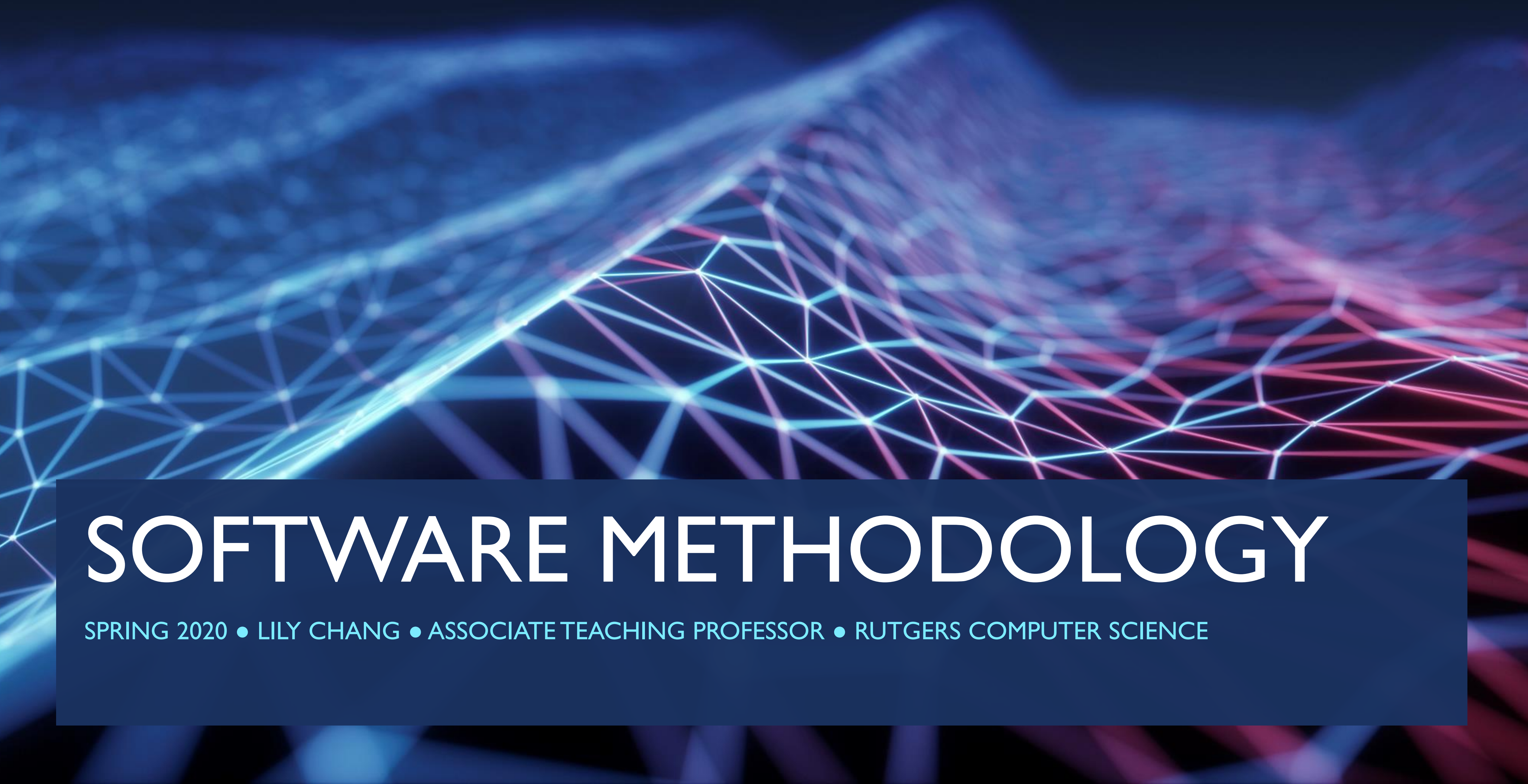
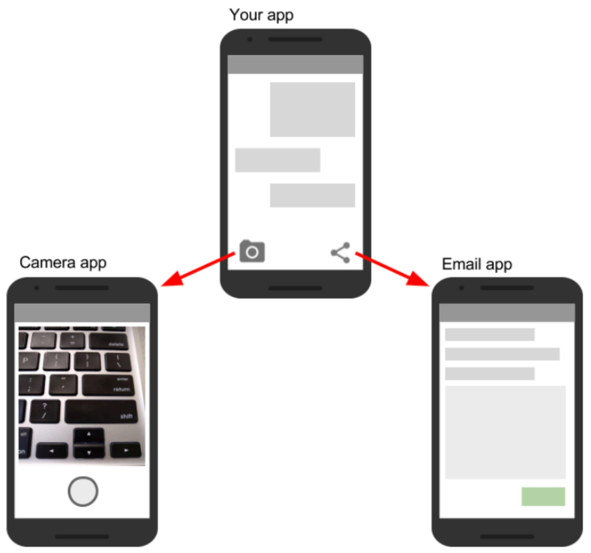


SOFTWARE METHODOLOGY

SPRING 2020 • LILY CHANG • ASSOCIATE TEACHING PROFESSOR • RUTGERS COMPUTER SCIENCE

ESSENTIAL COMPONENTS IN AN ANDROID APP

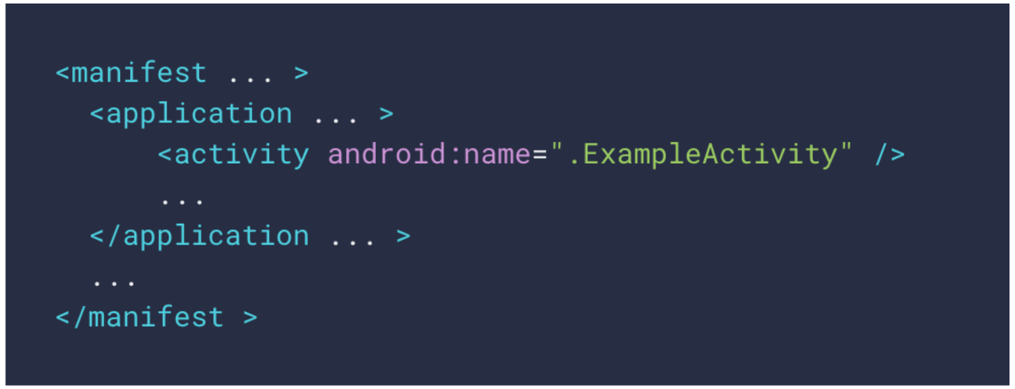
ACTIVITY



- The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place. Instead, the user journey often begins non-deterministically.
  - For example, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.
- The **Activity** class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.
  - When one app invokes another, the calling app invokes an activity in the other app,
- Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

ACTIVITY

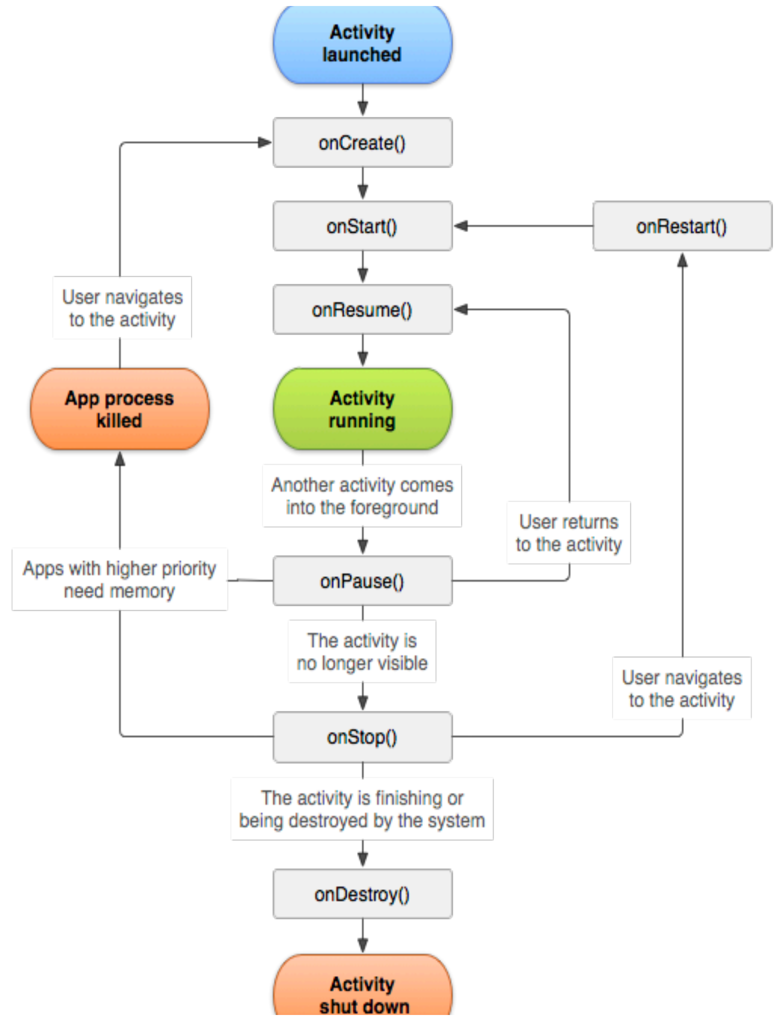
- An activity provides the window in which the app draws its UI. This window typically fills the screen, but may be smaller than the screen and float on top of other windows. Generally, one activity implements one screen in an app.
- Most apps contain multiple screens, which means multiple activities. Typically, one activity in an app is specified as the main activity, which is the first screen to appear when the user launches the app
- Each activity is loosely bound to the other activities; there are usually minimal dependencies among the activities in an app.
- To use activities in your app, you must register information about them in the app's manifest, and you must manage activity lifecycles appropriately.



