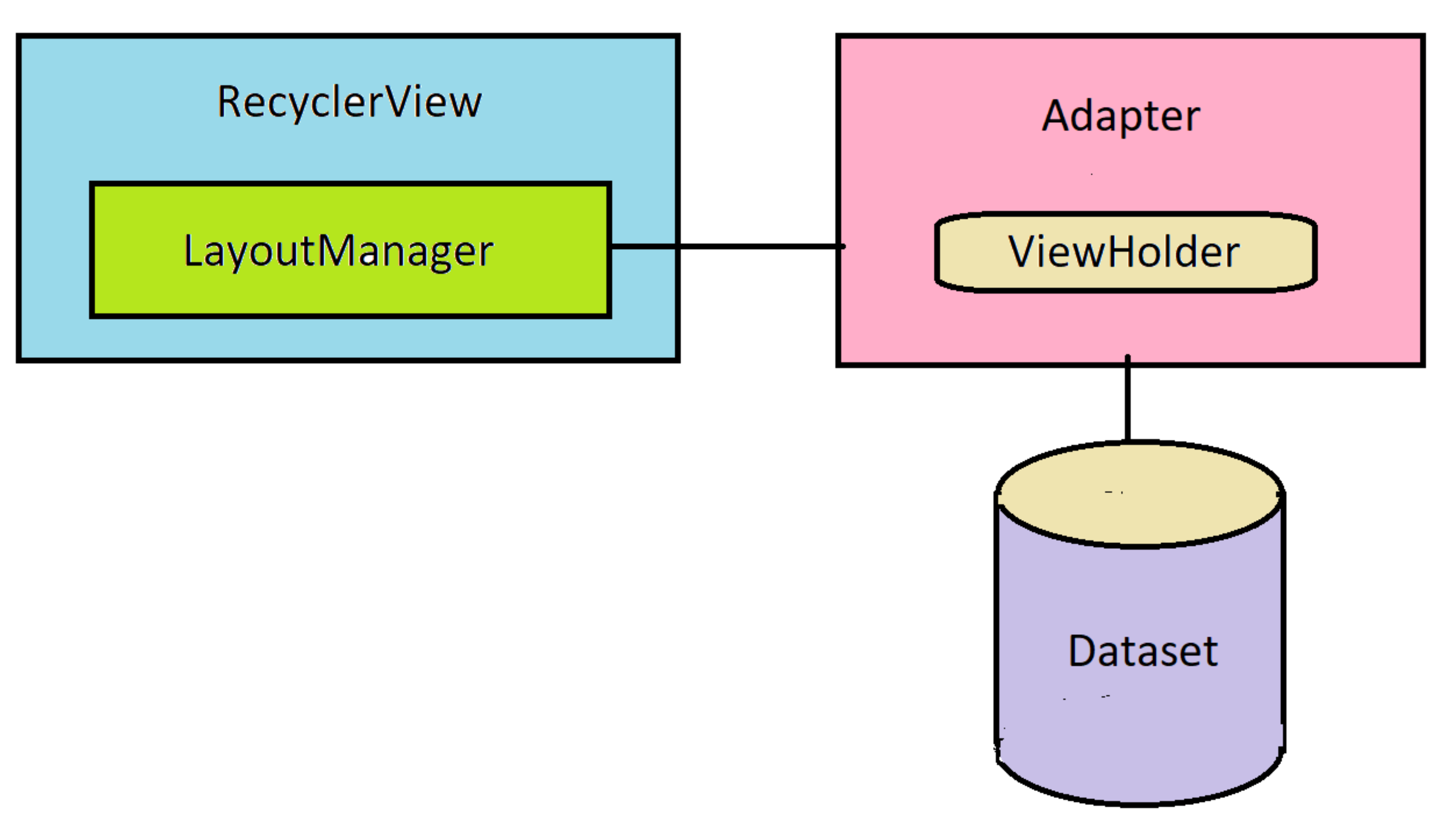
https://developer.android.com/guide/topics/ui/layout/recyclerview#java

