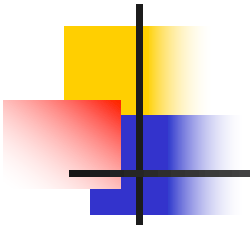
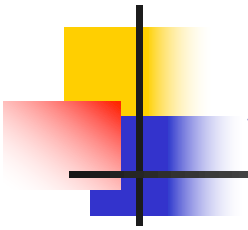
Second Application

Views, Layout Editor and Android Basics



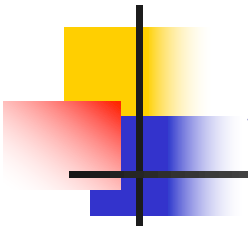
Overview

- Learn Android basics
- Create a new application with real (but simple) functionality
- Use the Layout Editor
- Implement listeners
- Use `Logcat` for debugging and error logging
- Install and debug your application on real hardware



Android Activities

- Before we go any further, we need to define a few terms of the Android apps and SDK.
- Typically, an Android app includes one or more Activities.
- Think of an Activity as a single screen/window for user to interact with an application.
- It is similar to a window in a JavaFX GUI program.



Android Activities

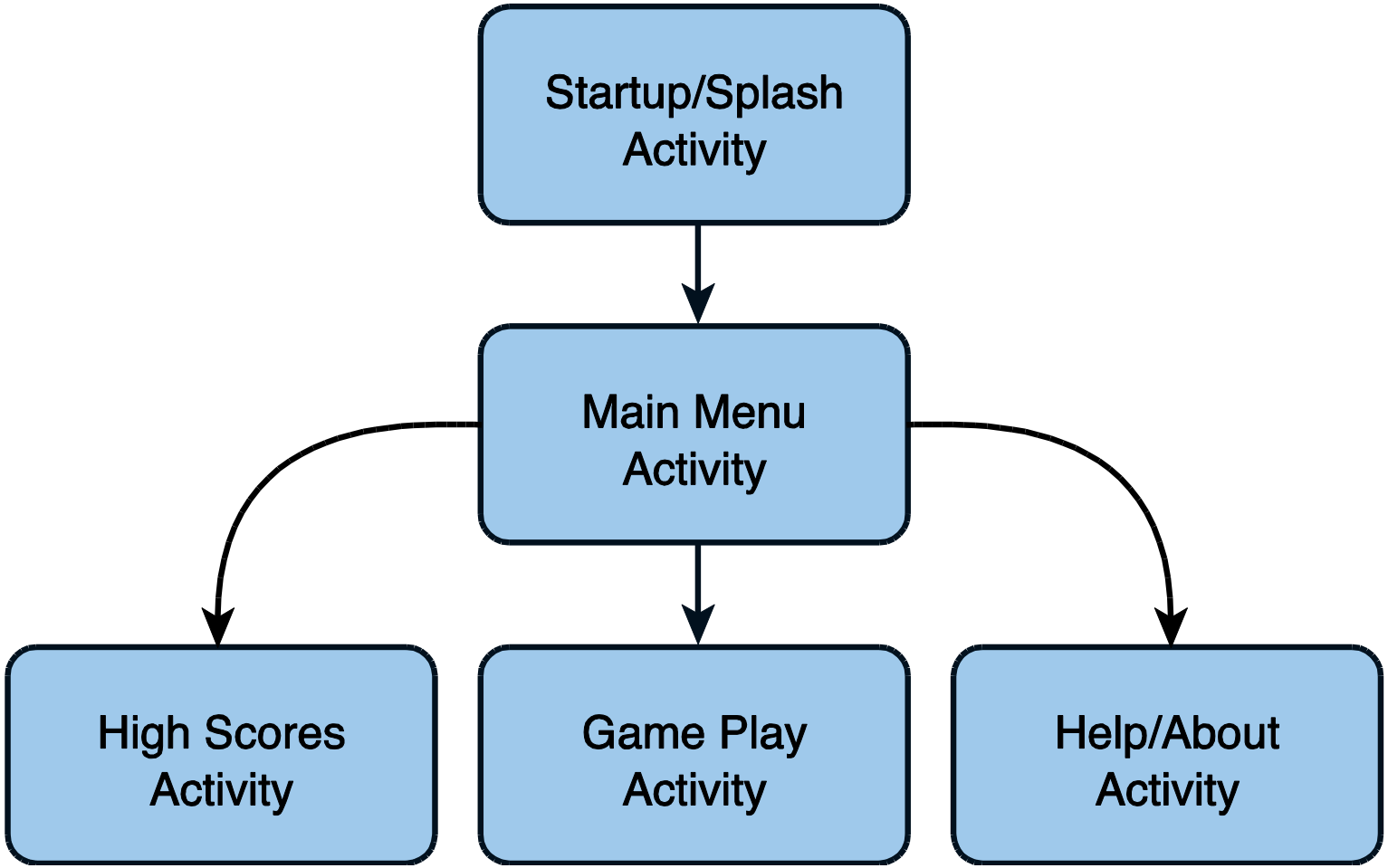
- Typically, an activity includes UI components or widgets and the users interacts with the app using these components and widgets.
- An Android app usually includes multiple activities.

Android Activities

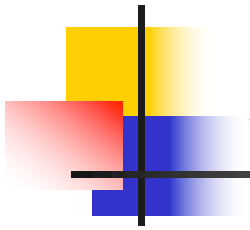
- As an example, a simple game application might have the following five activities:
  - Startup (or splash) screen
  - Main menu screen
  - Game play screen
  - High scores screen
  - Help/About screen

Performing Tasks with Activities

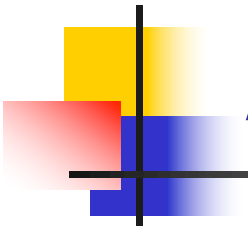
- The user navigates from one Activity to the next



Android Application Startup

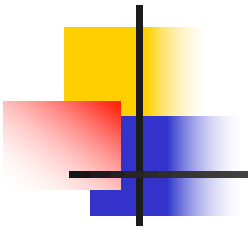


- The user elects to start an application of his or her choice.
- Usually, by tapping on the app's icon on the apps screen.
- Android recognizes which app must be started up, finds its main (launcher) activity, and initializes its lifecycle by calling its `onCreate()` method.
- Once the activity is fully functioning, it is placed on top of the Android's activity stack.



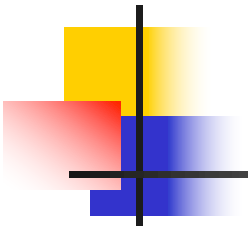
Android Application Startup

- When an Activity first starts, its `onCreate()` method is called. This method is an *event handler*, also referred to as a *callback*.
- `onCreate()` has a single parameter, a Bundle (it is `null` for a newly started Activity).
- The app performs a setup (layout creation and data initialization), for example, using `setContentView()`, inside `onCreate()`.



Android Views

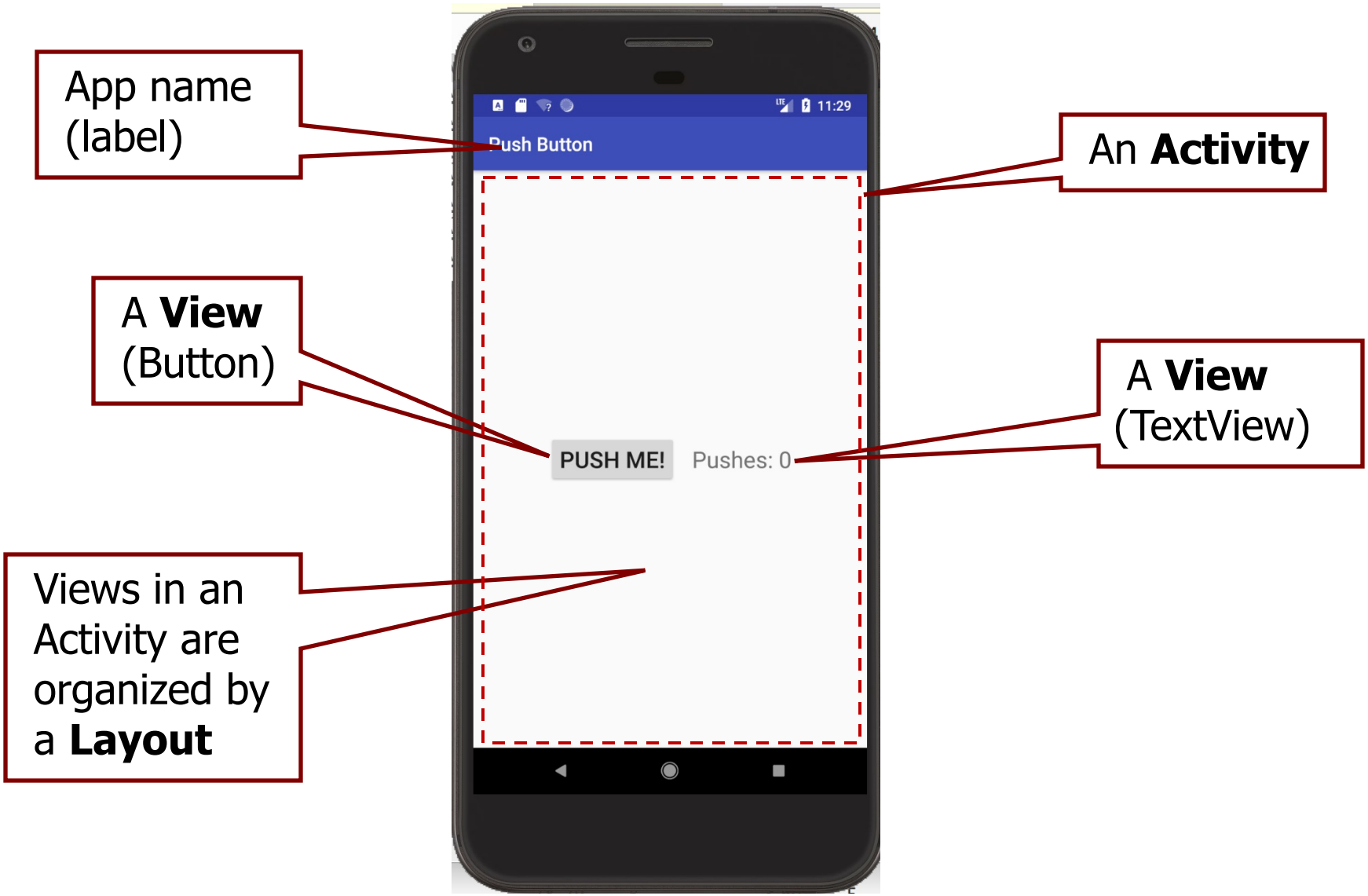
- An Activity (screen) includes *UI components* or *widgets* allowing users to interact with the app.
- The Android SDK has a Java package named `android.view`, which contains several interfaces and classes related to drawing components and widgets on the screen.
- However, when we refer to a View object, we actually refer to only one of the classes within this package.

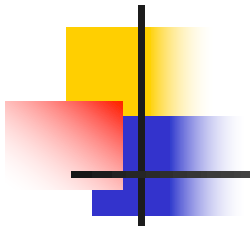


Android Views

- The `android.view.View` class is the base class for nearly all the user interface controls and layouts within the Android SDK.
- The View class is the basic *user interface building block* within Android. It represents a rectangular portion of the screen, usually with some contents inside.

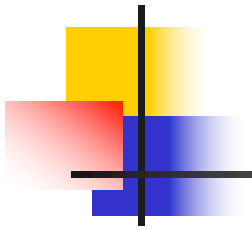
An Activity and Its Views





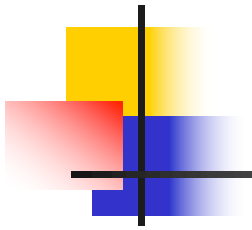
Android Controls

- The Android SDK also contains a Java package named `android.widget`.
- An Android control typically refers to a class within this package.
- Some of the most common objects include:
  - `TextView`
  - `EditText`
  - `Button`
  - `ImageView`
- As mentioned previously, all controls are typically derived from the `View` class.



Android Controls

- Layout resources are composed of different user interface controls.
  - Some are static, and you don't need to work with them programmatically.
  - Others you'll want to be able to access and modify in your Java code.
- Layout is responsible for the placement of its interface controls.
- It is typically stored in XML file and Android inflates (creates) a layout based on its (XML) specification.



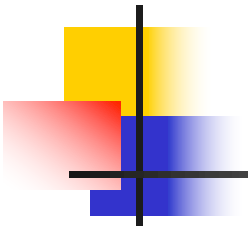
Android Controls

- Control objects in a layout are created by Android and thus references to them are initially unknown.
- Each control to be accessed programmatically usually has a unique identifier, specified using the `android:id` attribute.
- This identifier is used to access the control using the `findViewById()` method in the Java code.

Android Controls

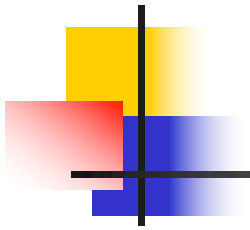
- In the past, it was necessary to cast the View returned to the appropriate control type.
- Recently, this casting has not been necessary due to type predictions.
- The following code illustrates how to access a `TextView` control using its unique identifier:

```
TextView tv = (TextView) findViewById(
                                    R.id.TextView1 );

TextView tv = findViewById( R.id.TextView1 );
```



Android Layout

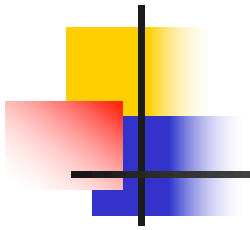
- A special type of control found within the `android.widget` package is called a layout.
- A layout control is still a `View` object.
- It is a parent container for organizing other controls (its children).
- Layouts determine how and where on the child controls are drawn (placed).



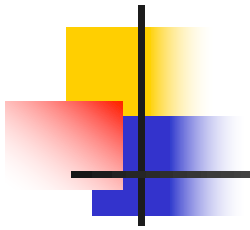
Android Layout

- Each type of layout control draws its children using particular rules.
- For instance, the `LinearLayout` control draws its child controls in a single horizontal row or a single vertical column.
- Similarly, a `TableLayout` control displays each child control in tabular format (in cells within specific rows and columns).

Android Layout



- Another type of a (very popular) layout is `ConstraintLayout`.
- It is a new type of layout, where positioning of a child control is defined (or as it is described – constrained) in terms of constraints with respect to the parent or other controls within the layout.



Android Layout

- Examples of constraints include:
  - `layout_constraintTop_toTopOf`
  - `layout_constraintBottom_toTopOf`

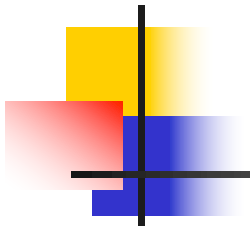
  so that you can specify constraints of one view *relative* to another view or the parent.