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

ACTIVITY LIFECYCLE

- Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When the user is done with the current activity and presses the Back button, the activity is popped from the stack and destroyed, and the previous activity resumes.
- Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity.
- Good implementation of the lifecycle callbacks can help ensure that your app avoids crashing


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

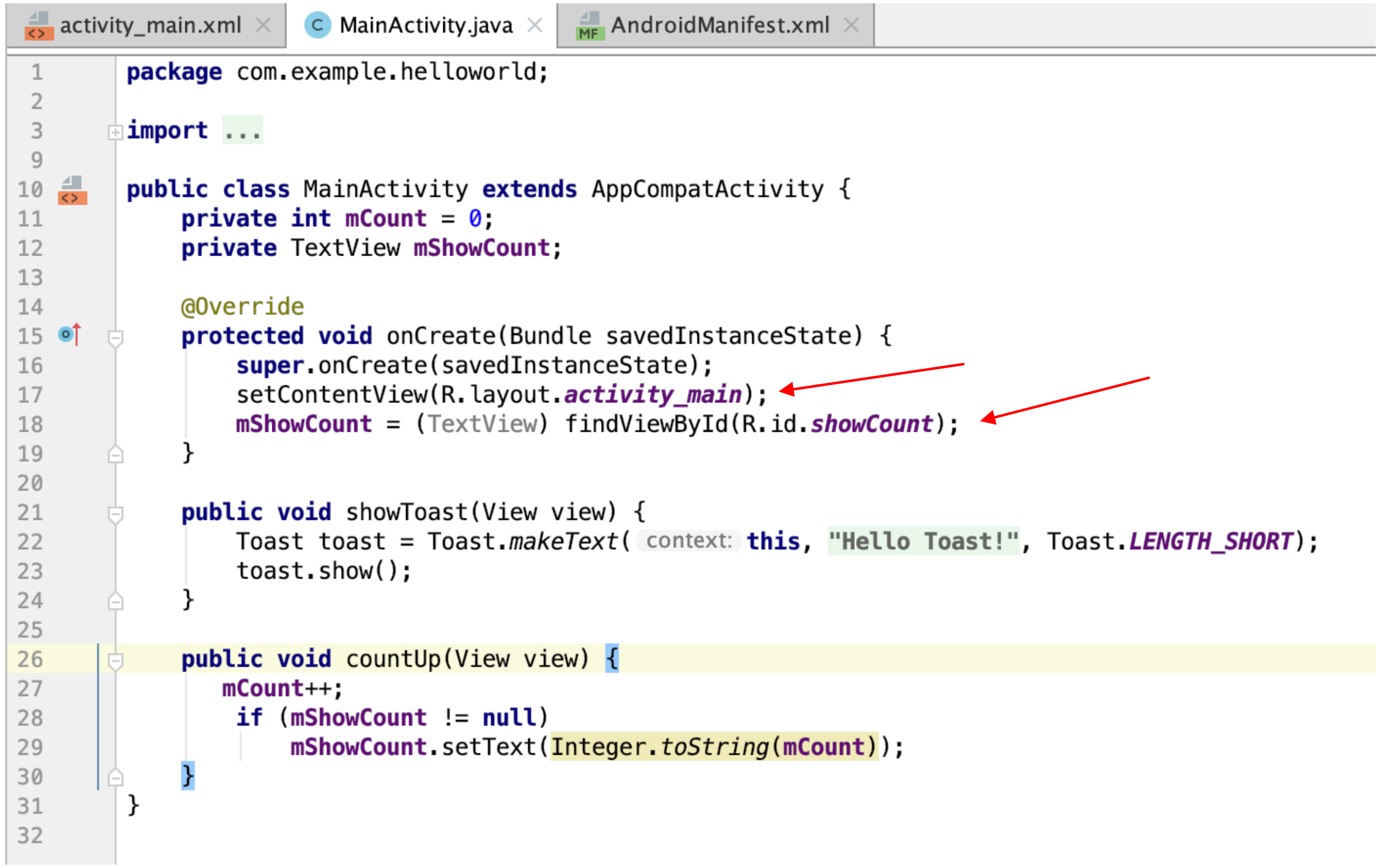
ONCREATE() METHOD

- You must implement this callback, which fires when the system first creates the activity.
- On activity creation, the activity enters the Created state. In the onCreate() method, you perform basic application startup logic that should happen only once for the entire life of the activity; for example, instantiate some class-scope variables.
- This method receives the parameter savedInstanceState, which is a Bundle object containing the activity's previously saved state, or null If the activity has never existed before

```
protected void onCreate(Bundle savedInstanceState) { }
```

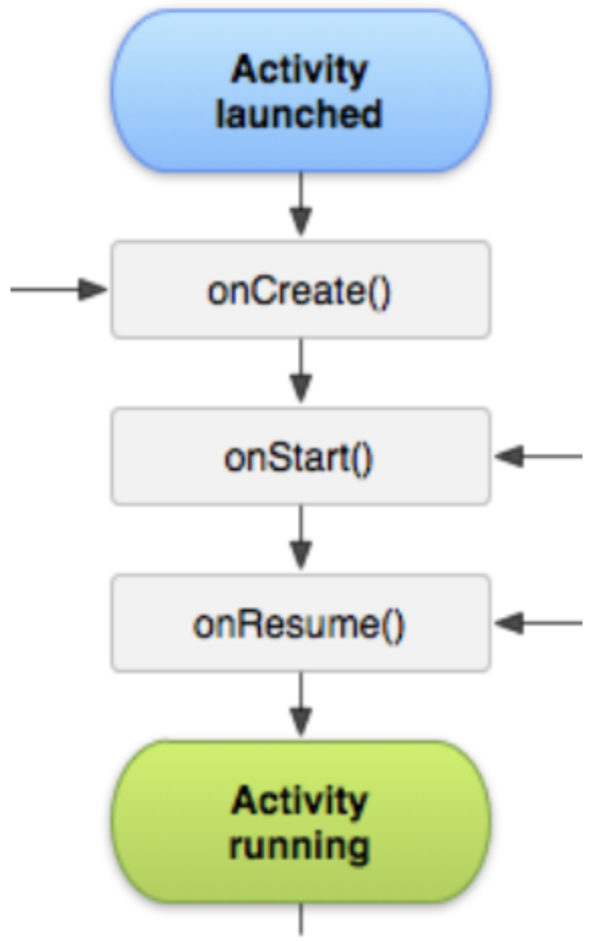
EXAMPLE

- You can bind data to lists, associate the activity with a ViewModel, and instantiate some class-scope variables.



activity_main.xml | MainActivity.java | AndroidManifest.xml

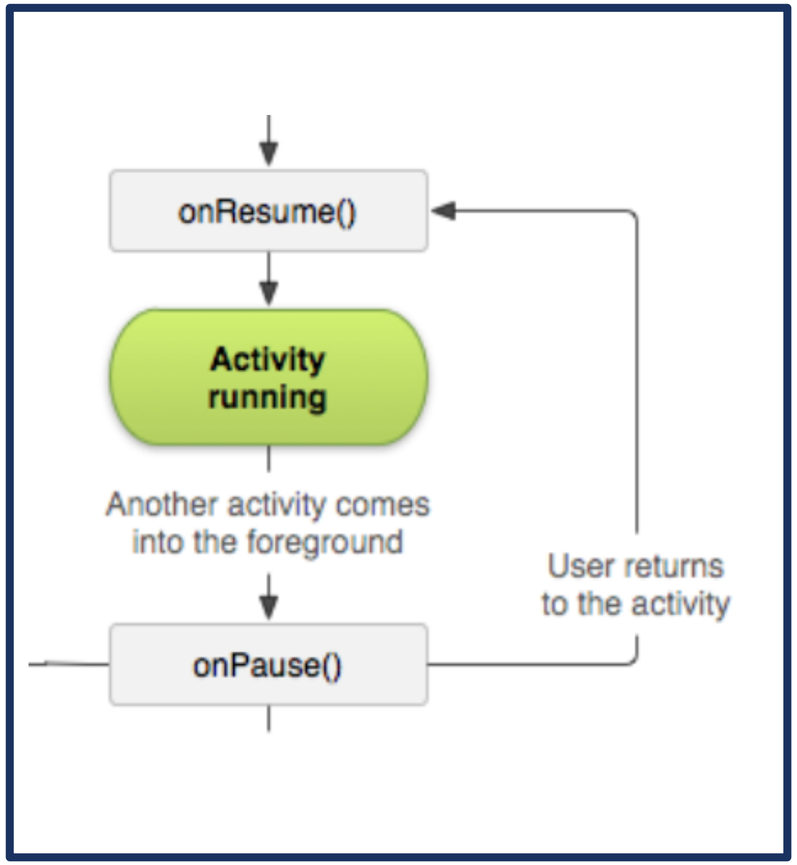
```
package com.example.helloworld;

import ...

public class MainActivity extends AppCompatActivity {
    private int mCount = 0;
    private TextView mShowCount;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mShowCount = (TextView) findViewById(R.id.showCount);
    }

    public void showToast(View view) {
        Toast toast = Toast.makeText( context: this, "Hello Toast!", Toast.LENGTH_SHORT);
        toast.show();
    }

    public void countUp(View view) {
       mCount++;
        if (mShowCount != null)
            mShowCount.setText(Integer.toString(mCount));
    }
}
```



ONSTART() METHOD

- Your activity does not reside in the Created state.
- After the onCreate() method finishes execution, the activity enters the Started state, and the system calls the onStart() and onResume() methods in quick succession
- When the activity enters the Started state, the system invokes this callback and makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive. For example, this method is where the app initializes the code that maintains the UI.
- The onStart() method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state. Once this callback finishes, the activity enters the Resumed state, and the system invokes the onResume() method.