RecyclerView

- ❑ RecyclerView **creates only as many view holders as are needed to display the on-screen portion** of the dynamic content, plus a few extra.
- ❑ As the user scrolls through the list, the RecyclerView takes the off-screen views and rebinds them to the data which is scrolling onto the screen.
- ❑ The **view holder objects are managed by an adapter**, which you create by extending RecyclerView.Adapter.
  - ➢ The **adapter creates view holders as needed**.
  - ➢ The **adapter also binds the view holders to their data** by assigning the view holder to a position, and calling the adapter's onBindViewHolder() method.

https://developer.android.com/guide/topics/ui/layout/recyclerview#java

RecyclerView



Recycler View and Layout Managers

- The RecyclerView.LayoutManager controls **how each item is displayed**
- A number of Layout Managers are available:
  - LinearLayoutManager - lays out items in a single vertical or horizontal list.
  - GridLayoutManager - similar to the Linear Layout Manager, but displays a grid.
    - When laid out vertically, each row can include multiple items, where **each is the same height**.
    - For horizontal orientation **each item in a given column must be the same width**.
  - StaggeredGridLayoutManager - similar to the Grid Layout Manager but creates a “staggered” grid, where **each grid cell can have a different height or width**, with cells staggered to eliminate gaps.

Using Recycler View and Layout Managers

- ❑ Open the build.gradle file for your app module and **add the support library to the dependencies section**:

```
dependencies {
 implementation 'androidx.recyclerview:recyclerview:1.1.0’
}
```

- ❑ **Add a RecyclerView widget to your layout**

```
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/my_recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    />
```

Using Recycler View and Layout Managers

- Create a custom layout for RecyclerView items:

```
<?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="wrap_content">
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/text_view"
        android:textSize="@dimen/textview_size" />
</LinearLayout>
```

- This is a simple implementation for a **data set that consists of an array of strings** displayed using TextView widgets.

Creating a Recycler View Adapter

- ❑ Create the **RecyclerView.Adapter** to feed data into the RecyclerView.
  - ➢ Override **onCreateViewHolder**, **onBindViewHolder**, **and getItemCount** methods.
- ❑ Create the **ViewHolder** to provide a reference to the views for each data item.
  - ➢ Create the views of the RycyclerView item.

```
internal class MyRecyclerViewAdapter(private var myDataset: Array<String>) :
RecyclerView.Adapter<MyRecyclerViewAdapter.MyViewHolder>() {
@NonNull
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
MyViewHolder {
        val v: View = LayoutInflater.from(parent.context)
           .inflate(R.layout.recyclerview_row_layout, parent, false)
        return MyViewHolder(v)
   }
```

Creating a Recycler View Adapter

```
override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        // - get element from your dataset at this position
        // - replace the contents of the view with that element
        holder.textView.text = myDataset[position]
    }
    override fun getItemCount(): Int {
        return myDataset.size
    }
// Create the ViewHolder to provide a reference to the views for each data item
    internal inner class MyViewHolder(v: View) : RecyclerView.ViewHolder(v),
View.OnClickListener {
        // each data item is just a string in this case
        val textView: TextView
        override fun onClick(v: View) {
            Toast.makeText(v.getContext(), textView.text, Toast.LENGTH_LONG).show()
        }
        init {
            textView = v.findViewById(R.id.textView)
            v.setOnClickListener(this)
        }
    }
}
```

Using a Recycler View Adapter

```
    // Replace the contents of a view (invoked by the layout manager)
    @Override
    override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
        // - get element from your dataset at this position
        // - replace the contents of the view with that element
        holder.textView.text = myDataset[position]
    }
    // Return the size of your dataset (invoked by the layout manager)
    override fun getItemCount(): Int {
        return myDataset.size
    }
}
```

Using a Recycler View Adapter

- In main activity:
  - Obtain a reference to the object

```
recyclerView = findViewById<View>(R.id.myRecyclerView) as
                                RecyclerView
```

- Connect it to a layout manager

```
layoutManager = LinearLayoutManager(this)
recyclerView!!.setLayoutManager(layoutManager)
```

- Read the array:

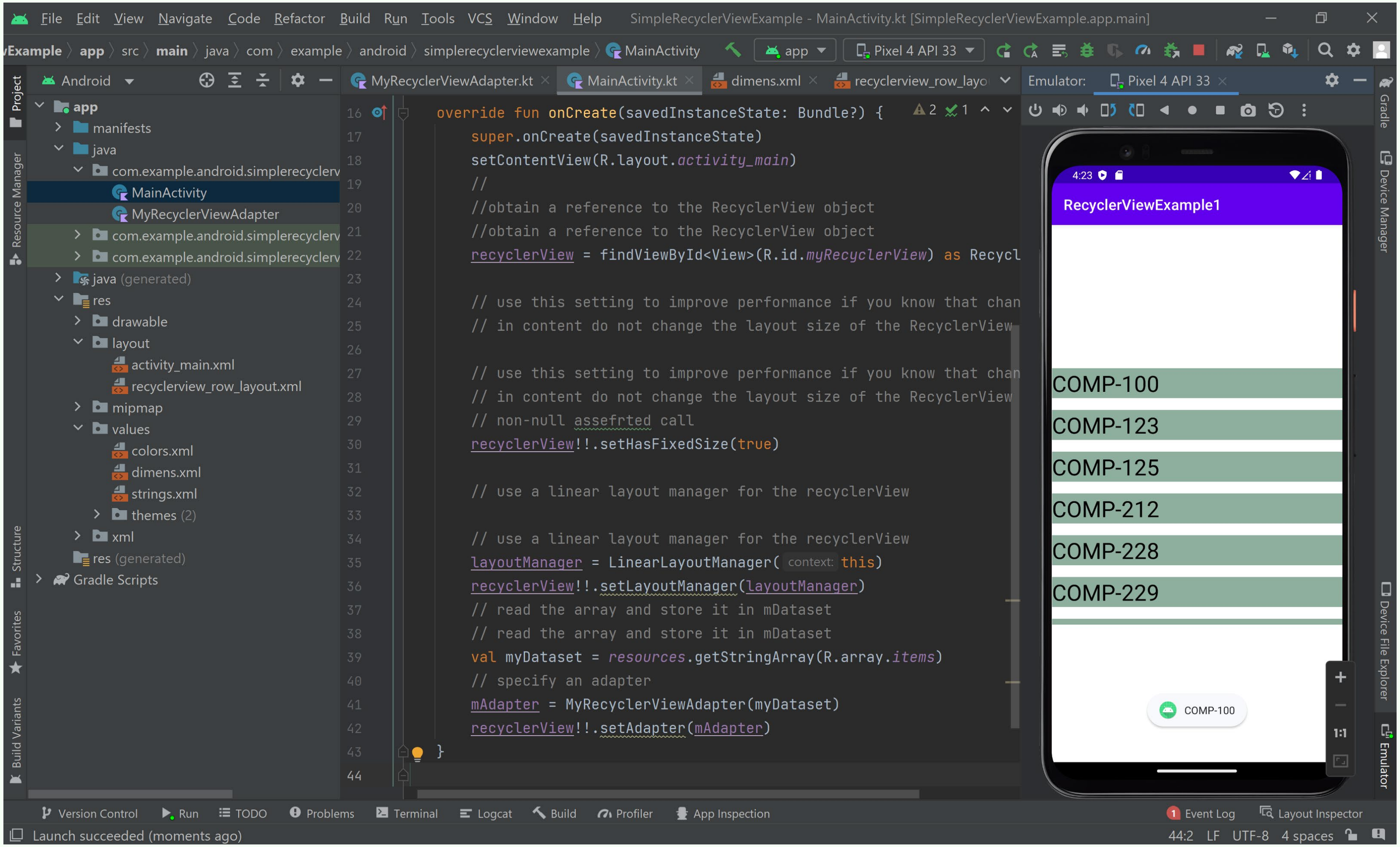
```
val myDataset = resources.getStringArray(R.array.items)
```

- Specify an adapter

```
mAdapter = MyRecyclerViewAdapter(myDataset)
recyclerView!!.setAdapter(mAdapter)
```