  We will talk about the `ConstraintLayout` in a separate lecture, later.

Building the Second Application

- Now, we will build an app similar to one of the JavaFX applications (Push Counter) we discussed before. Follow the steps below.
- To start, create a new AS project and select "Phone and Tablet" and an Empty Views Activity.
- Call the new project `Push Counter` and make sure the package name is `edu.uga.cs.pushcounter`. We will use package names of the form

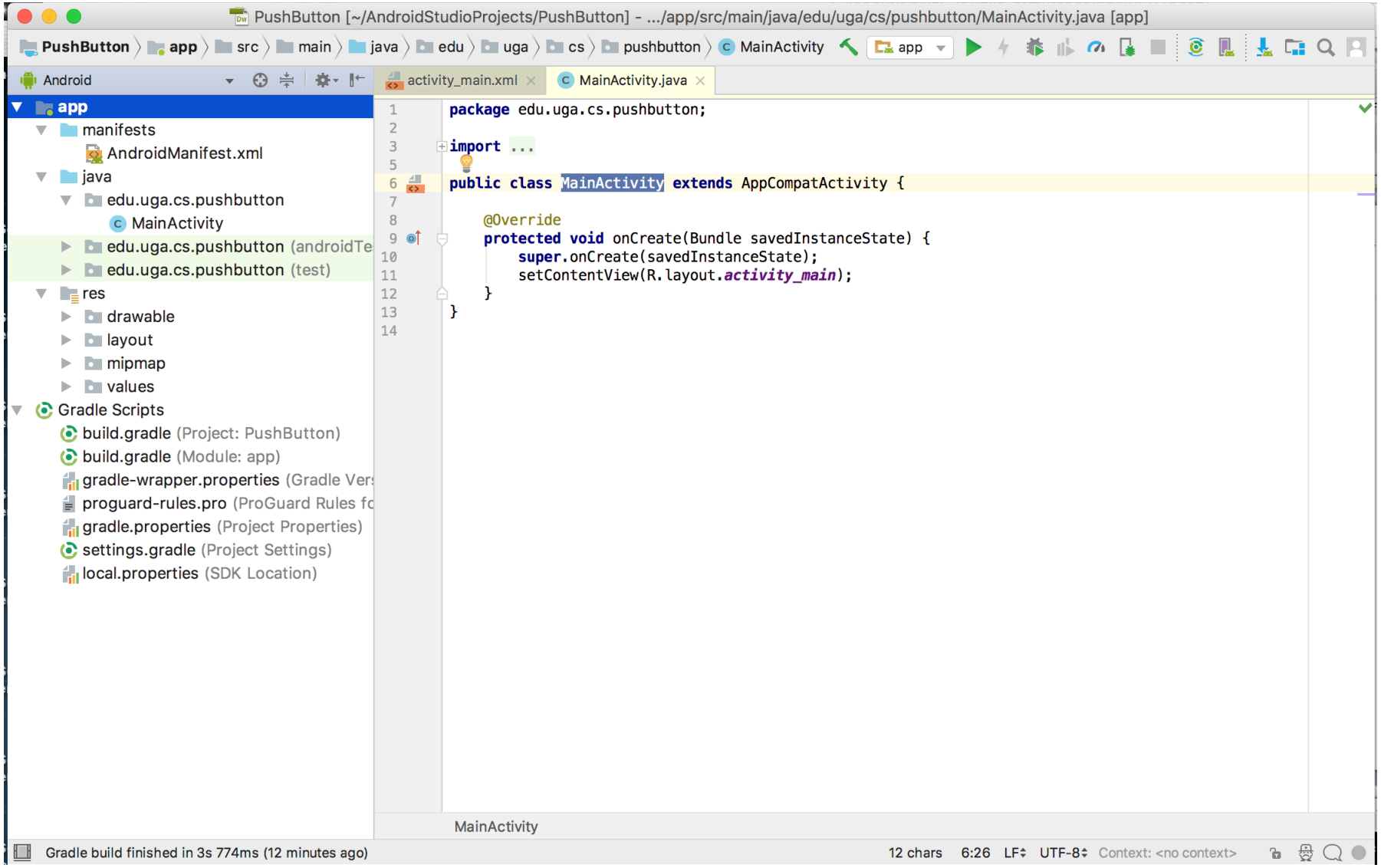
  `edu.uga.cs.`*`name`*.
- Select Java, API 24: Android 7.0 (Nougat) as the minimum SDK, Groovy DSL, and then click Finish.



Building the Second Application

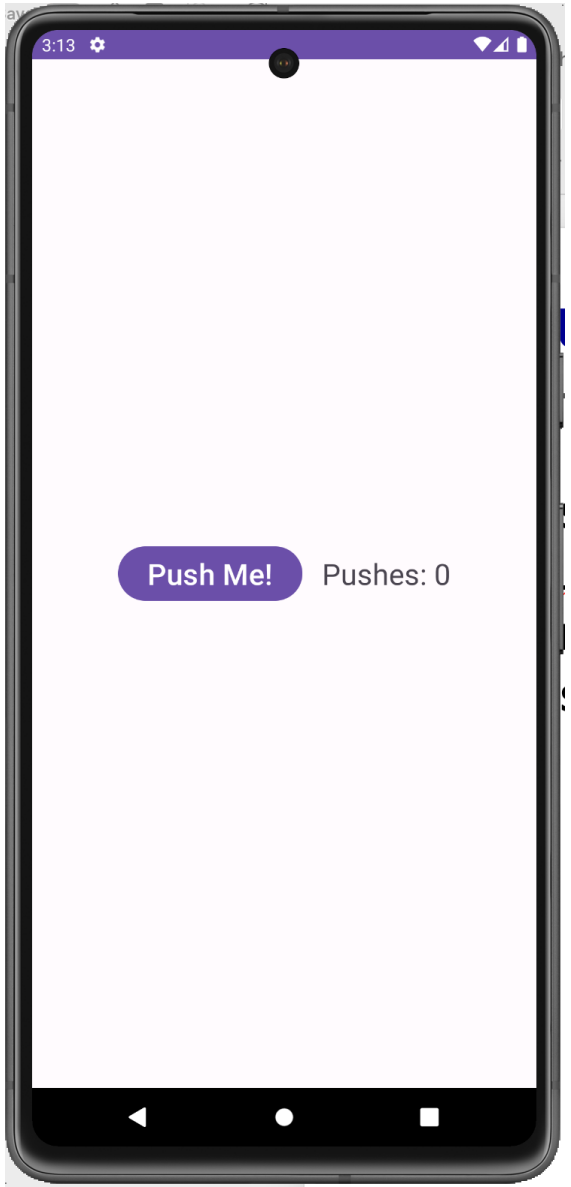
- Once the project has been created and the IDE starts, switch to the Android view (left panel), if not already set to Android.
- In the Android view of the project, expand the app folder then its subfolders:
  - `manifests`
  - `java` and its subdirectories, especially for the `edu.uga.cs.pushcounter` package
  - `res` and its subdirectories, especially one called `layout`

Android View of the Project



Building the Second Application

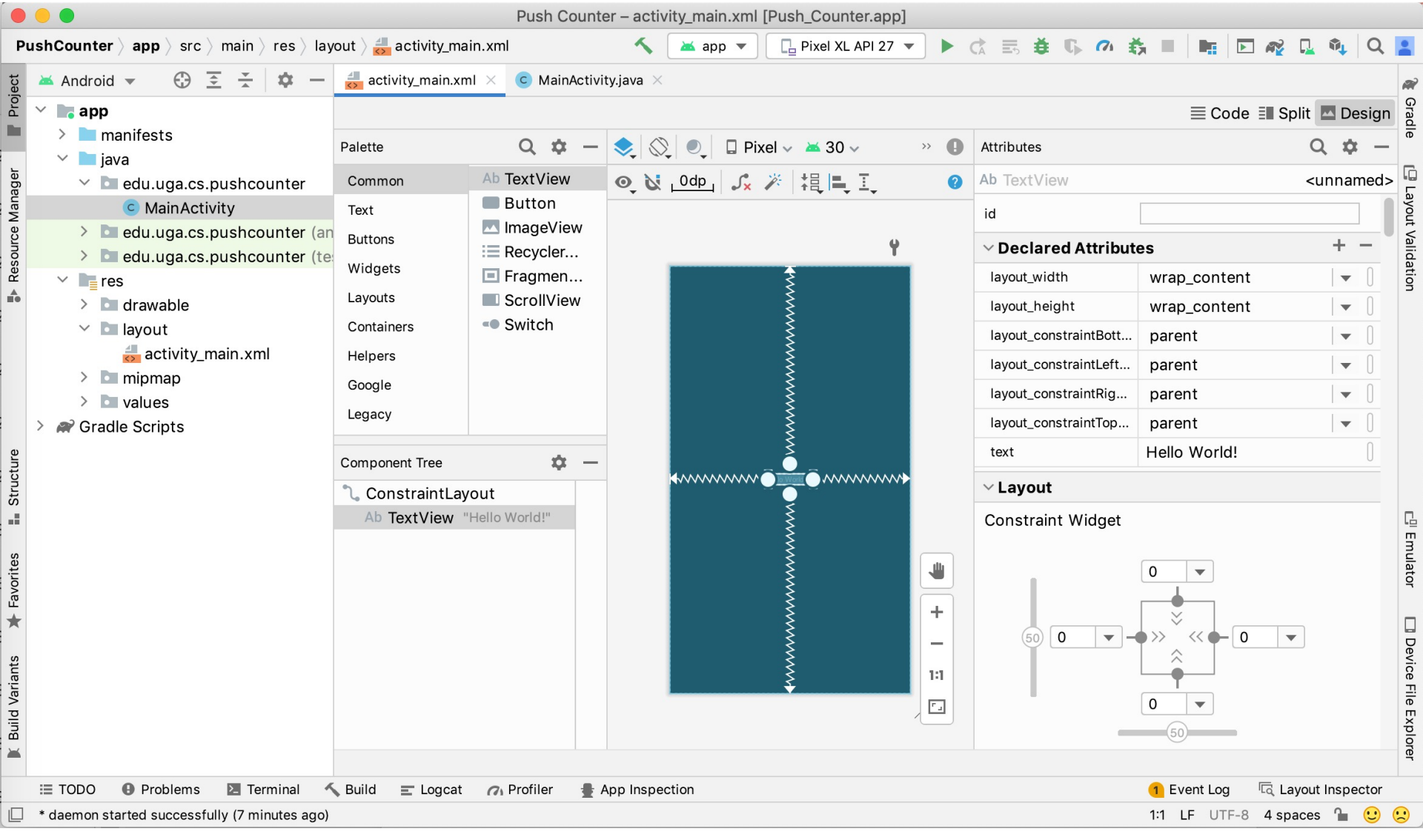
- First, we will create a LinearLayout for our app to look like this one.

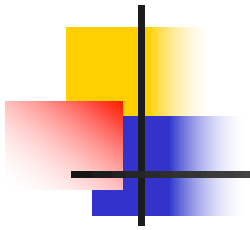


Building the Second Application

- Open the `activity_main.xml` tab
  - It should already be open, but if not, find it in the `app/res/layout` subdirectory in the Android view of the project (in the left pane) and double-click its name.
- Select the `Design` view (in the top right of the AS window); `Code` shows the XML code of the layout.
- Also, the `Design` view may be actually `Design` + `Blueprint`, if you see two rectangles (screen mockups). Using the  button in the menu bar, (above on the left of the design pane), select Blueprint. You may enlarge the blueprint view a bit.

Opening the Layout Editor

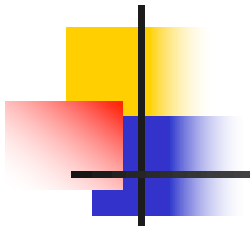




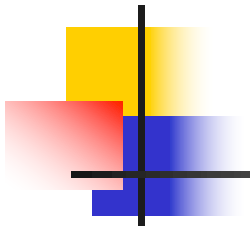
Building the Second Application

- Look for the `Component Tree` panel and select the `TextView "Hello World!"` by clicking on it. Then right-click and select `Delete` to delete it.
- The blueprint should be empty now (with no views); the Component Tree should have no children.
- In the `Component Tree` panel, select the `ConstraitLayout` right-click on it and select `Convert view…`
- In a new pop-up window, select `LinearLayout` and then click `Apply`. `Component Tree` should now show a `LinearLayout (horizontal)`.

Building the Second Application



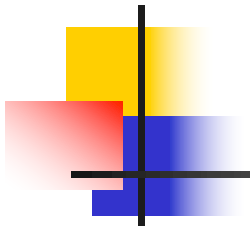
- In the `Component Tree` panel, select the `LinearLayout (horizontal)` right-click on it, and select `LinearLayout->Convert orientation to vertical`.
- `Component Tree` should now have a `LinearLayout (vertical)`.
- We just converted the `LinearLayout`'s orientation from horizontal to vertical.
- The layout should still be empty at this point, as it has no views inside it.



Building the Second Application

- Look for the `Palette` panel. Click on `Layouts` in that panel and then select `LinearLayout (horizontal)` and drag it to the `Component Tree` panel, right under the `LinearLayout` already in it. A thin blue line and a green circle with a plus should appear when you can drop (release the mouse button) the new LinearLayout.
- This will place a new *horizontal* LinearLayout inside of the already existing *vertical* LinearLayout.
- We just nested linear layouts, as while nesting an HBox inside a VBox in JavaFX. You can clearly see it in the `Component Tree` panel.