ONRESUME() METHOD

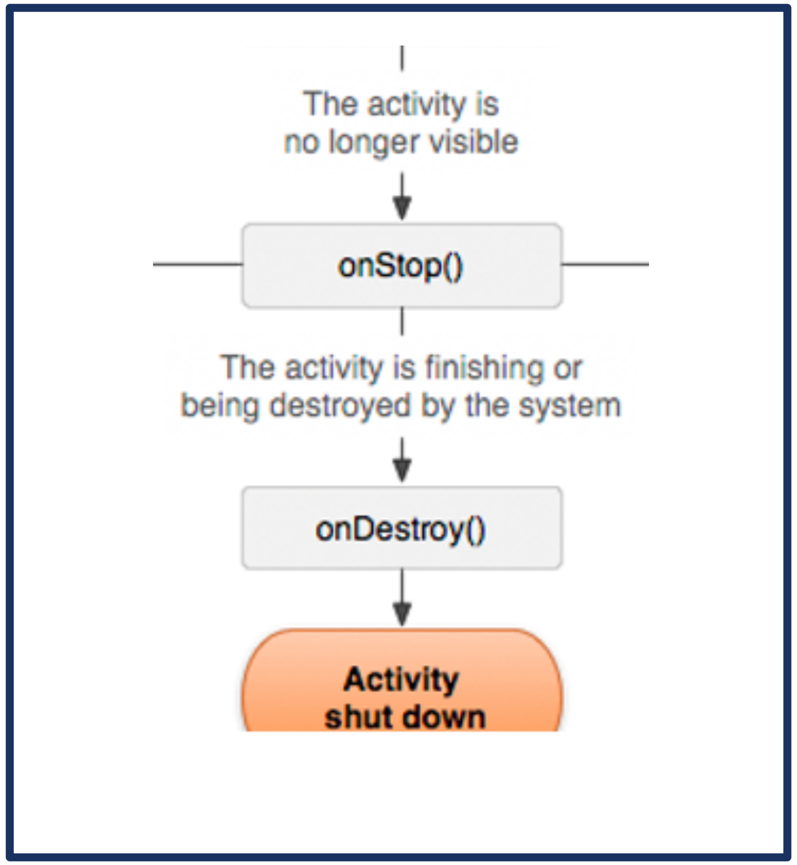

onResume()
Activity running
Another activity comes into the foreground
User returns to the activity
onPause()

- When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the onResume() callback.
- This is the state in which the app interacts with the user.
- The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user's navigating to another activity, or the device screen's turning off.
- When an interruptive event occurs, the activity enters the Paused state, and the system invokes the onPause() callback.
- If the activity returns to the Resumed state from the Paused state, the system once again calls onResume() method. For this reason, you should implement onResume() to initialize components that you release during onPause(), and perform any other initializations that must occur each time the activity enters the Resumed state.

ONPAUSE() METHOD

- The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed);
- It indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode).
- Use the onPause() method to pause or adjust operations that should not continue (or should continue in moderation) while the Activity is in the Paused state, and that you expect to resume shortly.
- You can also use the onPause() method to release system resources, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them.

ONSTOP() METHOD


The activity is no longer visible
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

- When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback.
- This may occur, for example, when a newly launched activity covers the entire screen. The system may also call onStop() when the activity has finished running, and is about to be terminated.
- In the onStop() method, the app should release or adjust resources that are not needed while the app is not visible to the user.
- You should also use onStop() to perform relatively CPU-intensive shutdown operations.

ONDESTROY() METHOD

- onDestroy() is called before the activity is destroyed. The system invokes this callback either because:
  - The activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity), or
  - The system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)
- If the activity is finishing, onDestroy() is the final lifecycle callback the activity receives. If onDestroy() is called as the result of a configuration change, the system immediately creates a new activity instance and then calls onCreate() on that new instance in the new configuration.
- The onDestroy() callback should release all resources that have not yet been released by earlier callbacks such as onStop().

NAVIGATING BETWEEN ACTIVITIES

- An app is likely to enter and exit an activity, perhaps many times, during the app's lifetime. For example, the user may tap the device's Back button, or the activity may need to launch a different activity, i.e., when an app needs to move from the current screen to a new one.
- Depending on whether your activity wants a result back from the new activity it's about to start, you start the new activity using either the startActivity() or the startActivityForResult() method. In either case, you pass in an **Intent** object.

COORDINATING ACTIVITIES

- When one activity starts another, they both experience lifecycle transitions.
- The first activity stops operating and enters the Paused or Stopped state, while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.
- The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process (app) and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:
    1. Activity A's onPause() method executes.
    2. Activity B's onCreate(), onStart(), and onResume() methods execute in sequence. (Activity B now has user focus.)
    3. Then, if Activity A is no longer visible on screen, its onStop() method executes.
- This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another.

INTENT OBJECT

- The Intent object specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application).
- An Intent object can also carry small amounts of data to be used by the activity that is started.
  - If the newly started activity does not need to return a result, the current activity can start it by calling the startActivity() method.
  - When working within your own application, you often need to simply launch a known activity

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

INTENTS

- An Intent is a messaging object you can use to request an action from another app component.
- Although intents facilitate communication between components in several ways, there are three fundamental use cases:
  - Starting an **activity**, which represents a single screen in an app
  - Starting a **service**, which is a component that performs operations in the background without a user interface
    - To ensure that your app is secure, always use an explicit intent when starting a Service and do not declare intent filters for your services
  - Delivering a **broadcast**, which is a message that any app can receive

INTENT – STARTING AN ACTIVITY

- An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to **startActivity()**. The Intent describes the activity to start and carries any necessary data.

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

- If you want to receive a result from the activity when it finishes, call **startActivityForResult().** Your activity receives the result as a separate Intent object in your activity's onActivityResult() callback; for example,

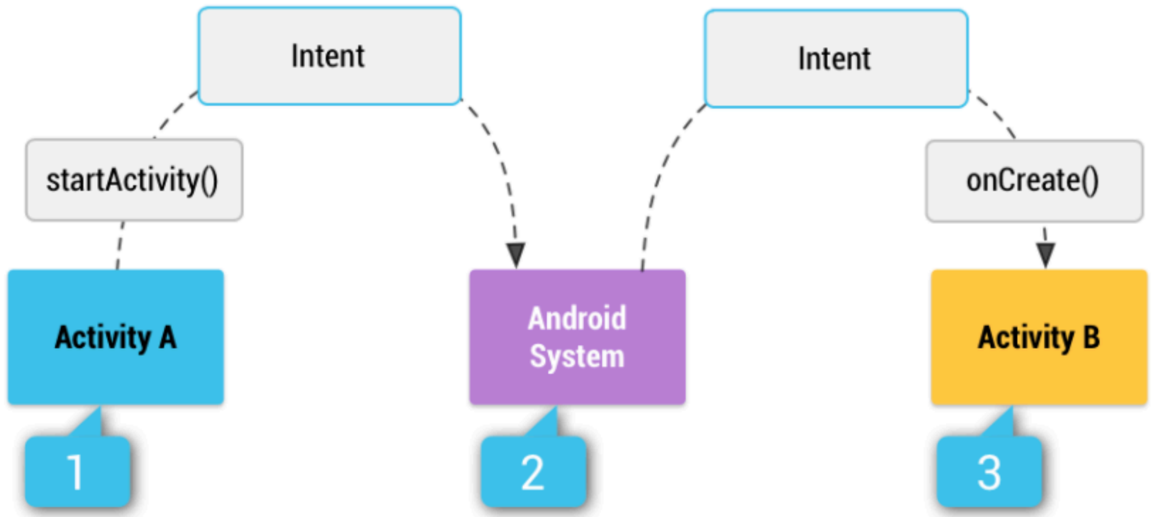
```
startActivityForResult(intent, TEXT_REQUEST);
```

INTENT TYPES

- There are two types of intents:
  - **Explicit intents** specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name. You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start.
  - **Implicit intents** do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.
    - For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.
    - Android system finds the appropriate component to start by comparing the contents of the intent to the **intent filters declared in the manifest file of other apps** on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

IMPLICIT INTENT

1. Activity A creates an Intent with an action description and passes it to startActivity().
2. The Android System searches all apps for an **intent filter** that matches the intent.
3. When a match is found, the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.



INTENT FILTER

- An intent filter is **an expression in an app's manifest file** that specifies the type of intents that the component would like to receive.
- For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent. Likewise, if you do not declare any intent filters for an activity, then it can be started only with an explicit intent

BUILDING AN INTENT

- An Intent object carries information that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), plus information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon).
- The primary information contained in an Intent is
  - Component name –a fully qualified class (package) name;
    - Without a component name, the intent is implicit
  - Action –a string that specifies the generic action to perform, (ACTION_VIEW, ACTION_SEND)
  - Data –the URI (a Uri object) that references the data to be acted on and/or the MIME type of that data.
  - Category –a string containing additional information about the kind of component that should handle the intent. (CATEGORY_BROWSABLE, CATEGORY_LAUNCHER, CATEGORY_APP_MAPS)
  - Extra –key-value pairs that carry additional information required to accomplish the requested action;

EXAMPLE – EXPLICIT INENT

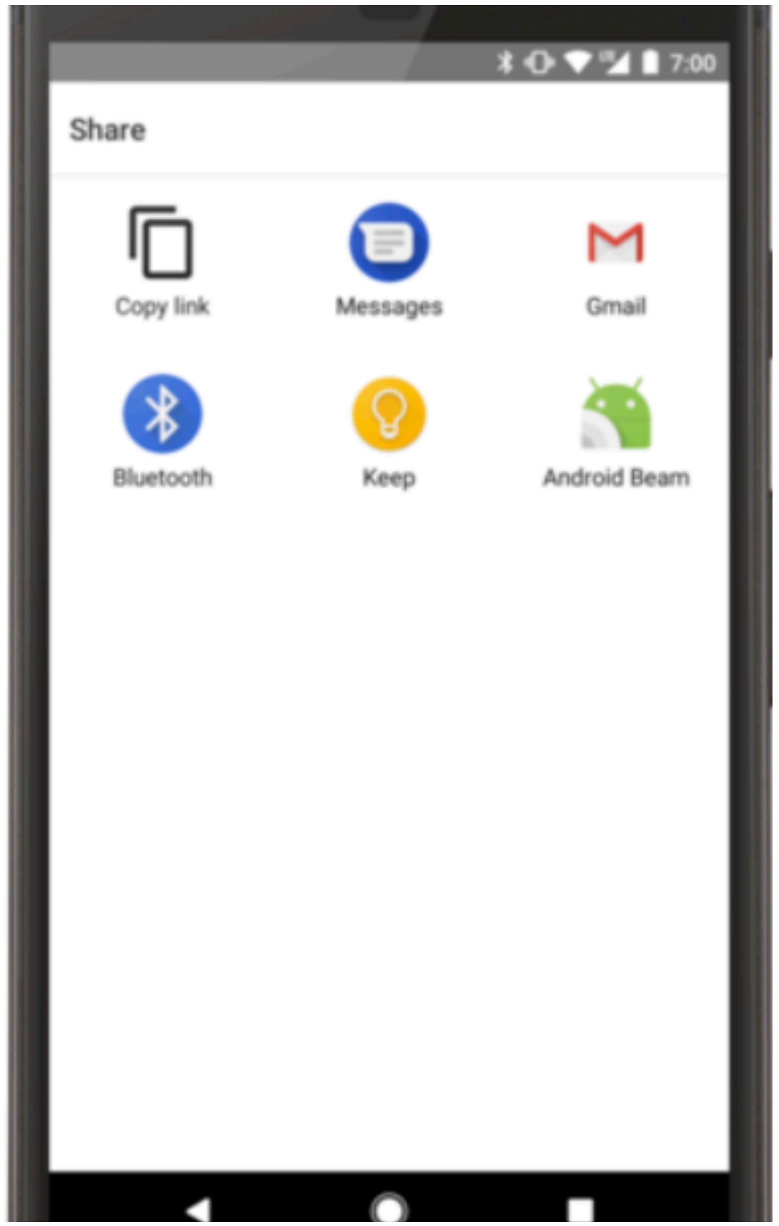
- You built a service in your app, named DownloadService, designed to download a file from the web, you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context
// The fileUrl is a string URL, such as "http://www.example.com/image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

EXAMPLE – IMPLICIT INTENT

- Is useful when your app cannot perform the action, but other apps probably can and you'd like the user to pick which app to use.
  - For example, if you have content that you want the user to share with other people, create an intent with the ACTION_SEND action and add extras that specify the content to share
- When startActivity() is called, the system examines all of the installed apps to determine which ones can handle this kind of intent; If multiple activities accept the intent, the system displays a dialog, as shown on the right, the user can pick which app to use.