- See **SimpleRecyclerViewExample** application

SimpleRecyclerViewExample



```kotlin
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //
        //obtain a reference to the RecyclerView object
        //obtain a reference to the RecyclerView object
        recyclerView = findViewById<View>(R.id.myRecyclerView) as Recycl

        // use this setting to improve performance if you know that chan
        // in content do not change the layout size of the RecyclerView

        // use this setting to improve performance if you know that chan
        // in content do not change the layout size of the RecyclerView
        // non-null assefrted call
        recyclerView!!.setHasFixedSize(true)

        // use a linear layout manager for the recyclerView

        // use a linear layout manager for the recyclerView
        layoutManager = LinearLayoutManager( context: this)
        recyclerView!!.setLayoutManager(layoutManager)
        // read the array and store it in mDataset
        // read the array and store it in mDataset
        val myDataset = resources.getStringArray(R.array.items)
        // specify an adapter
        mAdapter = MyRecyclerViewAdapter(myDataset)
        recyclerView!!.setAdapter(mAdapter)
    }
```

Launch succeeded (moments ago)

Declarative Data Binding

- The **Data Binding library** makes it possible to write **declarative layouts** that minimize the glue code needed to bind View elements to underlying data sources by **generating that code for you at compile time**.
- To implement data binding in your app, follow these steps:
  - Enabling Data Binding in your **application module's** build.gradle file:

```
android {
[... Existing Android Node ...]
    buildFeatures {
        viewBinding true
        dataBinding true
     }
dependencies {
[... Existing dependencies element ...]
    implementation 'androidx.databinding:databinding-runtime:7.2.1'
```

Declarative Data Binding

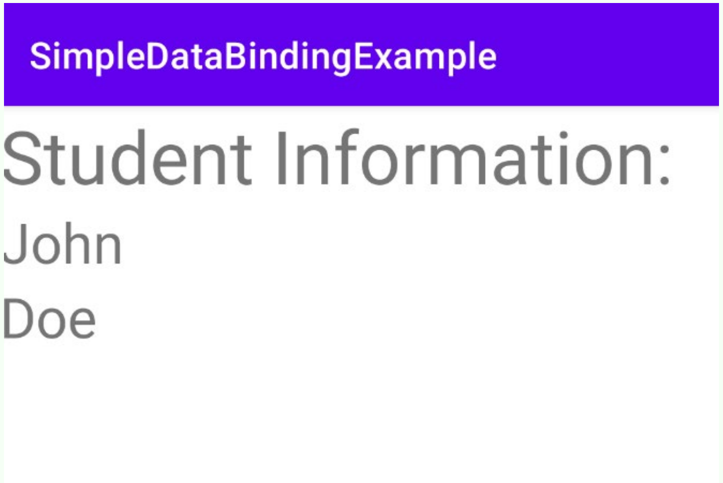
- Create a class to describe the data:

```
package com.example.android.simpledatabindingexample



internal class Student(val firstName: String, val lastName: String)
```

- Apply Data Binding to any layout by wrapping the elements of a layout file in a new <layout> element
  - Modify the layout as in the following:

Declarative Data Binding

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
   <data>
      <variable name="student"
type="com.example.android.simpledatabindingexample.Student"/>
   </data>
   <LinearLayout
      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="@{student.firstName}"
        />
          ……….
</LinearLayout>
</layout>
```



SimpleDataBindingExample

Student Information:

John

Doe

Declarative Data Binding

- ❑ A **data binding class** is generated for each layout file.
- ❑ The name of the class is based on the **name of layout file**: for activity_main.xml will be ActivityMainBinding, etc.
- ❑ The best practice method to create your bindings is to do it while inflating the layout:

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding: ActivityMainBinding =
            DataBindingUtil.setContentView(this,
R.layout.activity_main)
        val student = Student("John", "Doe")
        binding.student = student
    }
```

- ❑ **See SimpleDataBinding Example**

References

- ❑ Textbook
- ❑ https://material.io/guidelines/layout/units-measurements.html
- ❑ https://developer.android.com/guide/topics/ui/overview.html
- ❑ https://developer.android.com/guide/topics/ui/declaring-layout.html
- ❑ https://developer.android.com/guide/topics/ui/layout/recyclerview
- ❑ Lauren Darcey, Shane Conder: Introduction to Android Application Development: Android Essentials (5th Edition)