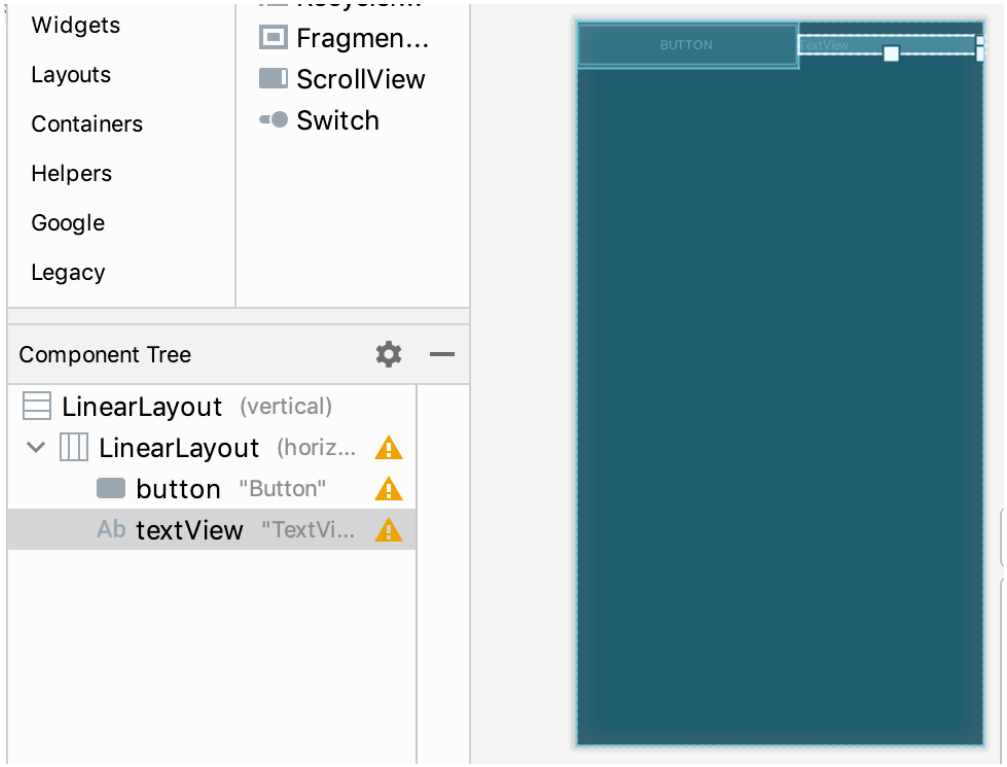
Building the Second Application

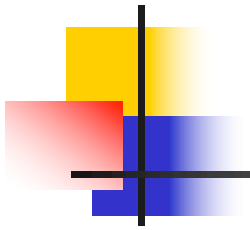


- Now, select the `Commons` tab in the `Palette` panel.
- Click and hold the the `Button` (second selection) and drag-and-drop it onto the layout editor (blueprint) representation of the activity.
- Again, in the `Palette`, click and hold the the `TextView` (first selection) and drag-and-drop it onto the layout editor (blueprint) representation of the activity.

Building the Second Application

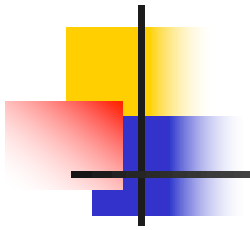
- The `Button` and `TextView` should be children within the nested horizontal `LinearLayout` and the `Component Tree` should reflect this hierarchy. The yellow triangles are warnings; don't worry now.





Building the Second Application

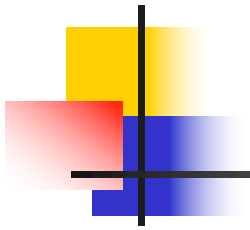
- We want to create a layout that is similar to the one in the JavaFX's Push Counter example.
- As you can see in the Blueprint view on the right, the `Button` and `TextView` are at the very top and are too wide.
- We will need to adjust their sizes and center them.



Building the Second Application

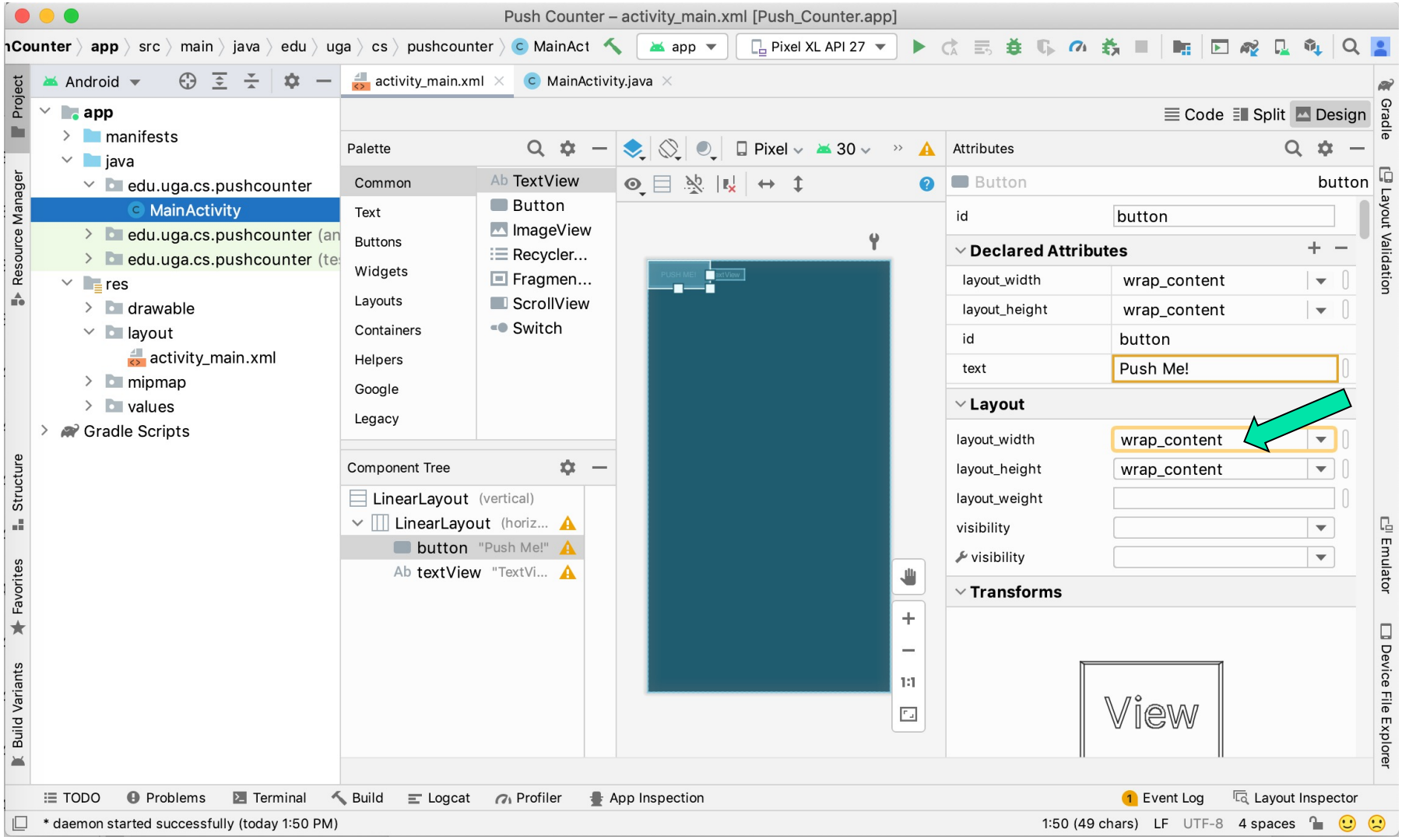
- In the `Component Tree`, select the `Button`. Its attributes should appear in the `Attributes` panel on the right.
- Change the `text` attribute from `Button` to `Push Me!`
- Look for the `layout_weight` attribute and delete its default value of `1` (select it and delete it). Its value should be left unspecified.
- Make sure that the `layout_width` and `layout_height` attributes are both set to `wrap_content`.

Building the Second Application

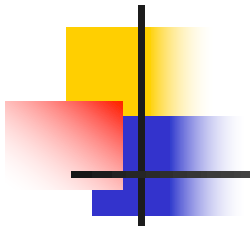


- In the `Component Tree`, select the `TextView`. Again, its attributes should appear in the `Attributes` panel on the right.
- Change the `text` attribute from `TextView` to `Pushes: 0`
- Again, look for the `layout_weight` attribute and delete its default value of `1` (select it and delete it). Its value should be left unspecified.
- Again, make sure that the `layout_width` and `layout_height` attributes are both set to `wrap_content`.

Creating a Layout in Layout Editor

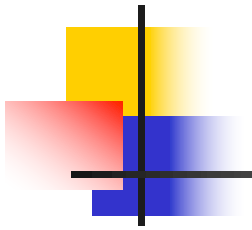


Building the Second Application



- Now, we will focus on sizing and centering of views.
- In the `Component Tree`, select the nested `LinerLayout (horizontal)`. Once again, its attributes should appear in the `Attributes` panel on the right.
- Set both the `layout_width` and `layout_height` attributes to `wrap_content`. Use the drop-down list to select attribute values for both.
- `wrap_content` sets the horizontal and/or vertical height of a view to be just large enough to hold the view's contents.

Building the Second Application

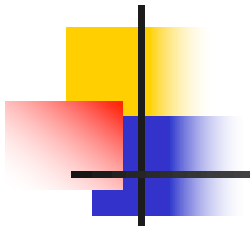


- Now, in the `Component Tree`, select the topmost `LinerLayout (vertical)`.
- In the `Attributes` panel look for the layout's `> gravity` attribute.
- The `gravity` attribute controls the positioning of the children views inside of the layout.
- Expand it (click on the `>`) and you will see a collection of various positioning attributes.
- Look for the `center` checkbox and toggle it. `False` should change to `true`.

Building the Second Application

- You may switch to the Code view for a moment, just to see the created XML specification of your layout:

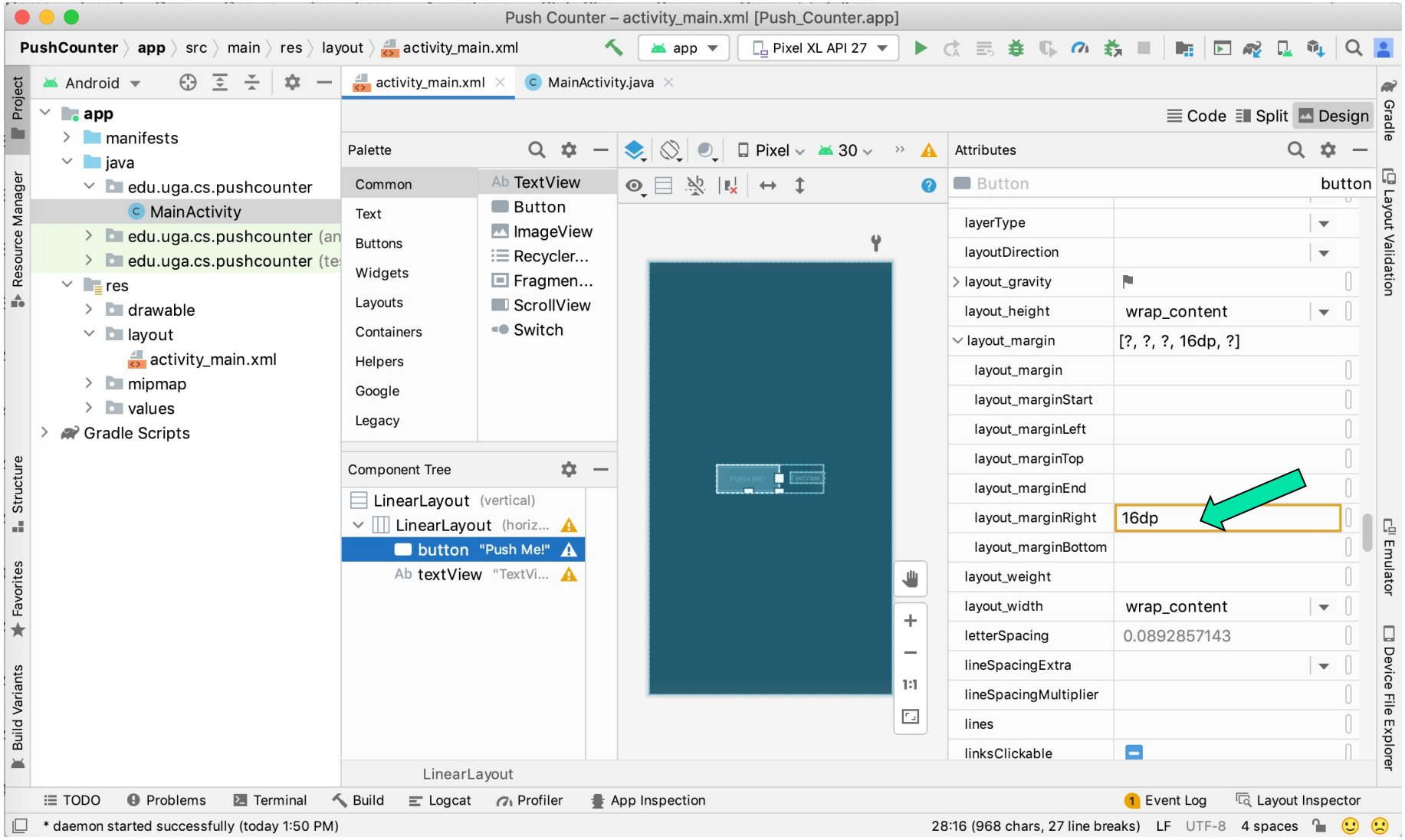
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"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

   <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

      <Button
           android:id="@+id/button"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="Push Me!" />

      <TextView
           android:id="@+id/textView"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="TextView" />
   </LinearLayout>
</LinearLayout>
```



Building the Second Application

- Now, we will increase the gap between the Button and TextView.
- Select the `Button` in the `Component Tree`.
- In the `Attributes` panel, scroll down to locate the `layout_margin` attribute and expand it.
- Look for the `layout_marginRight` and set its value to 16dp (dp stands for *device-independent pixels*). We will talk about it later.
- This will add a suitable gap between the button and its neighbor on the right (the `TextView`).

Building the Second Application

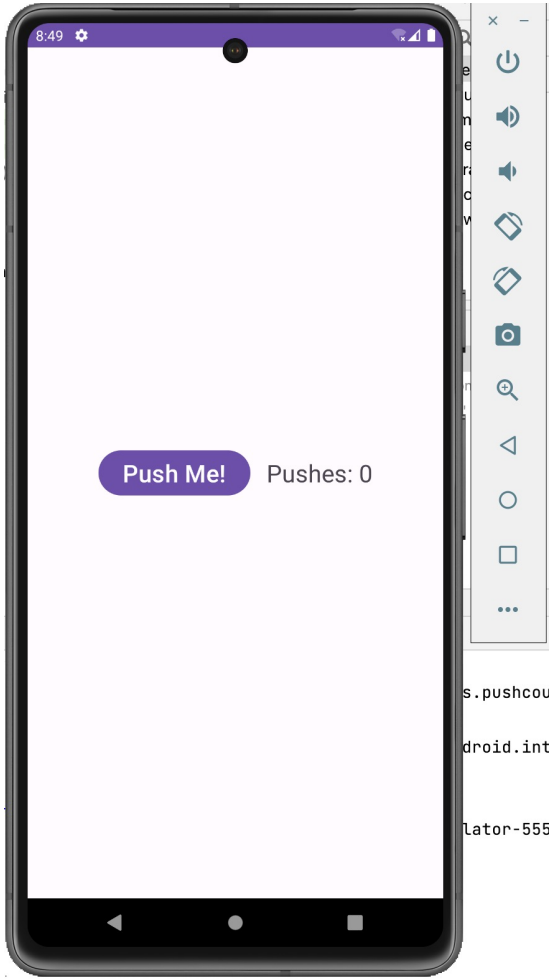


Building the Second Application

- While still having the button selected, scroll down the list of its attributes to `textAllCaps` (checkbox).
- Toggle the checkbox `textAllCaps` to `false` (it should be deselected). By default, the capitalization of letters in a button's text is set to ALL CAPS.
- Now, look for `textSize` and set it to `24sp` (sp unit is the scale-independent, or scalable pixels, used for font sizes). We will talk about it later.
- Finally, select the TextView in the Component Tree, scroll down its attributes to the `textSize` attribute and set it to `24sp`, as well.
- Switch from Blueprint to Design.

Push Button App in Pixel 7 AVD

- Run the app in the emulator. You should see a similar screen.



Complete activity_main.xml

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

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <Button
            android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginRight="16dp"
            android:text="Push Me!"
            android:textAllCaps="false"
            android:textSize="24sp" />

        <TextView
            android:id="@+id/textView"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Pushes: 0"
            android:textSize="24sp" />
    </LinearLayout>
</LinearLayout>
```

Implementing the main activity

- Now, it is time to add some functionality to the designed activity
- Switch to the `MainActivity.java` tab
  - It should already be ready, but if not, find it in the `app/java/edu.uga.cs.pushcounter` package in the Android view of the project (in the left pane) and open it by double clicking.
- Fragments of the code for the `MainActivity` class are available on eLC in the In-class Projects folder. You may copy/paste from there into your AS project.