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```



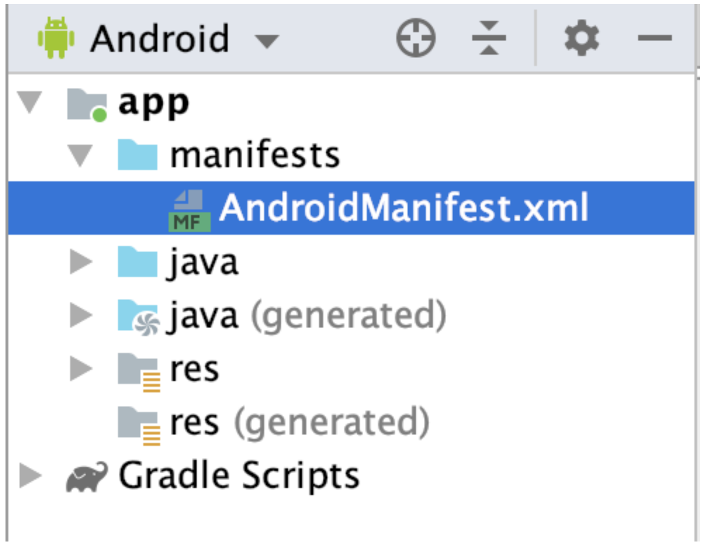
RECEIVING AN IMPLICIT INTENT

- To advertise which implicit intents your app can receive, declare one or more intent filters for each of your app components with an <intent-filter> element in your **manifest file**. Each intent filter specifies the type of intents it accepts based on the intent's action, data, and category.
- An app component should declare separate filters for each unique job it can do.
  - For example, one activity in an image gallery app may have two filters: one filter to view an image, and another filter to edit an image.

EXAMPLE FILTERS DEFINED IN MANIFEST FILE

```
<activity android:name "MainActivity">
    <!-- This activity is the main entry, should appear in app launcher -->
    <intent-filter>
        <action android:name "android.intent.action.MAIN" />
        <category android:name "android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity android:name "ShareActivity">
    <!-- This activity handles "SEND" actions with text data -->
    <intent-filter>
        <action android:name "android.intent.action.SEND"/>
        <category android:name "android.intent.category.DEFAULT"/>
        <data android:mimeType "text/plain"/>
    </intent-filter>
    <!-- This activity also handles "SEND" and "SEND_MULTIPLE" with media data -->
    <intent-filter>
        <action android:name "android.intent.action.SEND"/>
        <action android:name "android.intent.action.SEND_MULTIPLE"/>
        <category android:name "android.intent.category.DEFAULT"/>
        <data android:mimeType "application/vnd.google.panorama360+jpg"/>
        <data android:mimeType "image/*"/>
        <data android:mimeType "video/*"/>
    </intent-filter>
</activity>
```

THE MANIFEST FILE

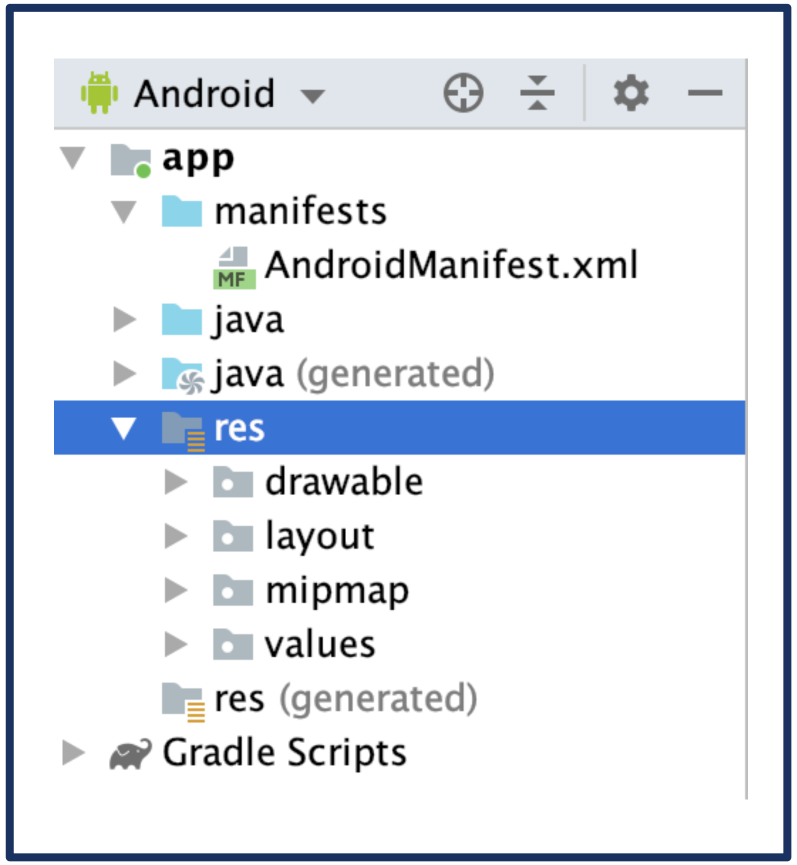


- Before the Android system can start an app component, the system must know that the component exists by reading the app's manifest file, **AndroidManifest.xml.**
- **The primary task of the manifest is to inform the system about the app's components**
- Your app must declare all its components in this file, which must be at the root of the app project directory.
- The manifest does a number of things in addition to declaring the app's components, such as the following:
  - Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
  - Declares the minimum API Level required by the app, based on which APIs the app uses.
  - Declares hardware and software features used or required by the app, such as a camera, bluetooth services, or a multitouch screen.
  - Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.

APP RESOURCES

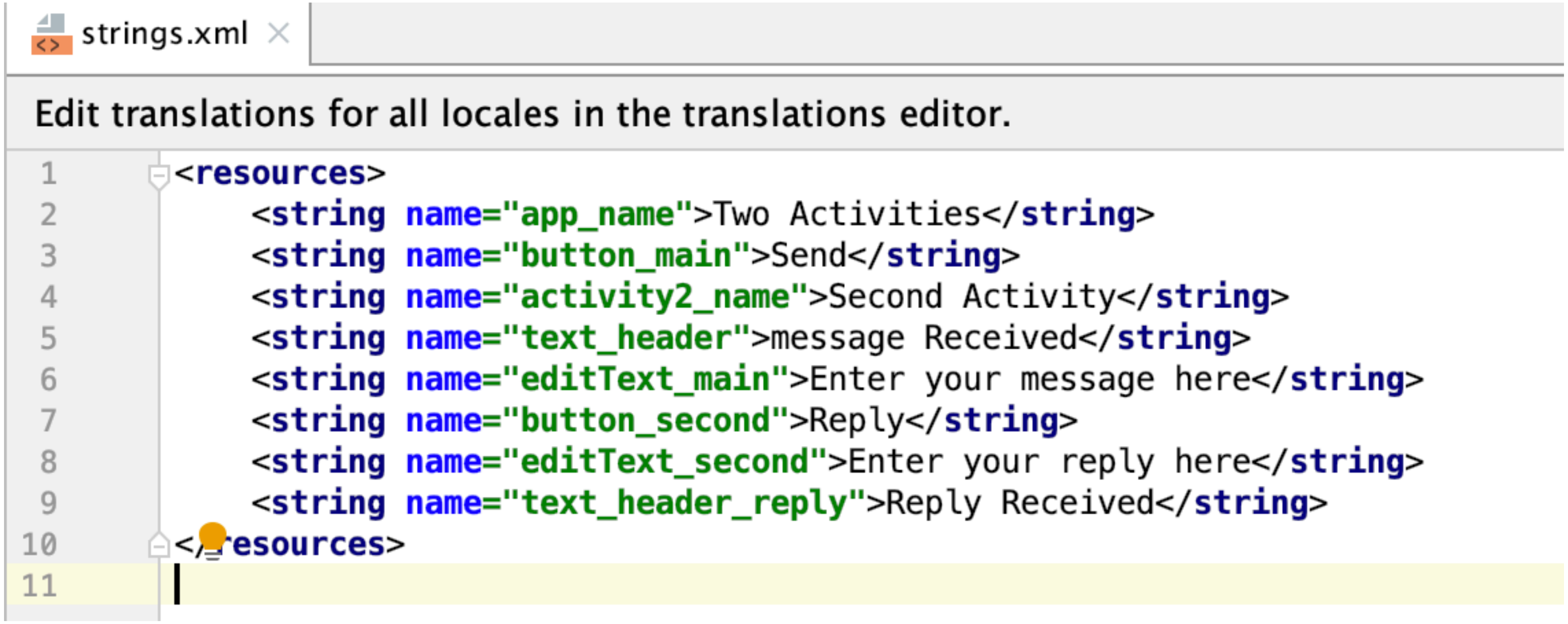
- Android requires resources that are separate from the source code, such as images, audio files, and anything relating to the visual presentation of the app.
  - For example, you can define animations, menus, styles, colors, and the layout of activity user interfaces with XML files.
- Using app resources makes it easy to update various characteristics of your app without modifying code. Providing sets of alternative resources enables you to optimize your app for a variety of device configurations, such as different languages and screen sizes.
- For every resource that you include in your Android project, the SDK build tools define a unique integer ID, which you can use to reference the resource from your app code or from other resources defined in XML.
  - For example, if your app contains an image file named logo.png (saved in the **res/drawable**), the SDK tools generate a resource ID named R.drawable.logo. This ID maps to an app-specific integer, which you can use to reference the image and insert it in your user interface.