Implementing the main activity

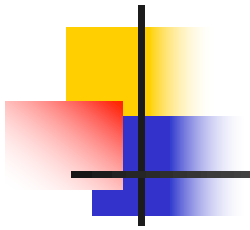
- Add the following instance variables, right after the class header to declare instance variables for the push counter and the view objects:

```
private int counter = 0;
private TextView countertView;
private Button pushButton;
```

- Add the following lines right after the last statement inside the `onCreate` method, i.e., after the call to `setContentView`, but NOT before it.

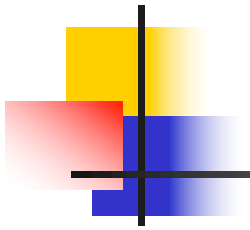
```
countertView = findViewById( R.id.textView );

pushButton = findViewById( R.id.button );
```

Implementing the main activity

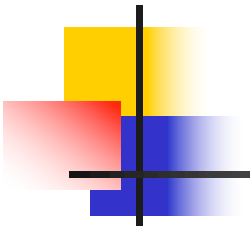


- Android reads an XML layout file during the build process and compiles it into a View resource.
- At runtime, the layout's view resource is used to inflate (render) the layout.
- A call to `setContentView` inside the `onCreate` callback inflates the main layout of the activity.
- The inflated layout and each view in it has a corresponding Java object that represents it visually, when displayed on the screen (layout objects themselves are not actually visible).

Implementing the main activity



- Each view defined in a layout should have a unique identifier.
- You can look-up that identifier in the `Attributes` panel for the view (look for the attribute `id` at the very top).
- A call to `findViewById` is used to obtain a reference to a UI-control object created by Android, during the process of inflating the layout.
- Our code does not create these objects and we don't have their references. We must gain access to them.



Implementing the main activity

- At this point, you will likely see several types (Java classes) highlighted in red.
- This indicates an undefined (unknown) identifier.
- For each unknown identifier, position the curser on an ident in red and type Left Alt+Enter (on Windows) or Option+Enter (on a Mac) to automatically add the needed import.

Implementing the main activity

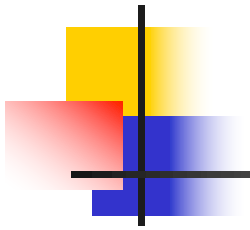
- Now, we need to define a listener for the button (its event handler).
- Add the following lines right after the last call to `findViewById`. It is a Java lambda expression defining the button's listener.

```
pushButton.setOnClickListener( (view) -> {
  counter++;
  countertView.setText( "Pushes: " + counter );
 } );
```

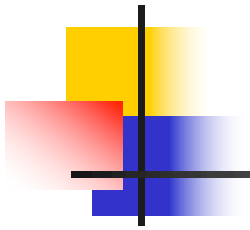
Complete MainActivity.java code

```
package edu.uga.cs.pushcounter;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    private int counter = 0;
    private TextView countertView;
    private Button pushButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        countertView = findViewById( R.id.textView );
        pushButton = findViewById( R.id.button );

        pushButton.setOnClickListener( (view) -> {
                counter++;
                countertView.setText( "Pushes: " + counter );
            } );
    }
}
```

Submitting the Project

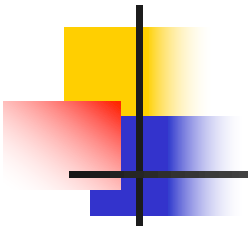


- Run the app in the emulator again to test its operation
- If your app is running well, click the Push button a few times and **take a screenshot of your emulator** running your implemented app and upload it to the **Push Counter** assignment folder on eLC.
- **You will have until midnight (11:59pm) to submit the screenshot.**
- **IMPORTANT**: if you didn't work alone, submit a text file with both your names, or add a comment to your screenshot submission; I must know who worked with you.



Sizes in Android

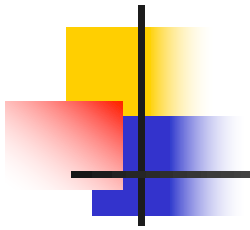
- Sizes in Android are often specified in DP units
- dp, as a unit, stands for density-independent pixels (or device-independent pixels).
- Traditionally, sizes of graphical elements on a digital screen have been measured in pixels (px), referring to numbers of pixels (dots) on the screen.
- However, physical screen sizes are measured in inches (or millimeters).
- A screen has a specific pixel density, usually measured in pixels per inch, or as we normally refer to it, dots per inch (dpi).



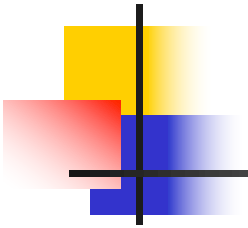
Sizes in Android

- Font sizes are usually measured in points (pt).
- A point represents the smallest measurable unit in print.
- In desktop publishing, 1 point, called a desktop publishing point (DTP) is exactly 1/72 of an inch (or, 72 points = 1in). DTP is commonly referred to as point (pt), as well.
- DTP should not be confused with a printing press point, which has varied in size over the times and across different countries (printing press point was introduced in Italy in 1517).

Sizes in Android

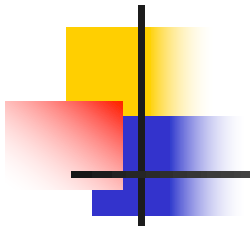


- Note that the absolute size of one pixel (a dot) can vary from screen to screen.
- For example, on a typical 24-inch monitor with a resolution of 1920x1200, the screen resolution is 94.34 dpi (dots per inch).
- So, one pixel (dot) measures about 1/94 of an inch (0.11), or 0.2692mm (it is called dot pitch).
- But, on a 7-inch Google Nexus, with the same resolution (1920x1200), the screen resolution is 323.45 dpi, and one pixel measures just .0031 of an inch (or 0.0785mm). Much smaller than on a bigger monitor with the same resolution!



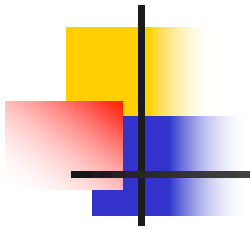
Sizes in Android

- So, there is a *disconnect* between the actual size of a graphical element
  - measured in pixels and
  - measured in inches (or millimeters), as displayed on a specific screen.
- For example, consider a text field of size 94 x 24 px (measured in pixels) on a 24-inch monitor, used in our example on the previous slide.
- This text field will roughly appear to be 1 inch wide and 1/4 inch high (the monitor's resolution is just over 94dpi, so 94px = 1 inch).



Sizes in Android

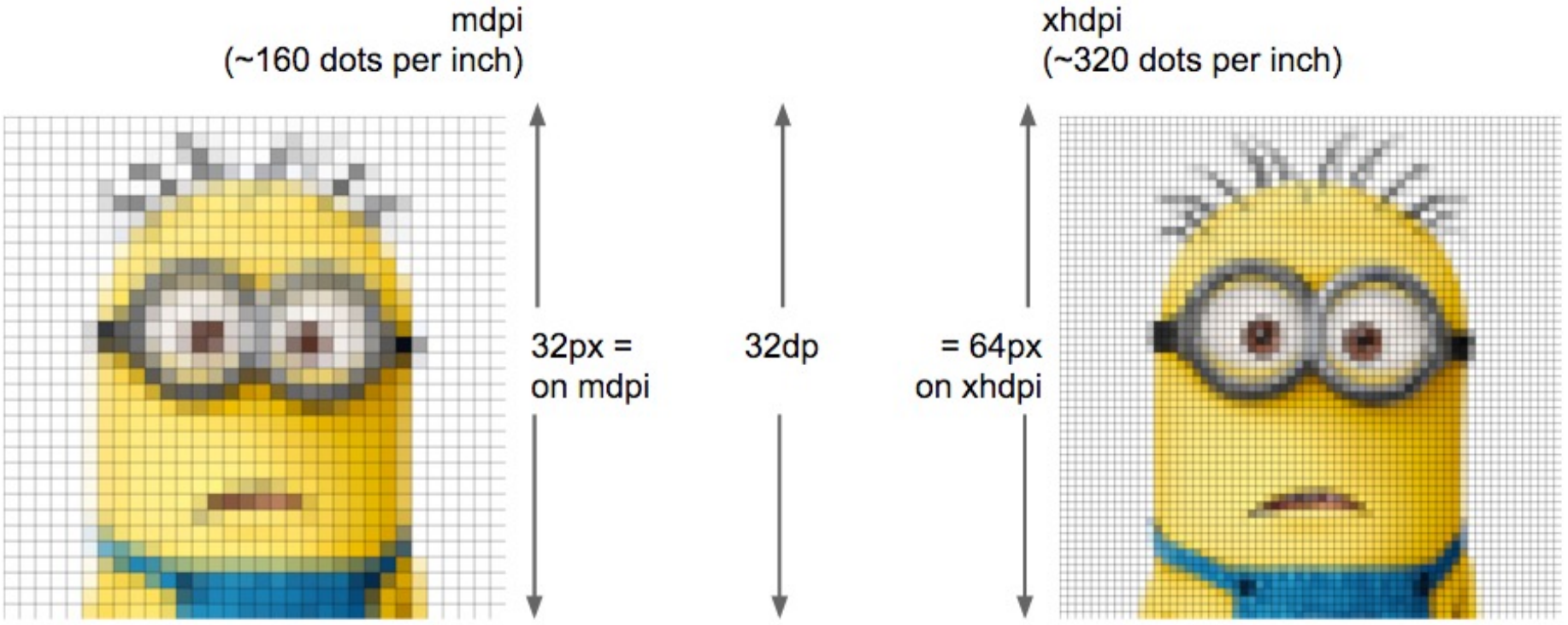
- However, consider the same text field on a Google Pixel XL's screen, with resolution of 534 dpi.
- The same text field will now have the width of just .18 of an inch! (94/534 = .176029 of an inch).
- In general, images measured in pixels appear smaller and smaller as the screen resolution increases.



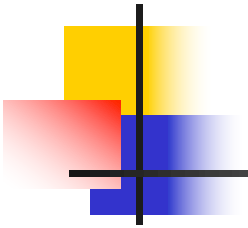
Sizes in Android

- In Android development, we will be using resolution-independent units.
- dp (density-independent pixel) is an abstract unit based on the dpi of a screen. As a standard, it is set to be the size of one pixel on a display with 160 dpi.
- 1 dp equals 1/160 of inch or 0.15875 mm, which fixes the absolute size of one pixel on a screen.
- So, in our Push Counter app example, the 16dp margin size we just used will be in fact 0.1in (or 2.54mm) on *any* screen!

Sizes in Android

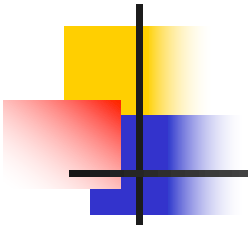


Images are from a post on stackoverflow.com



Sizes in Android

- A somewhat similar idea has been used on Apple devices.
- A **point** is a unit (abstract) that helps retain a consistent visual content, regardless of how it is displayed.
- Information about the image's scale factor is included by appending "@1x", "@2x" or "@3x" to the image's filename.



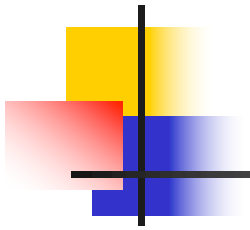
Sizes in Android

- Font sizes in Android are measured in sp units.
- sp (scale-independent pixel) is very similar to dp but refers to the user's font size preference.
- When we use such resolution-independent sizes, the images, views, text, etc. will retain their *intended* actual sizes.

Lifecycle of an Android Activity

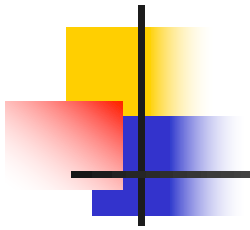
- Android allows multiple apps to run concurrently (provided memory and processing power are available).
- Usually, only one is executing at any given time and others are in the background.
  - NOTE: beginning with version 7, Android added the ability to run apps in the split-screen mode and allow two apps at the same time.
  - Also, beginning with version 12, Android added multi-window mode, and allows now *multiple* apps apps running simultaneously.

Lifecycle of an Android Activity



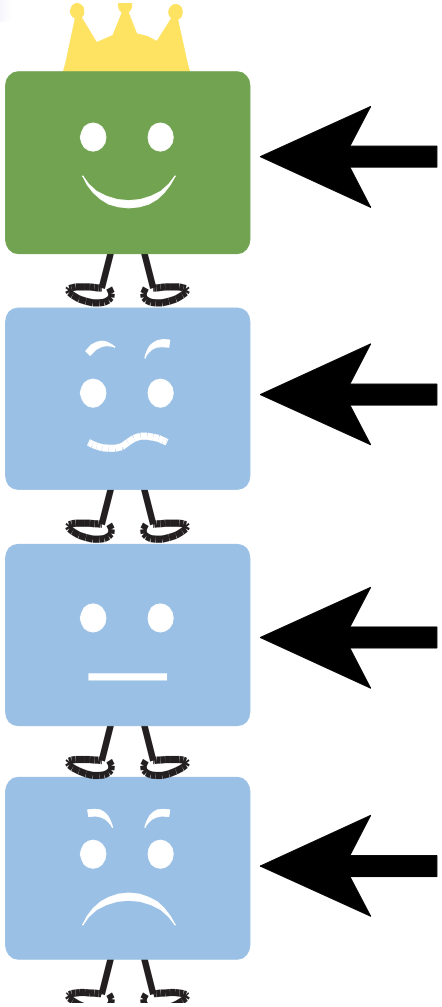
- Applications can be interrupted and stopped when events such as phone calls occur.
- With no split-screen, there can be only one active application visible to the user at a time — specifically, a single activity of an application is in the foreground, at any given time.
- This semester, we will only focus on one-activity-at-a-time development.

Lifecycle of an Android Activity



- Android keeps track of **all** activities currently running by placing them on the activity stack.
- The activity stack is often referred to as the "back stack." But it is not a stack, technically speaking.
- When a new activity starts, the activity on top of the activity stack (the current foreground activity) is stopped, and the new activity is pushed onto the top of the stack.
- When that new activity finishes, it is removed from the activity stack, and the previous Activity in the stack resumes.

Lifecycle of an Android Activity



I am the top Activity.
User can see and interact with me!

I am the second Activity in the stack.
If the user hits Back or the top Activity is destroyed, the user can see and interact with me again!

I am an Activity in the middle of the stack.
Users cannot see and interact with me until everyone above me is destroyed.

I am an Activity at the bottom of the stack.
If those Activities above me use too many resources, I will be destroyed!

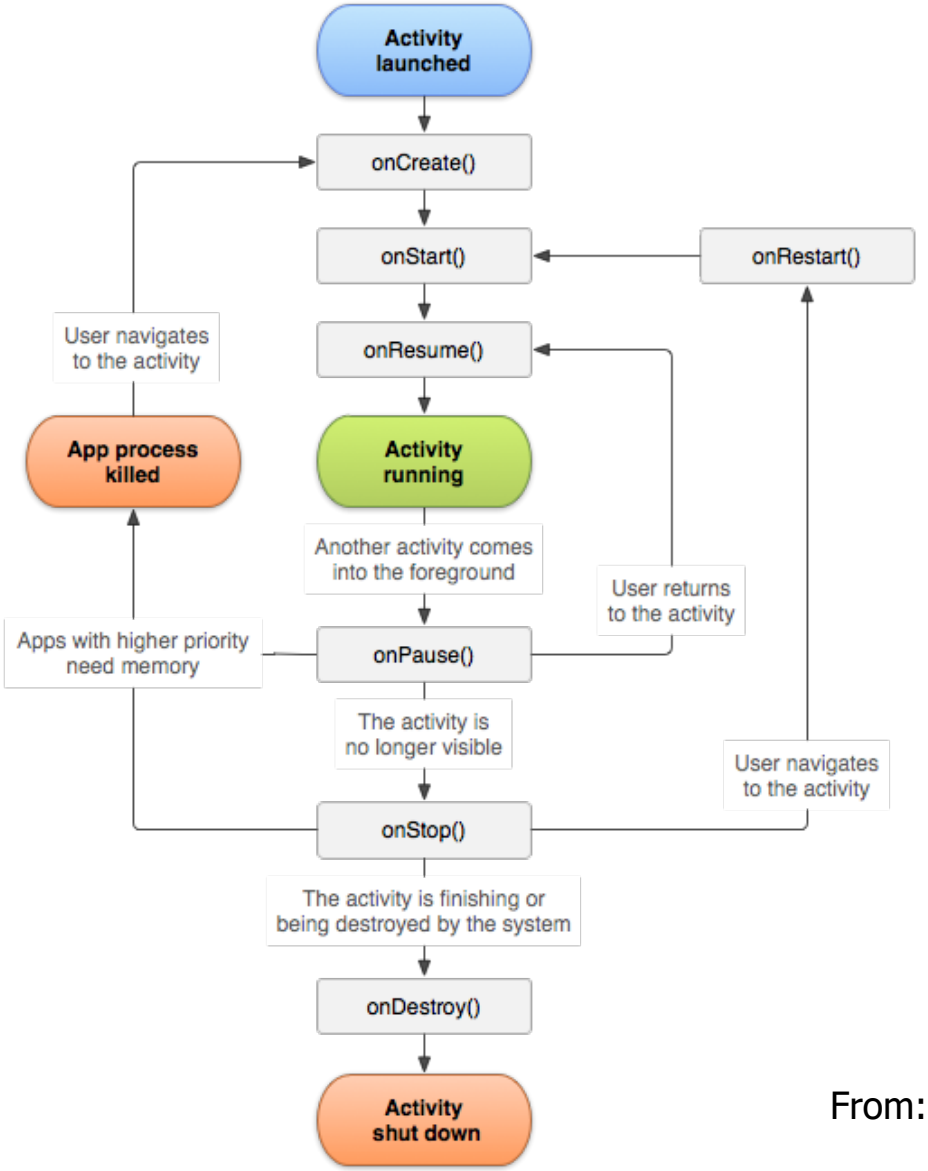
Note that the user can pick an arbitrary activity from the list of activities on the stack and bring it to the top, resuming it.

Activity's Callbacks

- An Activity class may override several additional methods involved in its lifecycle.

```
public class MyActivity extends Activity {
    protected void onCreate(Bundle
                                savedInstanceState);
    protected void onStart();
    protected void onResume();
    protected void onRestart();
    protected void onPause();
    protected void onStop();
    protected void onDestroy();
}
```

Lifecycle of an Android Activity



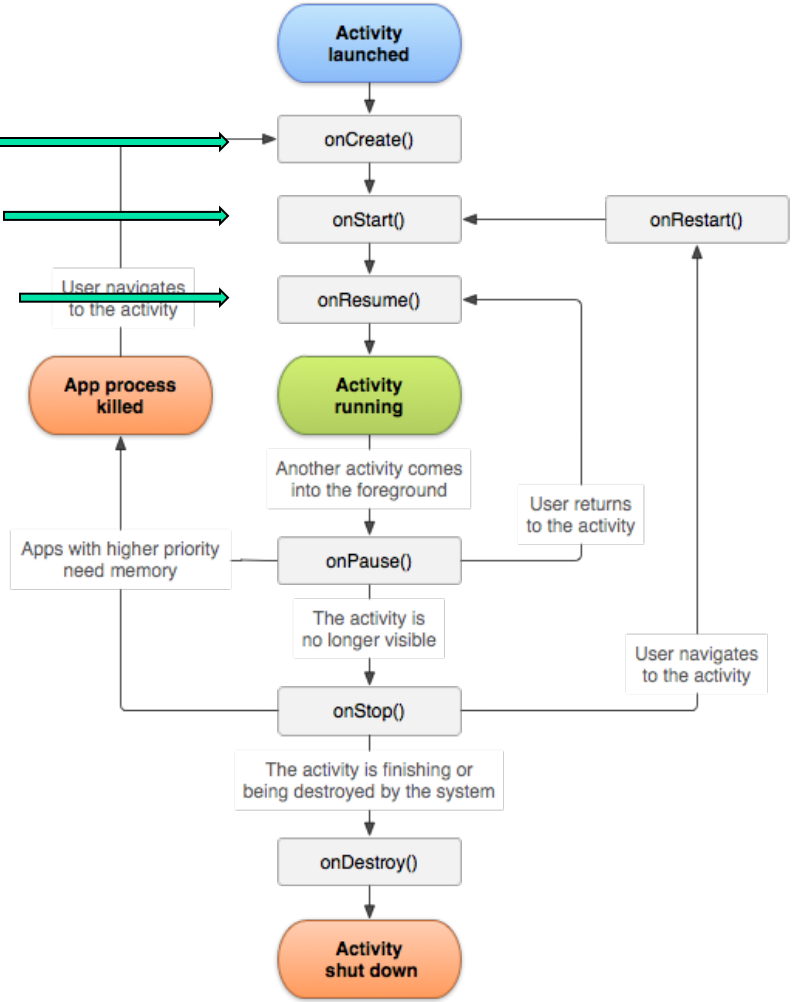
From: developer.android.com

Common Callback Sequences

- Starting an activity (the user starts an app)

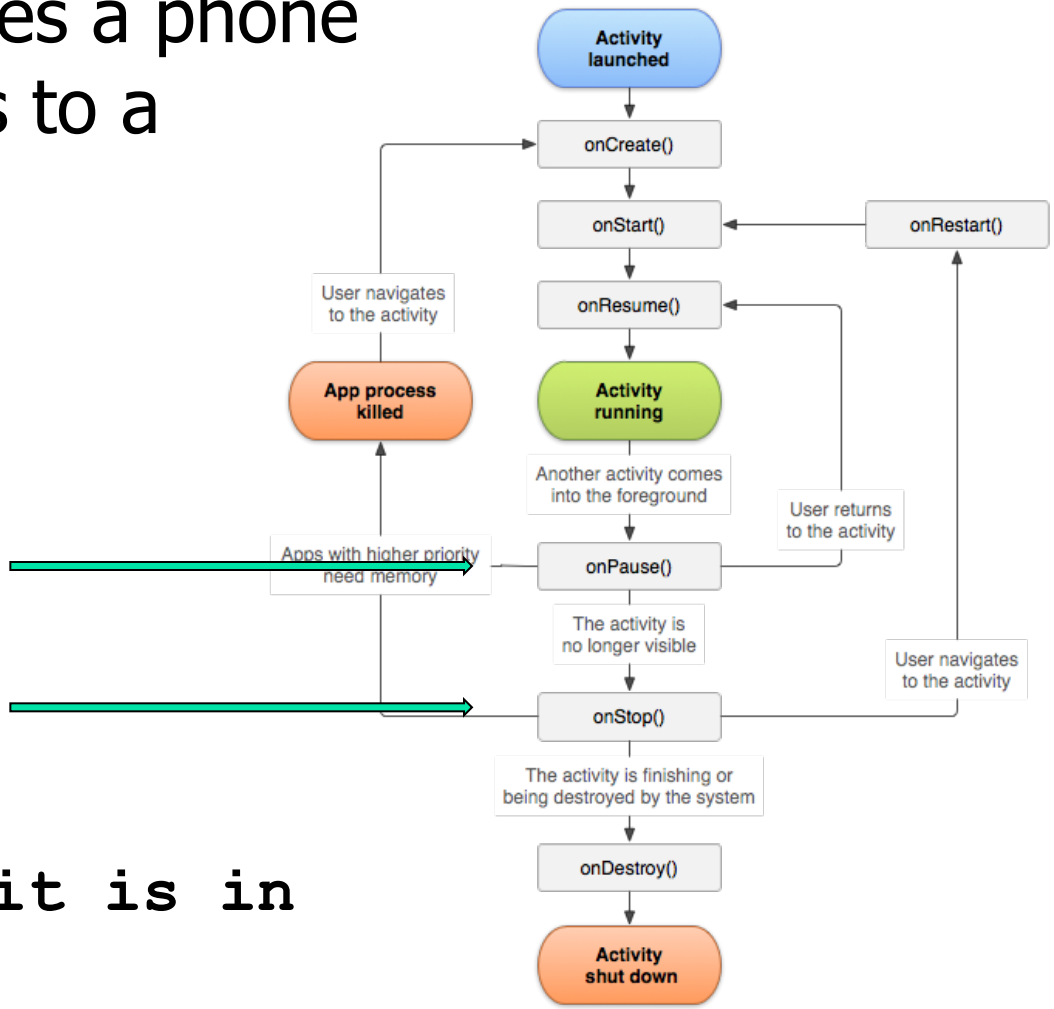
1. `onCreate()`
2. `onStart()`
3. `onResume()`

The activity is running in foreground



Common Callback Sequences

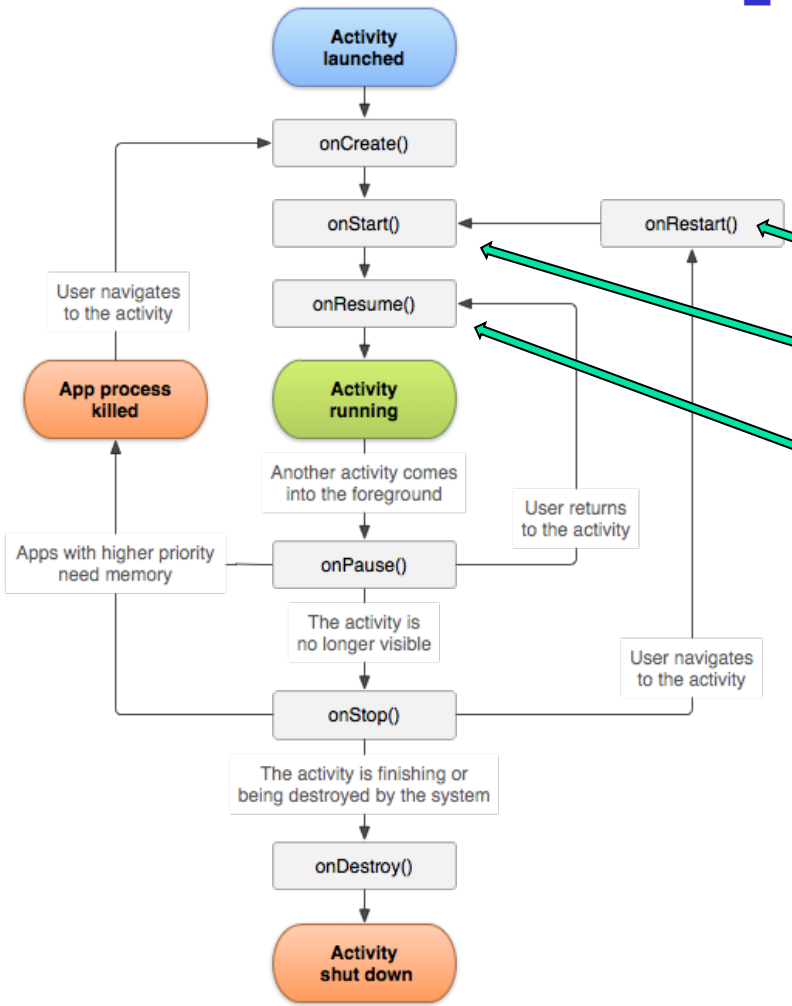
- Stopping an activity (e.g., the user receives a phone call or switches to a different app)