GROUPING RESOURCE TYPE



- A drawable resource is a general concept for a graphic that can be drawn to the screen and which you can retrieve with APIs such as getDrawable(int) or apply to another XML resource with attributes such as android:drawable and android:icon
- A layout resource defines the architecture for the UI in an Activity or a component of a UI.
- mipmap resource contains drawable files for different launcher icon densities.
- values resoure includes XML files that contain simple values, such as strings, integers, and colors.

EXAMPLE – A TYPICAL STRING.XML FILE IN RES/VALUES



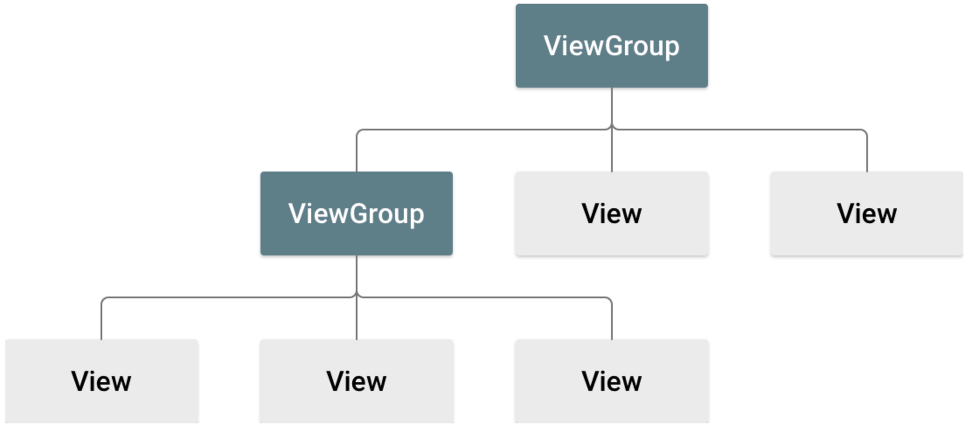
Edit translations for all locales in the translations editor.

```xml
<resources>
    <string name="app_name">Two Activities</string>
    <string name="button_main">Send</string>
    <string name="activity2_name">Second Activity</string>
    <string name="text_header">message Received</string>
    <string name="editText_main">Enter your message here</string>
    <string name="button_second">Reply</string>
    <string name="editText_second">Enter your reply here</string>
    <string name="text_header_reply">Reply Received</string>
</resources>
```

USER INTERFACE & NAVIGATION

- Your app's user interface is everything that the user can see and interact with.
- Android provides a variety of pre-built UI components such as structured layout objects and UI controls that allow you to build the graphical user interface for your app.
- Android also provides other UI modules for special interfaces such as dialogs, notifications, and menus

LAYOUT

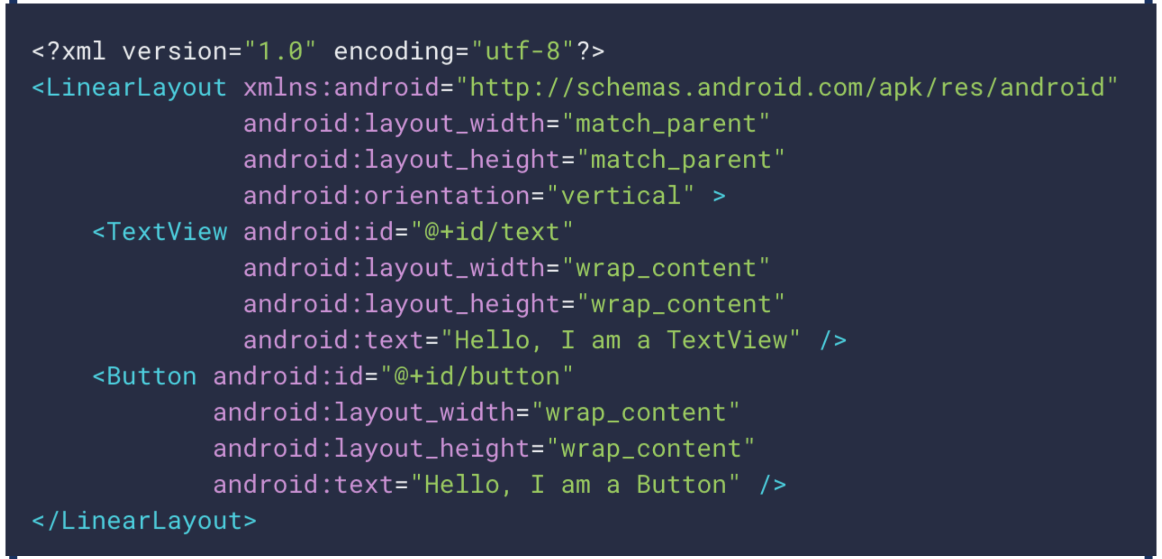


- A layout defines the structure for a user interface in your app, such as in an activity.
- All elements in the layout are built using a hierarchy of **View** and **ViewGroup** objects.
- A View usually draws something the user can see and interact with.
- A ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects

LAYOUT

- You can declare a layout in two ways:
  - Declare UI elements in XML. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts. You can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.
  - Instantiate layout elements at runtime. Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.
- Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations

EXAMPLE – LAYOUT



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

- Each layout file must contain exactly one root element, which must be a View or ViewGroup object.
- Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout.
- For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

LOADING THE LAYOUT

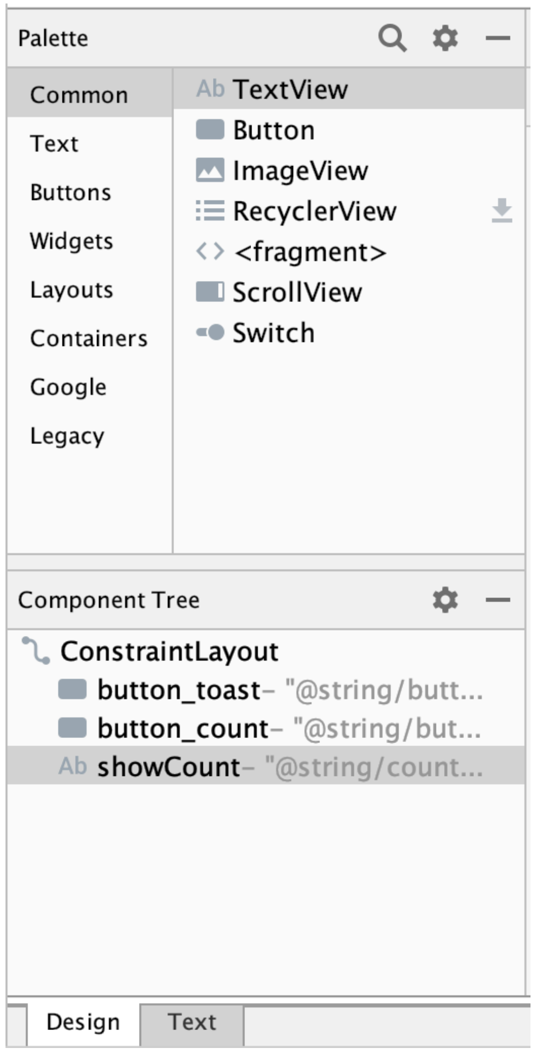
- When you compile your app, each XML layout file is compiled into a **View** resource.
- You should load the layout resource from your app code, in your Activity.onCreate() callback implementation.
- Do so by calling setContentView(), passing it the reference to your layout resource in the form of: **`R.layout.layout_file_name`**. For example, if your XML layout is saved as main_layout.xml, you would load it for your Activity like

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

VIEWS

- The UI consists of a hierarchy of objects called views — every element of the screen is a View.
- The View class represents the basic building block for all UI components, and the base class for classes that provide interactive UI components such as buttons, checkboxes, and text entry fields.
- A View has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the density-independent pixel (dp).
- Every View and ViewGroup object supports their own variety of XML attributes.
- Any View object may have an integer ID associated with it, to uniquely identify the View within the tree.