1. **`onPause()`**
2. **`onStop()`**

`The activity is stopped (it is in background)`

Common Callback Sequences



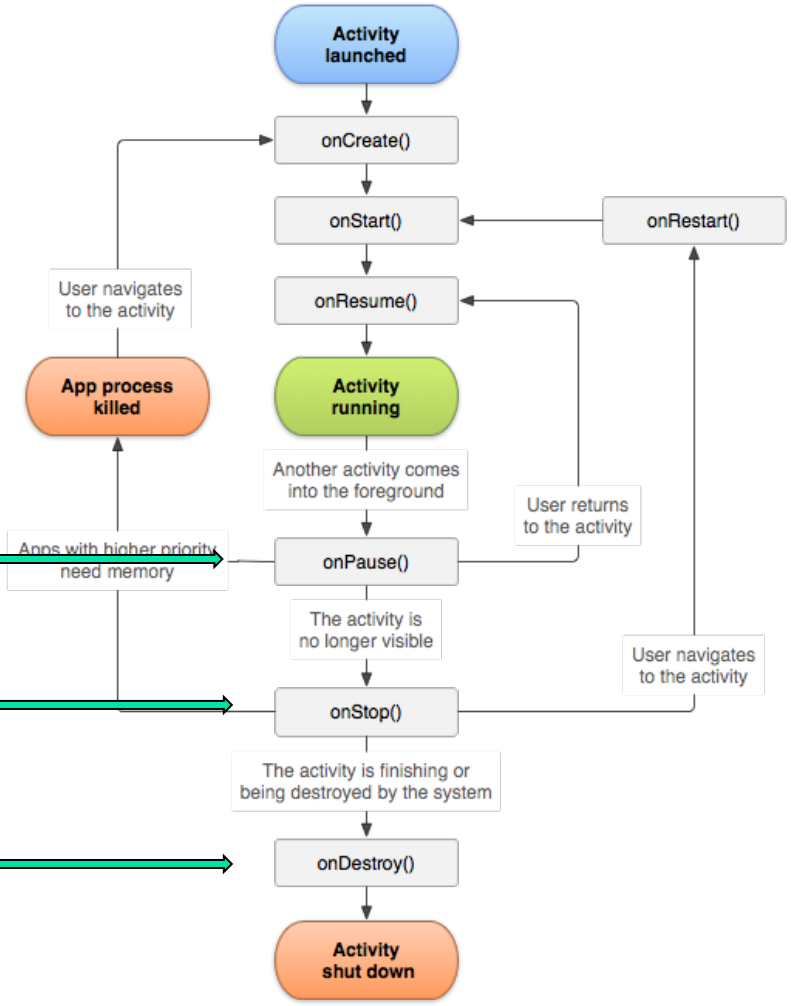
- Resuming an activity (e.g., user finishes a call or switches back to this app)

1. `onRestart()`
2. `onStart()`
3. `onResume()`

The activity is running again (it is back in foreground)

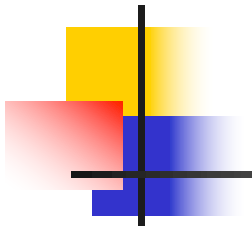
Common Callback Sequences

- Terminating an activity (the user kills the app)



1. **`onPause()`**
2. **`onStop()`**
3. **`onDestroy()`**

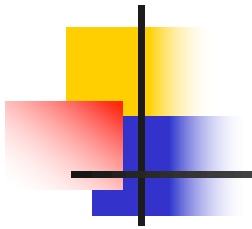
`The activity is killed`



Adding Logging Support

- Android provides logging, which is a useful feature for tracing and debugging apps
- Also, logging is a valuable resource for learning Android lifecycle.
- User interface to Android logging is implemented in the `Log` class of the `android.util` package.
- More information about Android log is available at:

  https://developer.android.com/studio/debug/logcat



Adding Logging Support

- Import the `Log` class using the import statement:

```
import android.util.Log
```

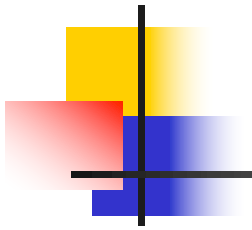
- Add a constant DEBUG_TAG (you can use a different name) for collecting your debug messages:

```
private static final String DEBUG_TAG = "Go Dawgs"
```

- For example, in the `onCreate()` method you may add:

```
Log.i( DEBUG_TAG, "MainActivity.onCreate()" );
```

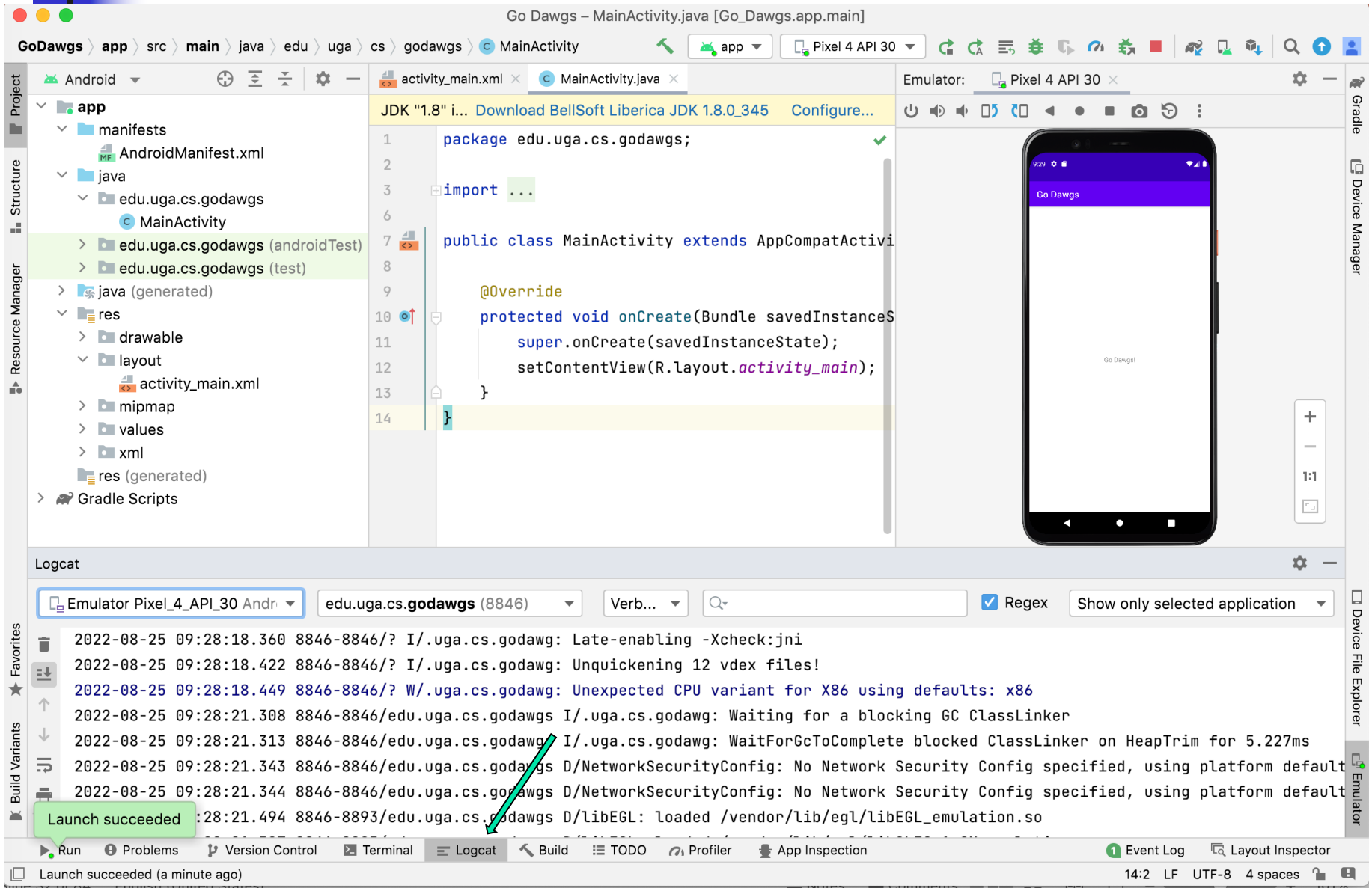
- Logging messages are available in the Logcat panel in Android Studio.

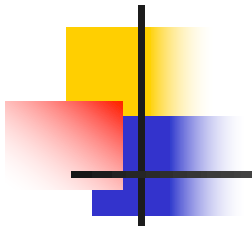


Adding Logging Support

- You can use the following methods to log certain information:
  - `Log.e()` - log error messages
  - `Log.w()` - log warning messages
  - `Log.i()` - log info messages
  - `Log.d()` - log debug messages
  - `Log.v()` - log verbose messages
- The above methods can be used to log messages in according to their verbosity level, from ERROR (least verbose) to VERBOSE (most verbose).
- It is then possible to set `Logcat` to show messages only **at or above** a desired level.

Adding Logging Support

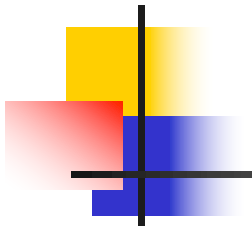




Tracing errors

- What if your application does not work as intended?
- For example, what if it simply crashes?
- Before going any further, you need to become familiar with tracing the potential problems.
- To illustrate some useful tools, let's create an error in the `Push Counter` application, **on purpose**!

Tracing errors

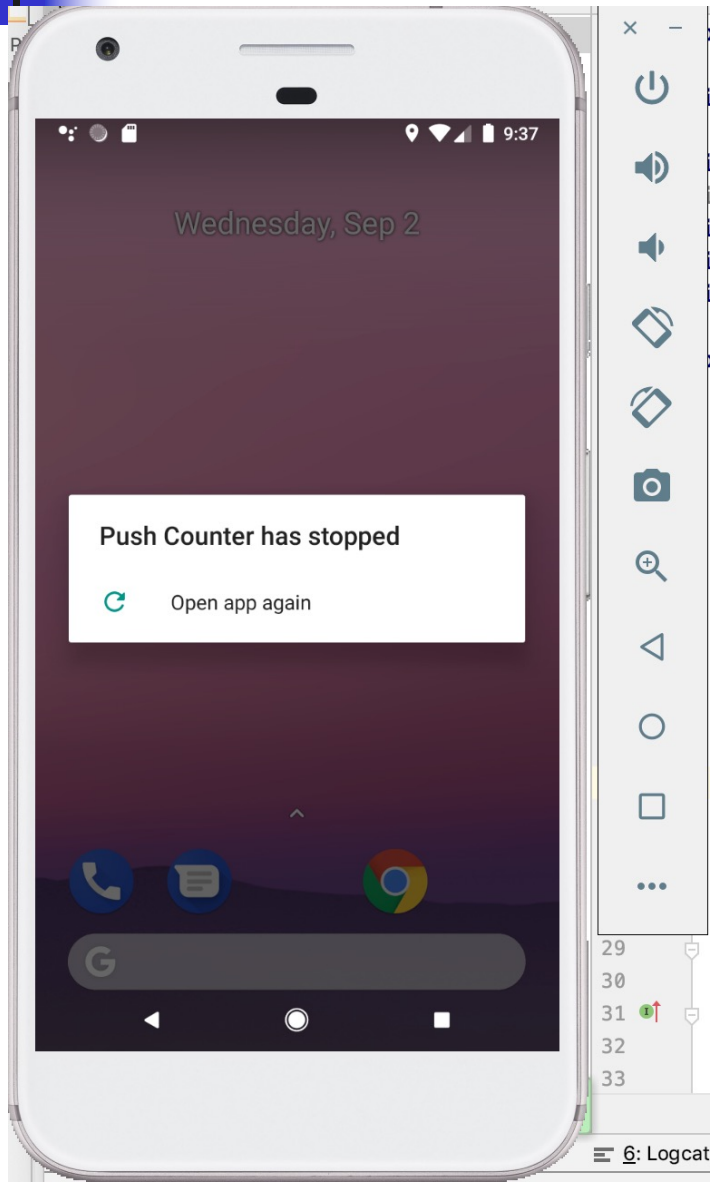


- In your project, go back to the source file called `MainActivity.java`.
- Comment out the line which set the countertView variable to the Text object (`countertView` will be `null` as a result):

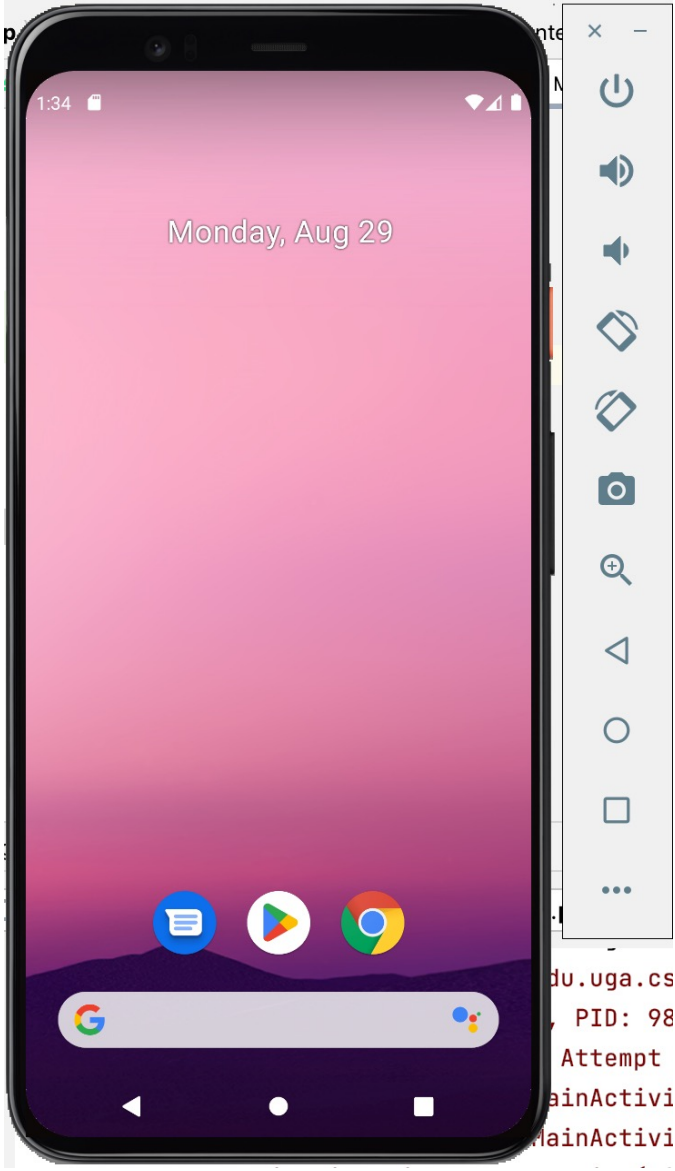
```
//countertView = findViewById( R.id.textView );
```

- Run the emulator again and click on the `Push Me!` Button.
- You will a screen letting you know that the application stopped! A real bummer!

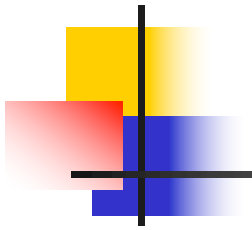
Tracing errors



Or just the home screen will be visible



Tracing errors

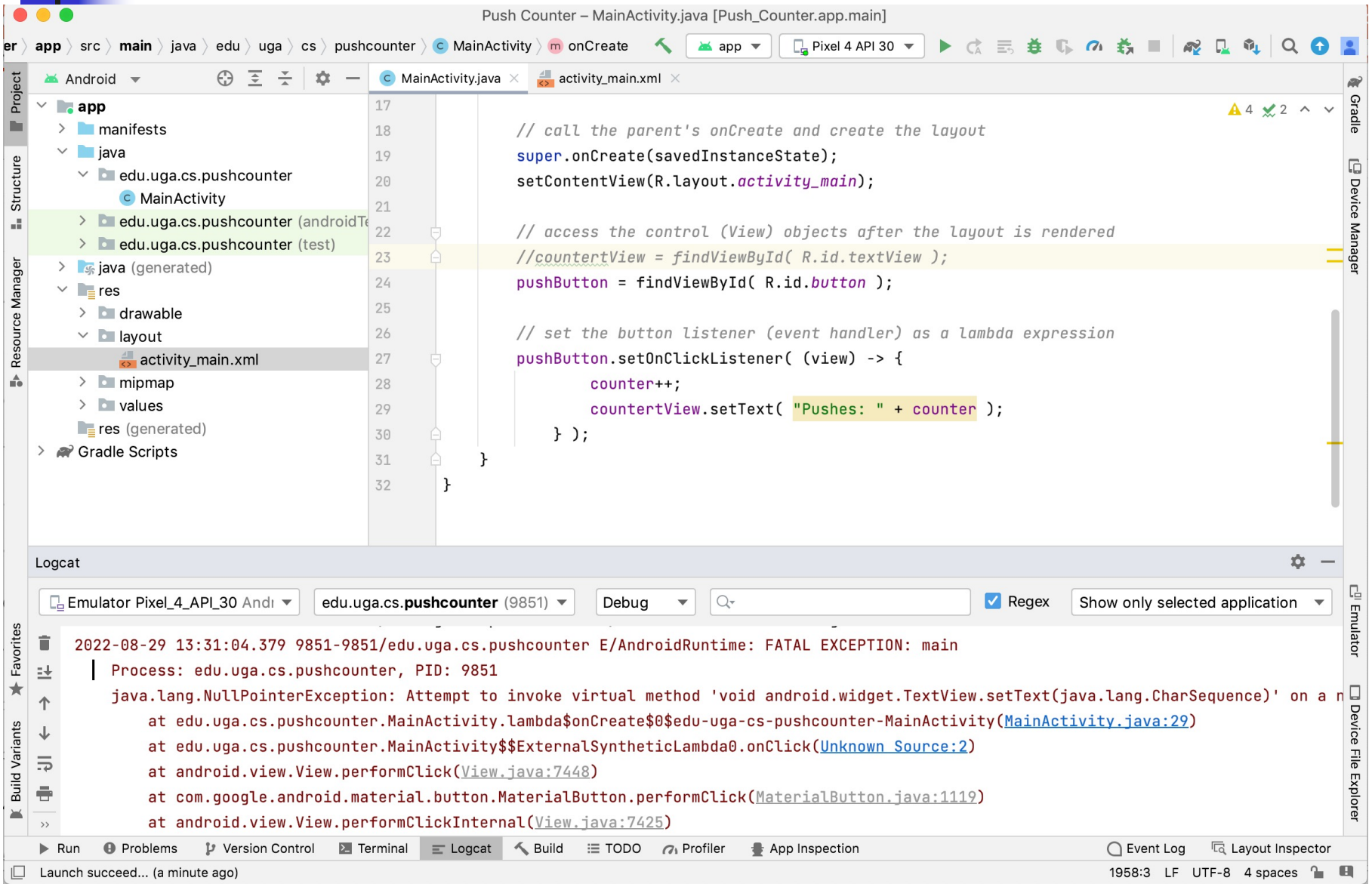


- Let us see what happened.
- In the Android Studio menu, select

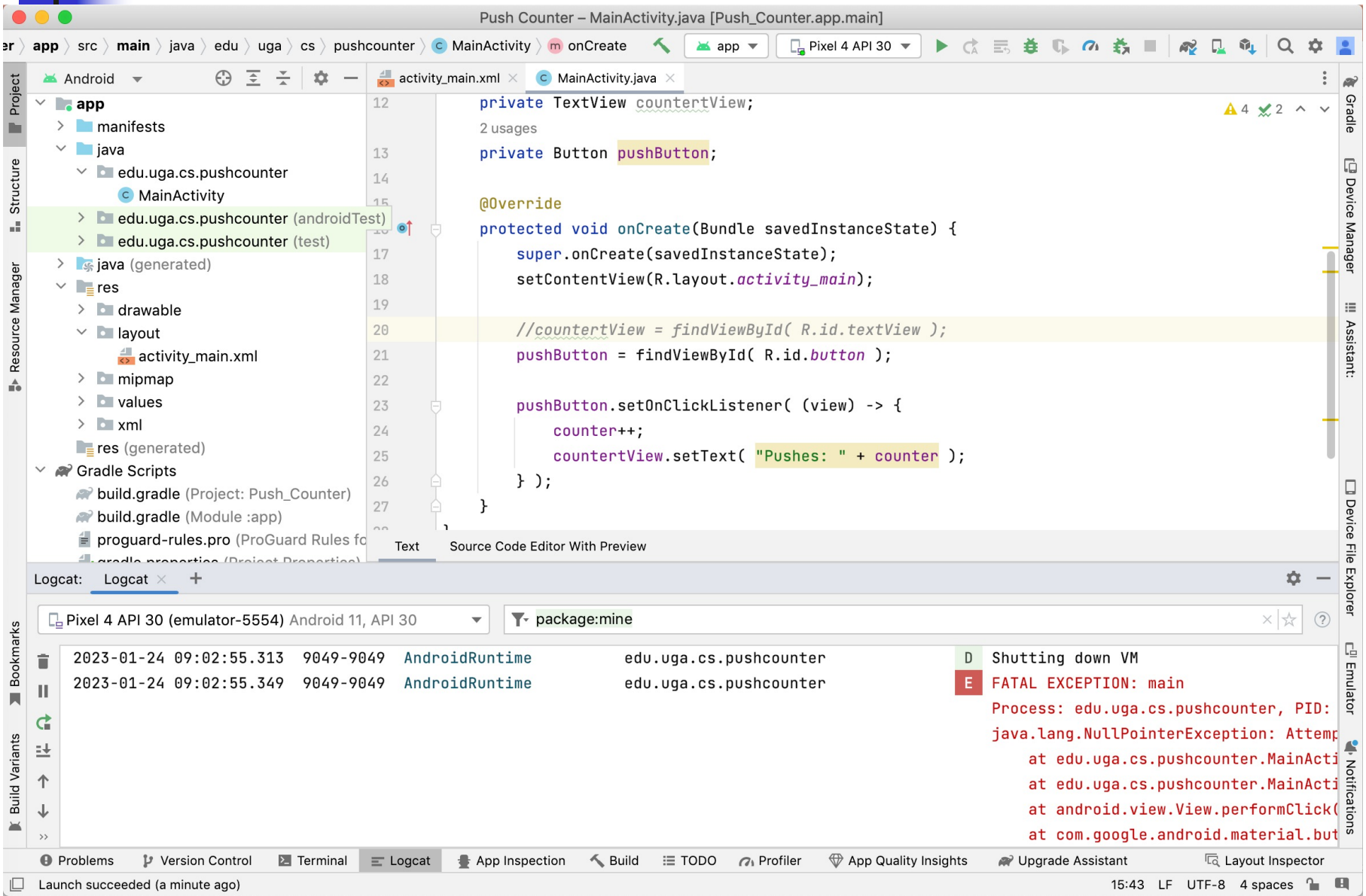
  `View` -> `Tool windows` -> `Logcat`

- A `Logcat` panel will open at the bottom of the Android Studio window.
- This is a log of the major events in the lifecycle of your application's run.
- You will be able to see the source of the exception, most likely with the info on the offending class and a specific line in the source file.

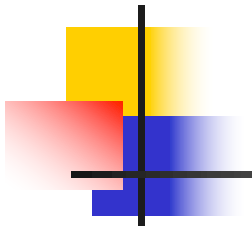
Tracing errors



Tracing errors (logcat v2)



Debugging in the Emulator



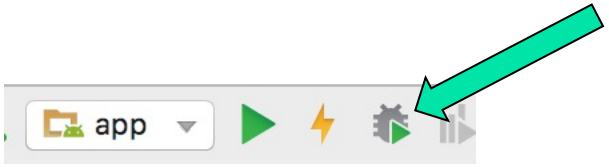
- We can use the debugger within Android Studio to inspect/debug our application
- Since we know the line which caused the exception, set the breakpoint on the line just before it, that is:

```
counter++;
```

- You can do this by clicking on the gray area just to the left of the line above.
- A red circle to the left of the line should appear.

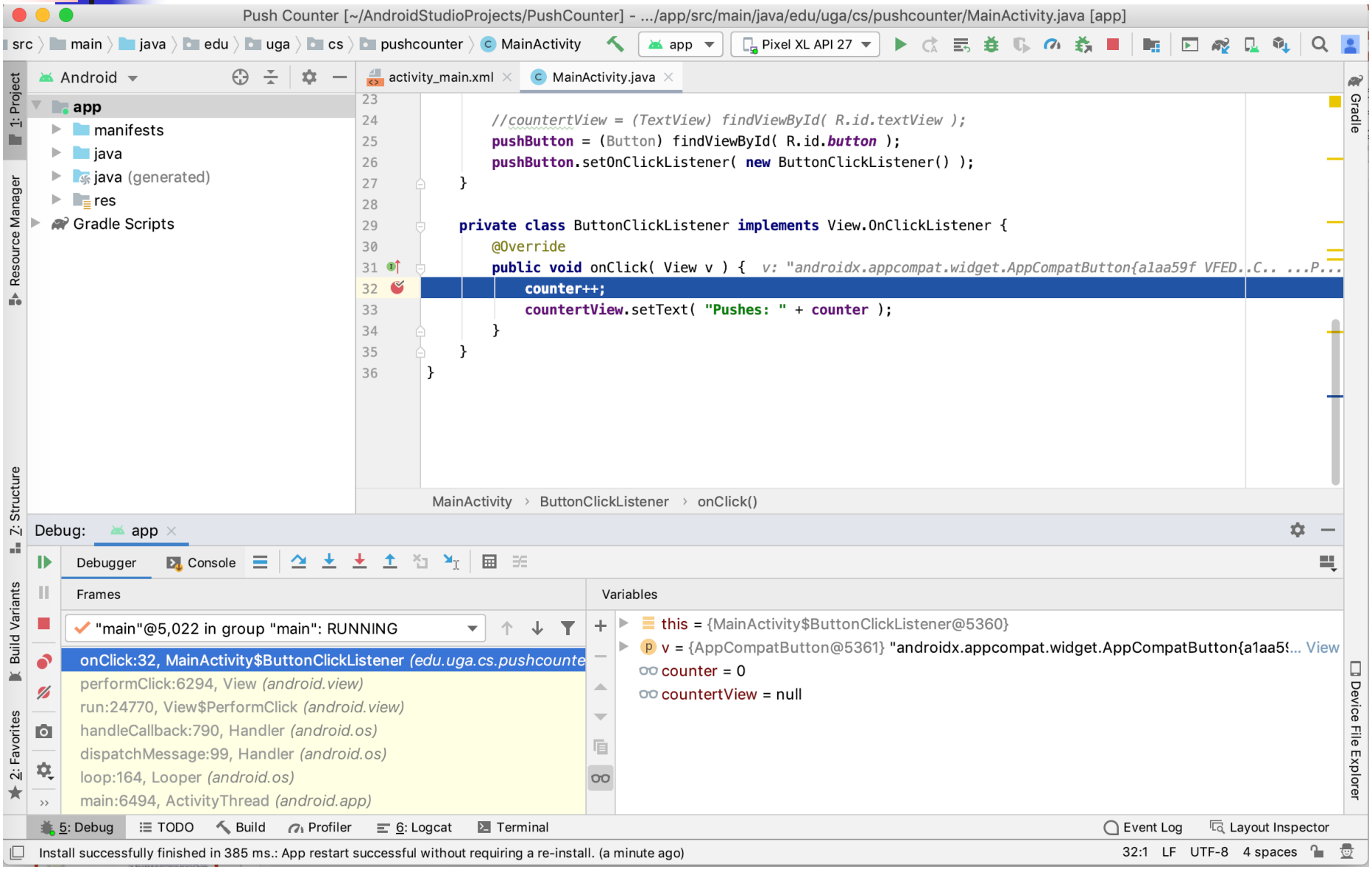
Debugging in the Emulator

- Start the application, but this time use the `Debug 'app'` selection in the `Run` menu, or click on the Debug icon