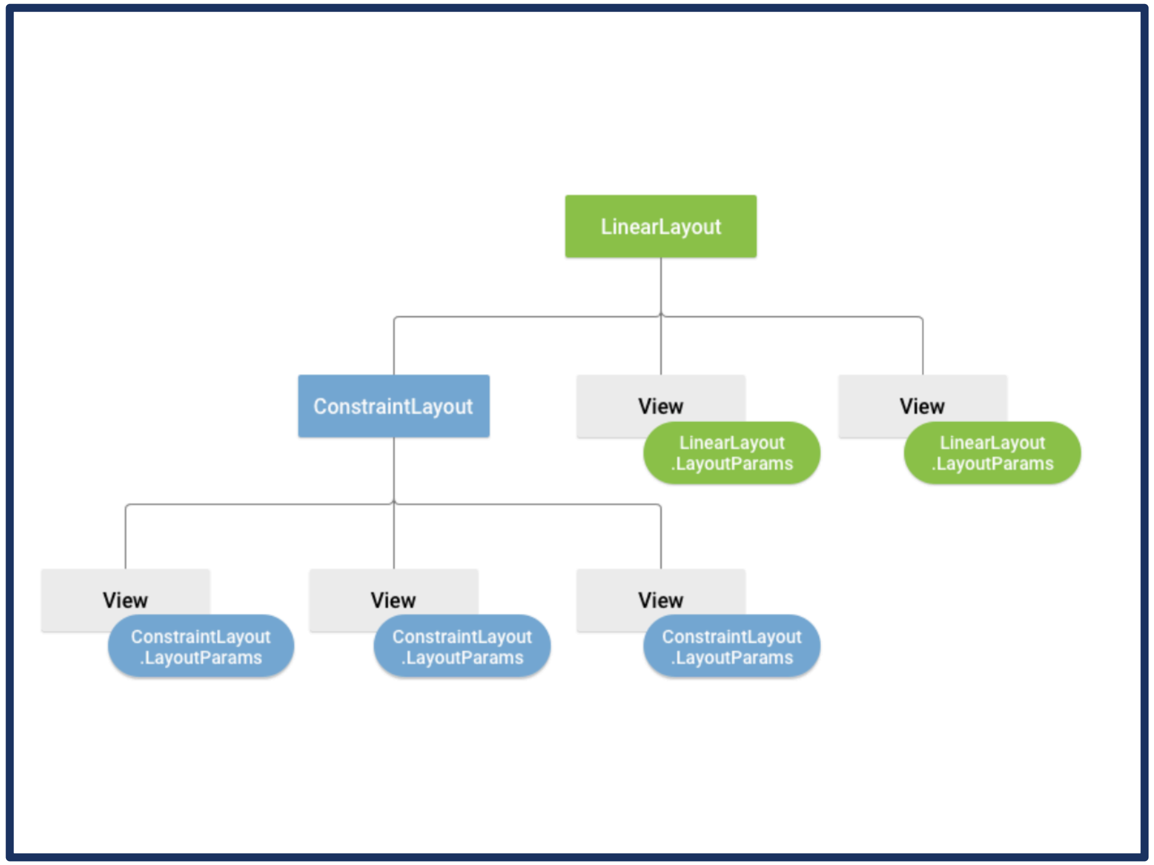
```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```



ANDROID COMMON VIEWS

- The Android system provides hundreds of predefined View subclasses. Commonly used View subclasses described over several lessons include:
  - **TextView** for displaying text
  - **EditText** to enable the user to enter and edit text
  - **Button** and other clickable elements (such as RadioButton, CheckBox, and Spinner) to provide interactive behavior
  - **ScrollView** and RecyclerView to display scrollable items
  - **ImageView** for displaying images
  - **ConstraintLayout** and **LinearLayout** for containing other views and positioning them

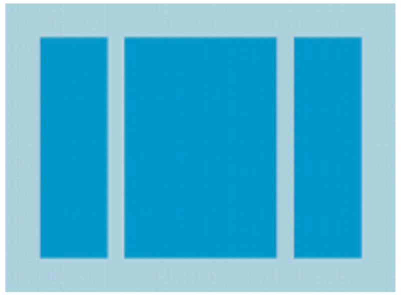
LAYOUT PARAMETERS


LinearLayout
ConstraintLayout
View
LinearLayout .LayoutParams
View
LinearLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams

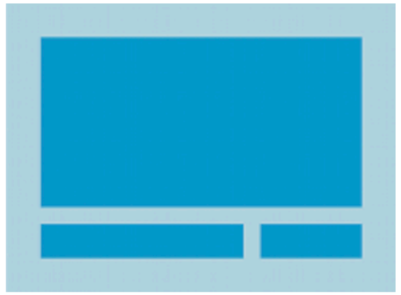
- XML layout attributes named layout_something define layout parameters for the View that are appropriate for the ViewGroup in which it resides.
- Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define the size and position for each child view, as appropriate for the view group.

COMMON LAYOUTS

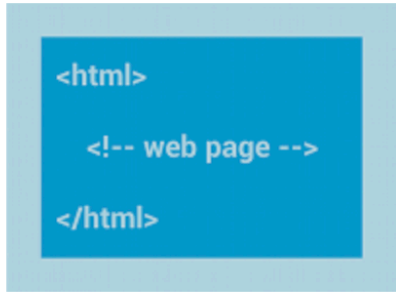
Linear Layout



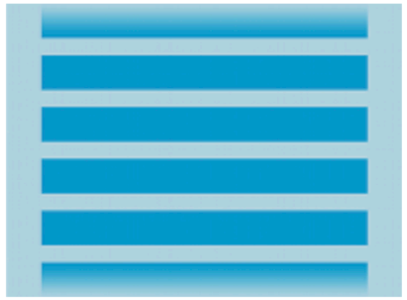
Relative Layout



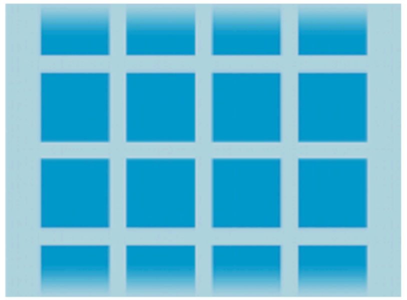
Web View



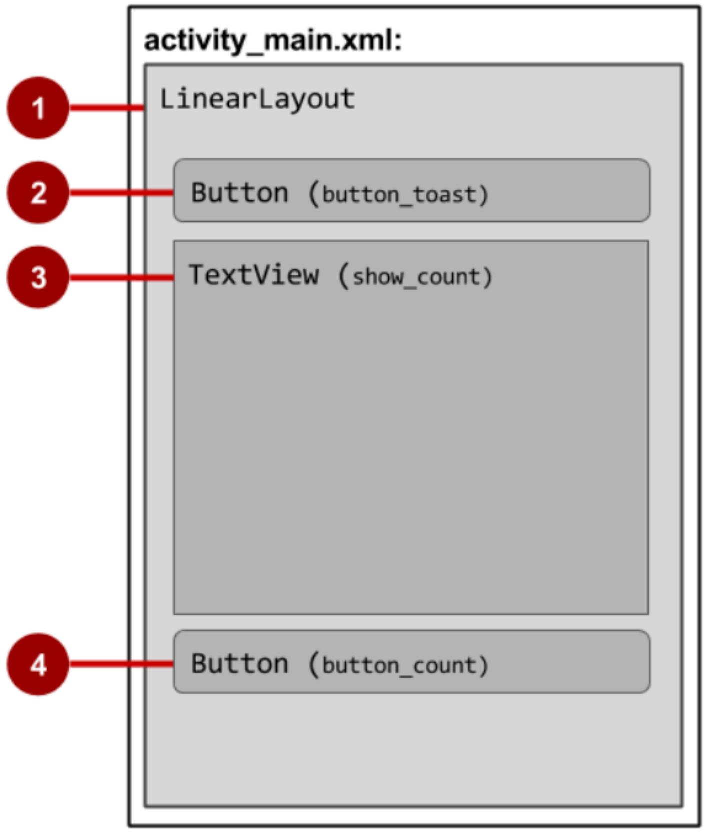
List View

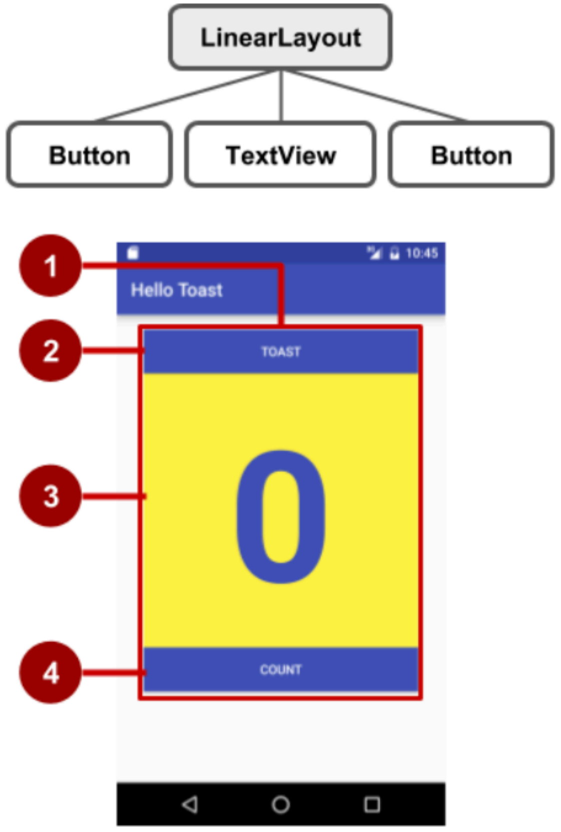


Grid View

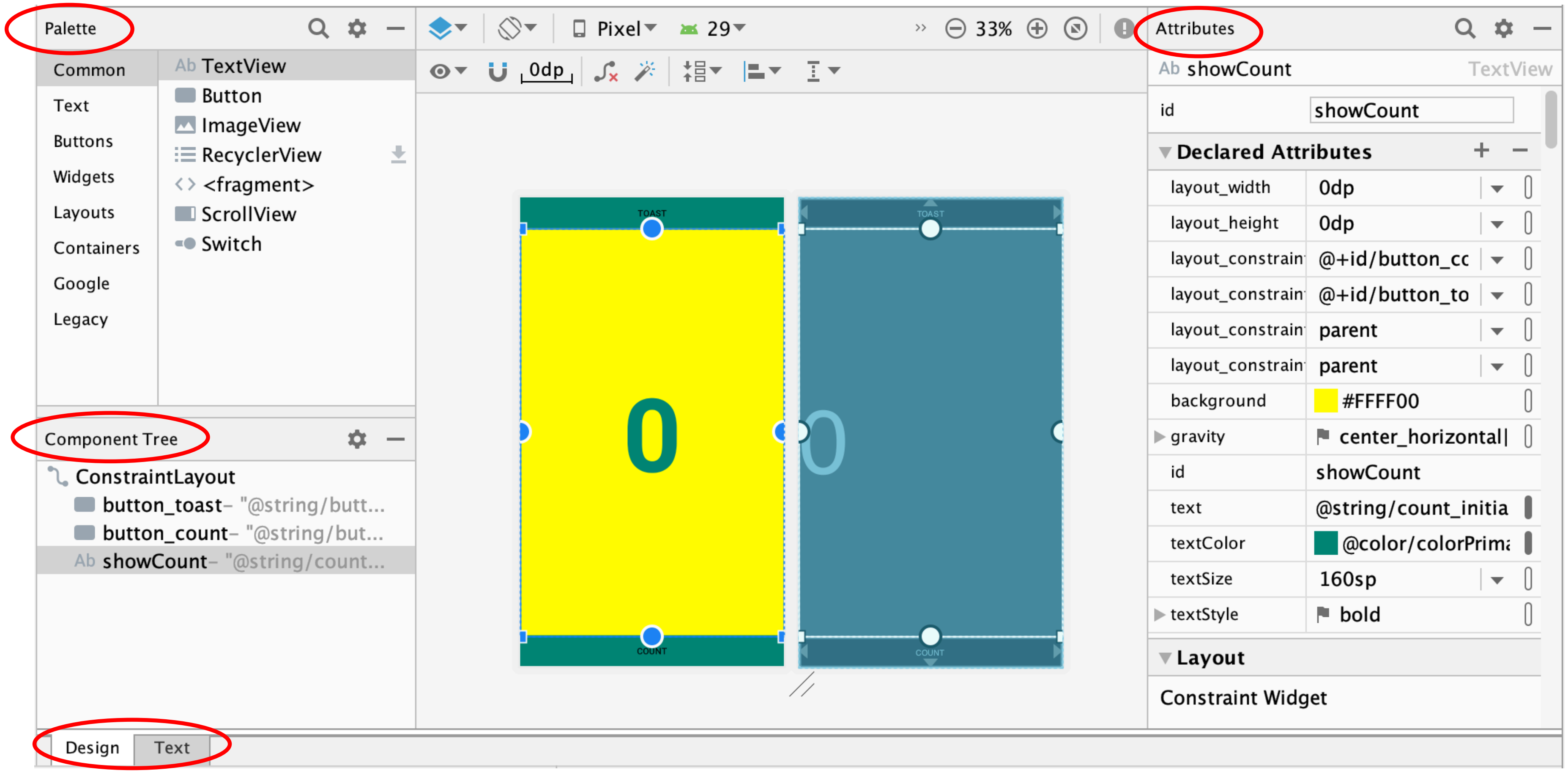


EXAMPLE – A LINEARLAYOUT





THE LAYOUT EDITOR IN ANDROID STUDIO



Component Tree:

```
ConstraintLayout
    button_toast– "@string/butt...
    button_count– "@string/but...
    showCount– "@string/count...
```

Attributes: showCount TextView

id: showCount

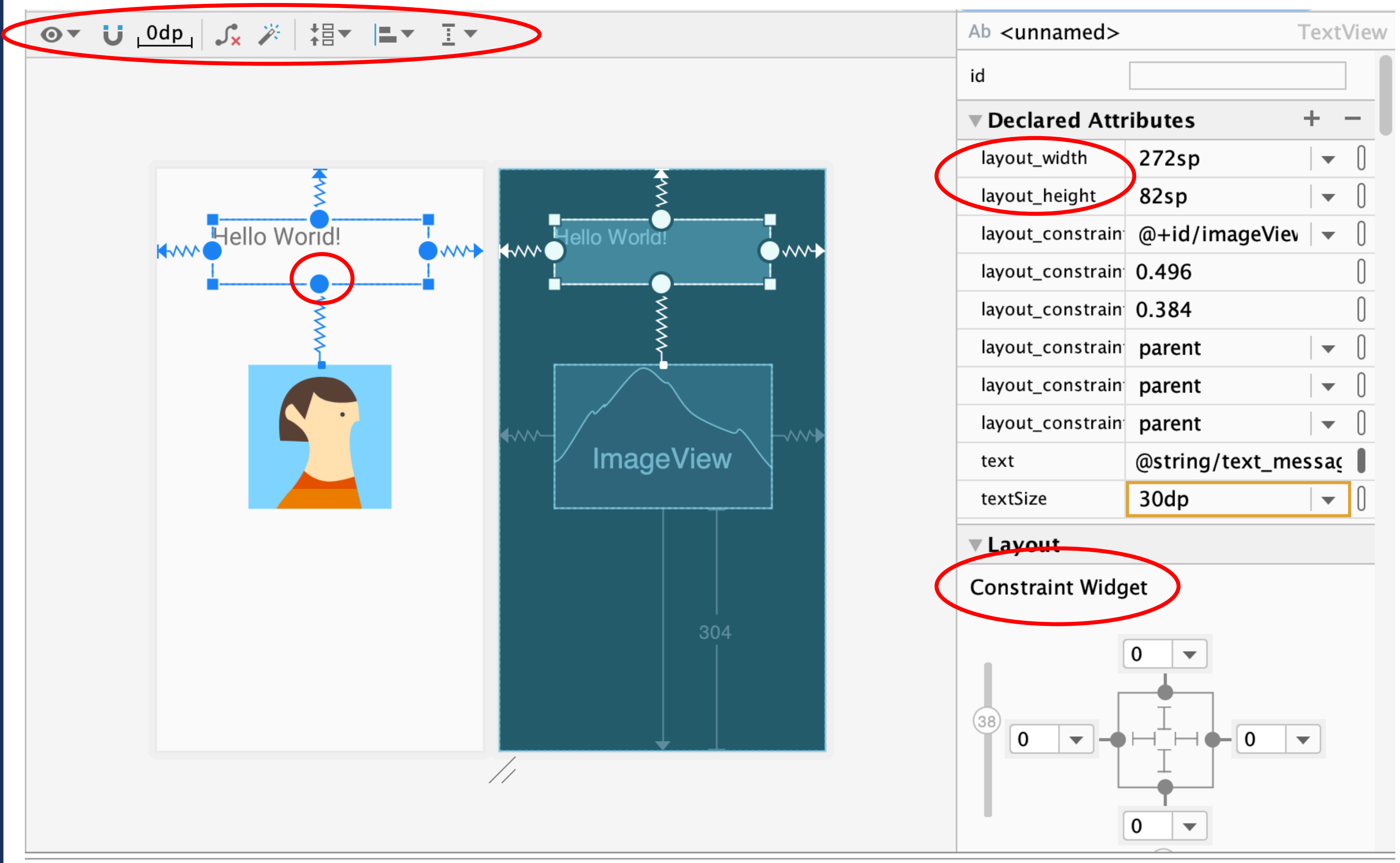
layout_width	0dp
layout_height	0dp
layout_constrain	@+id/button_cc
layout_constrain	@+id/button_to
layout_constrain	parent
layout_constrain	parent
background	🟨 #FFFF00
gravity	center_horizontal|
id	showCount
text	@string/count_initia
textColor	🟩 @color/colorPrim
textSize	160sp
textStyle	bold

CONSTRAINT LAYOUT

- 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.
- All the power of ConstraintLayout is available directly from the Layout Editor's visual tools, because the layout API and the Layout Editor were specially built for each other. So you can build your layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML.

CONSTRAINT LAYOUT IN LAYOUT EDITOR

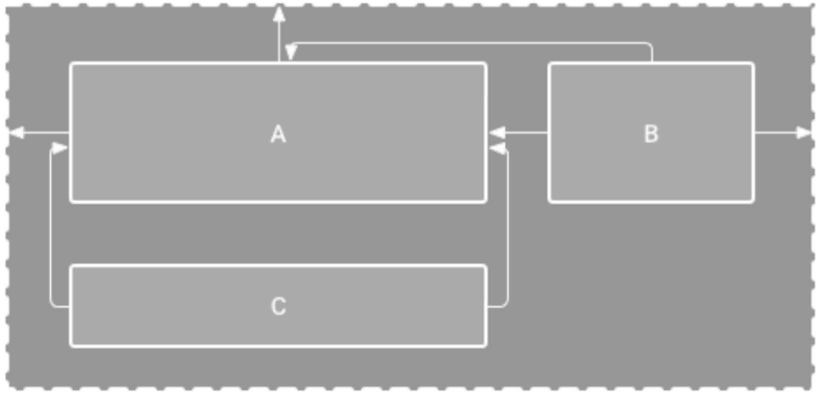
- layout_width and layout_height
  - match_constraint-as big as its parent (minus padding)
  - wrap_content-just big enough to enclose its content (plus padding).
  - exact values

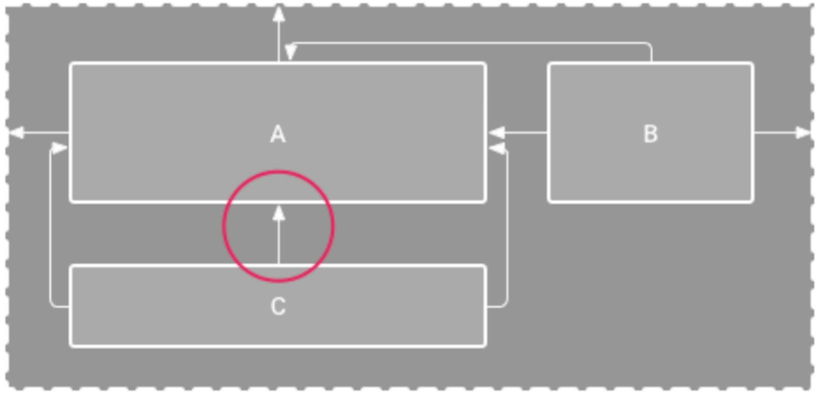


layout_width	272sp
layout_height	82sp
layout_constrain	@+id/imageViev
layout_constrain	0.496
layout_constrain	0.384
layout_constrain	parent
layout_constrain	parent
layout_constrain	parent
text	@string/text_messag
textSize	30dp

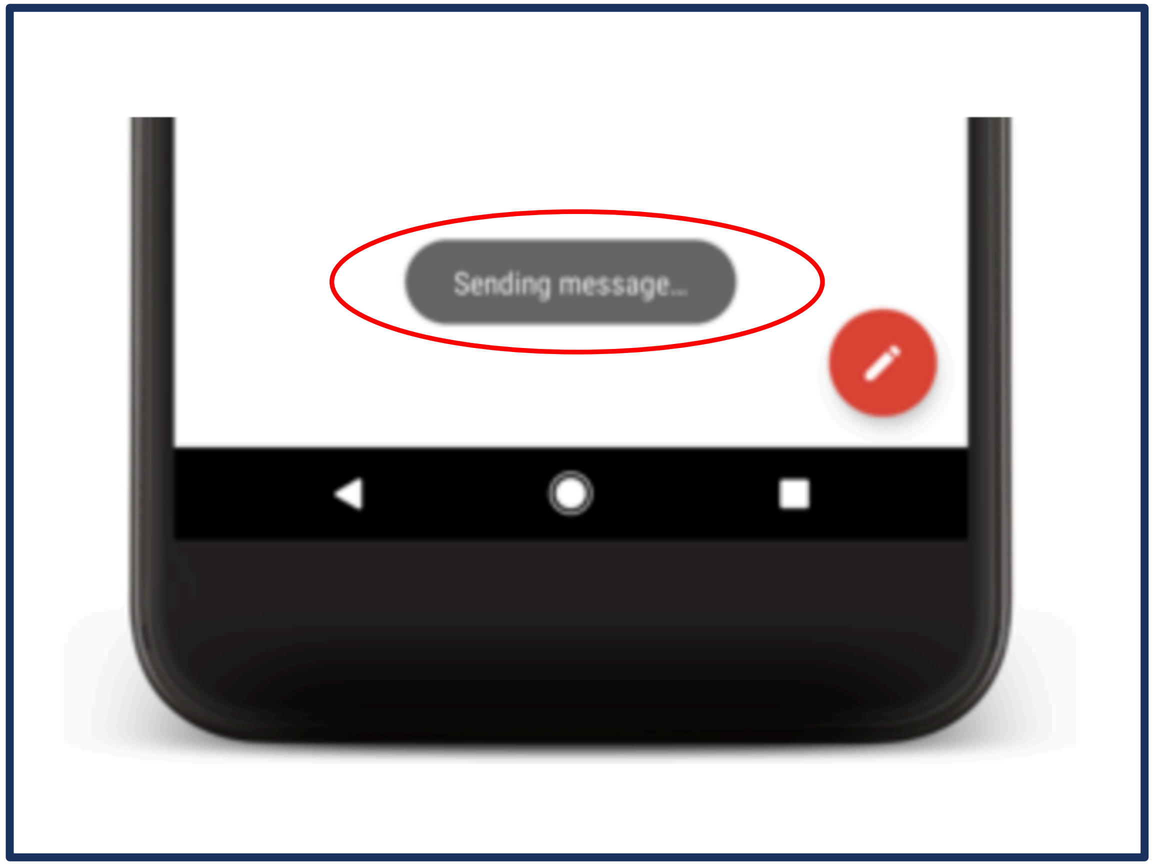
CONSTRAINT LAYOUT – EXAMPLE

- The layout on the left looks good in the editor, but there's no vertical constraint on view C. When this layout draws on a device, view C horizontally aligns with the left and right edges of view A, but appears at the top of the screen because it has no vertical constraint.
- With the layout on the right, View C is now vertically constrained below view A





TOAST



- A toast provides simple feedback about an operation in **a small popup**.
- It only fills the amount of space required for the message and the current activity remains visible and interactive.
- Toasts automatically disappear after a timeout.
- For example, clicking Send on an email triggers a "Sending message..." toast, as shown in the following screen capture
- **Toasts are not clickable**. If user response to a status message is required, use a Notification.

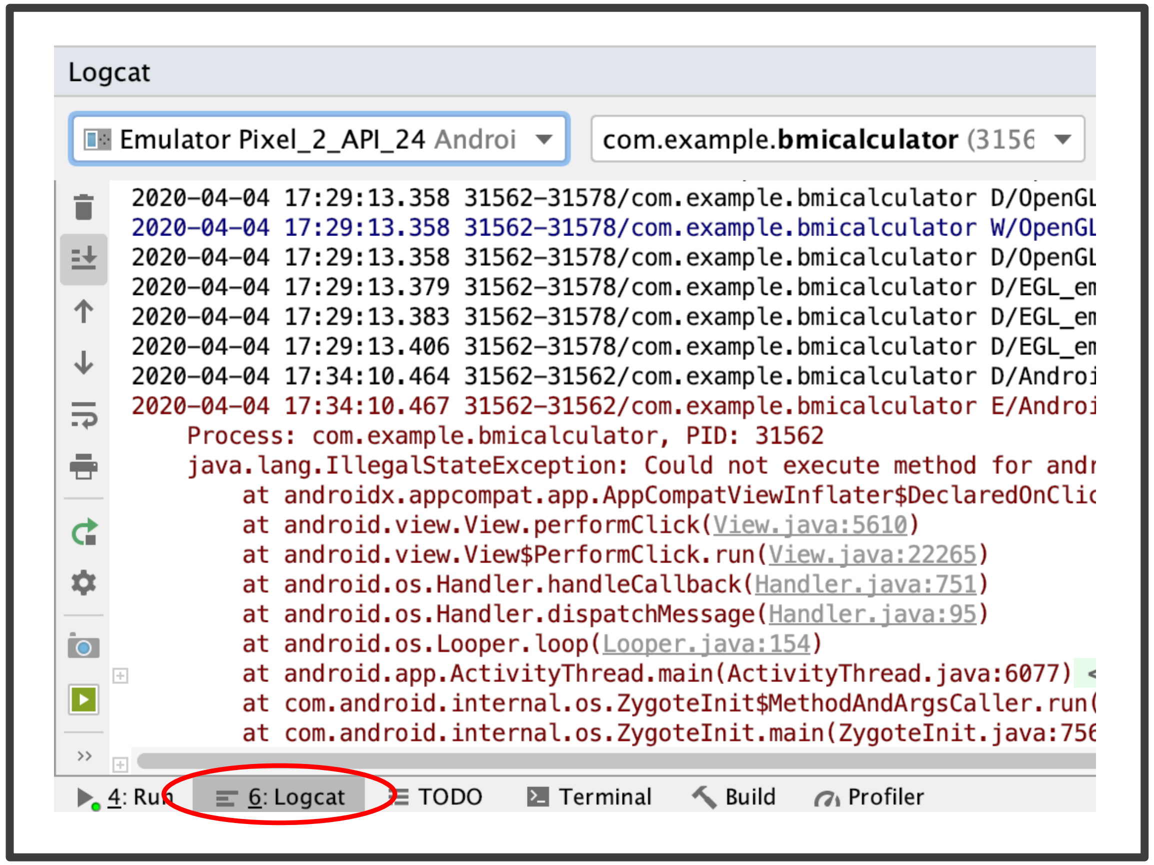
TOAST – EXMAPLE

- First, instantiate a Toast object with one of the makeText() methods. This method takes three parameters: the application Context, the text message, and the duration for the toast. It returns a properly initialized Toast object. You can display the toast notification with show()
- This example demonstrates everything you need for most toast notifications. You should rarely need anything else.