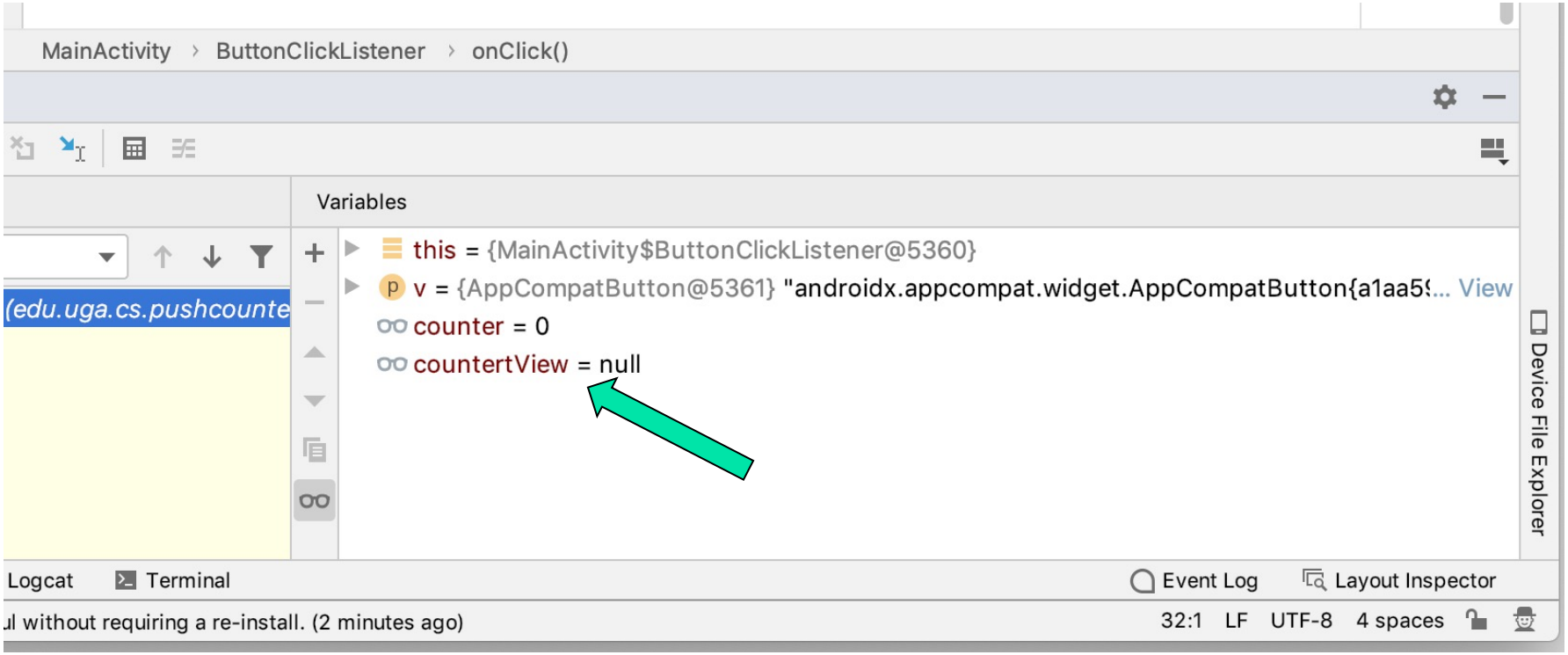
- The app will start. Again, click on the `Push Me!` button. The app will stop at the set break point.
- Now, you should be able to control the debugger using a number of selections in the debug panel:

Debugging in the Emulator



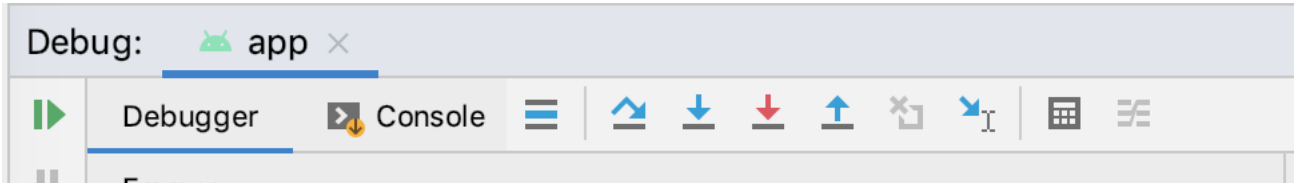
Debugging in the Emulator

- You should be able to plainly see the problem:

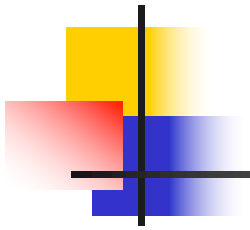


Debugging in the Emulator

- You should be able to control the debugger using a number of selections in the debug panel:



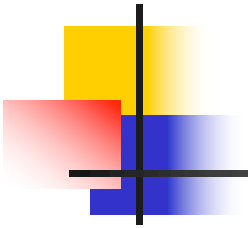
- For example, you can use the  icon (also the F8 shortcut on a Mac) to move the execution forward (you should be able to step through the execution to the point which causes the problem).



Testing App on Android Device

- Let us run the application on a real Android device.
- On Mac OS, you may have to install Android File Transfer from

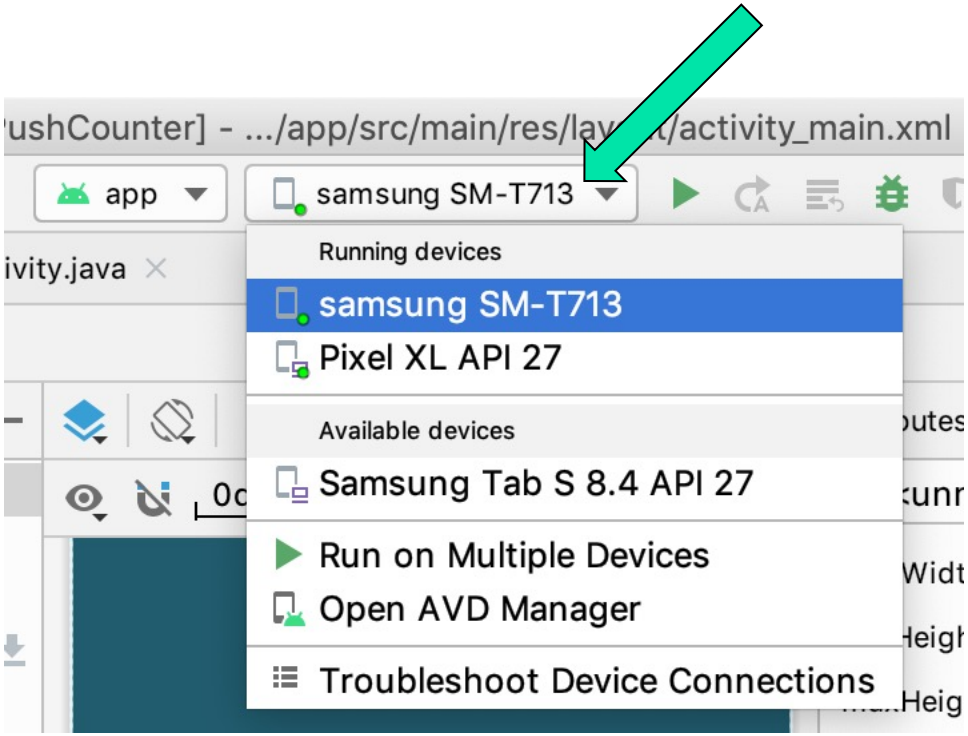
  https://www.android.com/filetransfer/
- Connect an Android device to your computer using a USB cable.



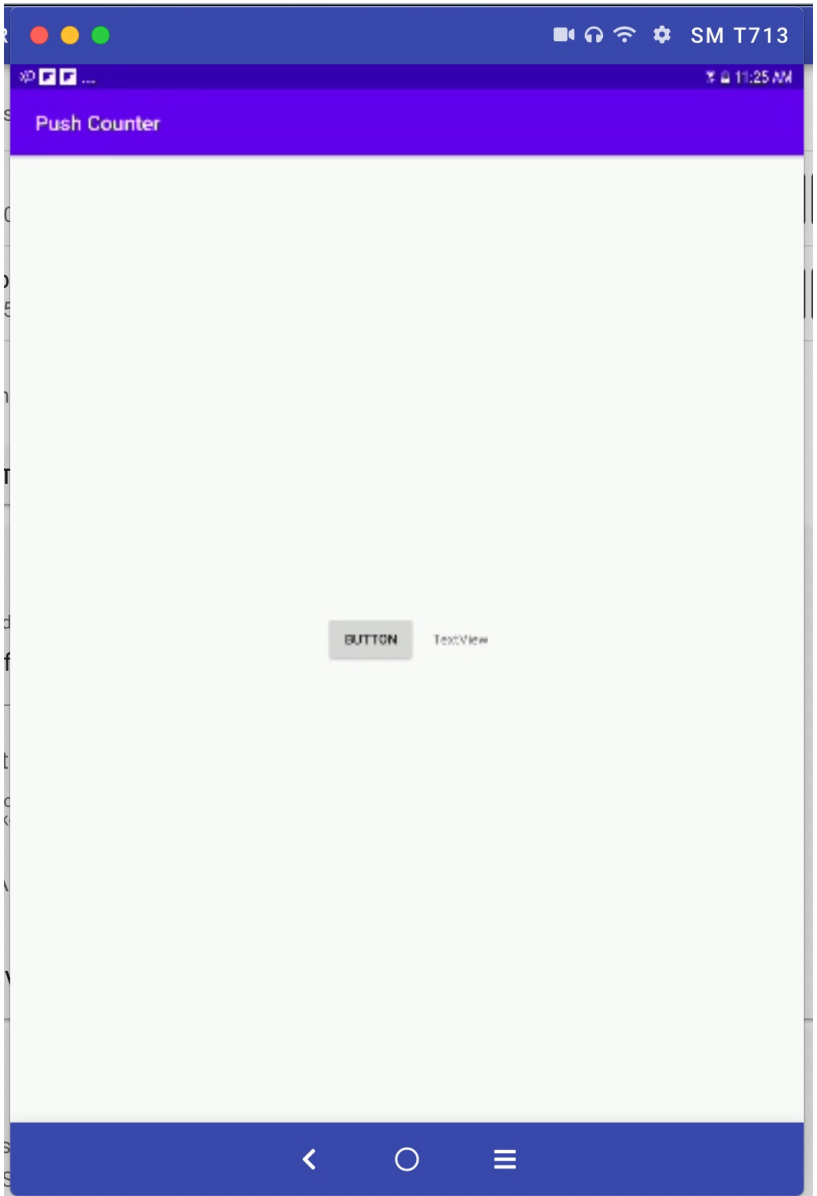
Debugging Application on Hardware

- You should now see a real Android device listed as an option in the selection of devices drop down – for example, a Samsung tablet.
- Select it to run the app.

Debugging Application on Hardware

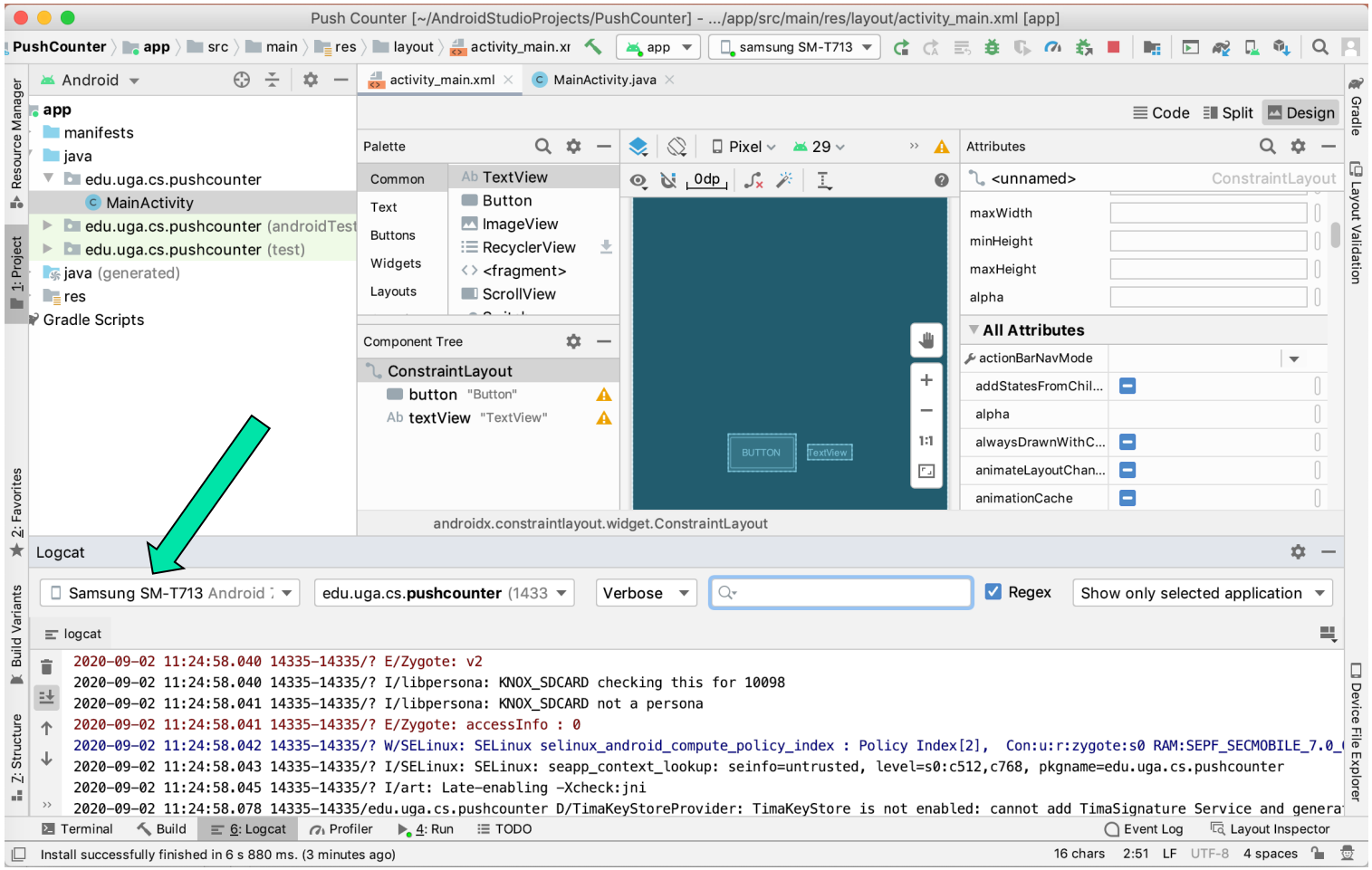


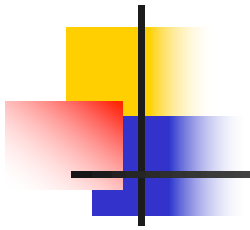
Debugging Application on Hardware



Debugging Application on Hardware

- You will see the Logcat content for this run (you may have to switch to that device's logcat):





Summary

- We have learned how to add sample applications and create new projects of our own.
- We have learned how to build, run, and debug our projects.
- We have learned the difference between the `Android` symbolic view and `Project` view of Android Studio.
- We are now able to log errors.

References and More Information

- Android Activity:
  - *https://developer.android.com/reference/android/app/Activity*
- Activity Lifecycle:
  - *https://developer.android.com/guide/components/activities/activity-lifecycle*
- Android Linear Layout:
  - *https://developer.android.com/guide/topics/ui/layout/linear*
  - *https://developer.android.com/reference/android/widget/LinearLayout*
- Android Button:
  - *https://developer.android.com/reference/android/widget/Button*
- Android TextView:
  - *https://developer.android.com/reference/android/widget/TextView*