```
Context context = getApplicationContext();
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

```
Toast.makeText(context, text, duration).show();
```

VIEW LOGS WITH LOGCAT


Logcat
Emulator Pixel_2_API_24 Androi
com.example.bmicalculator (3156
4: Run
6: Logcat
TODO
Terminal
Build
Profiler

```
2020-04-04 17:29:13.358 31562-31578/com.example.bmicalculator D/OpenGL
2020-04-04 17:29:13.358 31562-31578/com.example.bmicalculator W/OpenGL
2020-04-04 17:29:13.358 31562-31578/com.example.bmicalculator D/OpenGL
2020-04-04 17:29:13.379 31562-31578/com.example.bmicalculator D/EGL_em
2020-04-04 17:29:13.383 31562-31578/com.example.bmicalculator D/EGL_em
2020-04-04 17:29:13.406 31562-31578/com.example.bmicalculator D/EGL_em
2020-04-04 17:34:10.464 31562-31562/com.example.bmicalculator D/Androi
2020-04-04 17:34:10.467 31562-31562/com.example.bmicalculator E/Androi
    Process: com.example.bmicalculator, PID: 31562
    java.lang.IllegalStateException: Could not execute method for andr
        at androidx.appcompat.app.AppCompatViewInflater$DeclaredOnClic
        at android.view.View.performClick(View.java:5610)
        at android.view.View$PerformClick.run(View.java:22265)
        at android.os.Handler.handleCallback(Handler.java:751)
        at android.os.Handler.dispatchMessage(Handler.java:95)
        at android.os.Looper.loop(Looper.java:154)
        at android.app.ActivityThread.main(ActivityThread.java:6077) <
        at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:756
```

- The Logcat window in Android Studio displays system messages, such as when a garbage collection occurs, and messages that you added to your app with the **Log class.**
- It displays messages in real time and keeps a history so you can view older messages.
- When an app throws an exception, logcat shows a message followed by the associated stack trace containing links to the line of code.

WRITE LOG MESSAGES

- Log.e(String, String) (error)
- Log.w(String, String) (warning)
- Log.i(String, String) (information)
- Log.d(String, String) (debug)
- Log.v(String, String) (verbose)

https://developer.android.com/reference/android/util/Log#v(java.lang.String,%20java.lang.String)

BUNDLE

- A Bundle is a collection of data, stored as key/value pairs. To pass information from one activity to another, you put keys and values into the intent extra Bundle from the sending activity, and then get them back out again in the receiving activity.

```
//in MainActivity
public static final String EXTRA_MESSAGE =
        "com.example.android.twoactivities.extra.MESSAGE";
private EditText mMessageEditText;
...
    protected void onCreate(Bundle savedInstanceState) {
        ...
        mMessageEditText = findViewById(R.id.editText);
    }
    public void launch2ndActivity(View view) {
        ...
        Intent intent = new Intent(this, SecondActivity.class);
        String message = mMessageEditText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
```

BUNDLE – RECEIVING THE MESSAGE IN THE 2ND ACTIVITY

```
//in SecondActivity
protected void onCreate(Bundle savedInstanceState) {
    ...
    Intent intent = getIntent();
    String message =
        intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
    TextView textView = findViewById(R.id.text_message);
    textView.setText(message);

     ...
}
```

SUMMARY

- Versions of Android have a version number, API level, and code name.
- Android Studio is a special version of IntelliJ IDEA that interfaces with the Android Software Development Kit (SDK) and the Gradle build system.
- A typical Android app is composed of activities, layouts, and resource files.
- Layouts describe what your app looks like. They're held in the app/src/main/res/layout folder.
- Activities describe what your app does, and how it interacts with the user.
- The activities you write are held in the app/src/main/java folder.
- AndroidManifest.xml contains information about the app itself.
- An AVD is an Android Virtual Device. It runs in the Android emulator and mimics a physical Android device.
- An APK is an Android application package. It's like a JAR file for Android apps, and contains your app's bytecode, libraries, and resources. You install an app on a device by installing the APK.
- Android apps run in separate processes using the Android runtime (ART)

THANK YOU