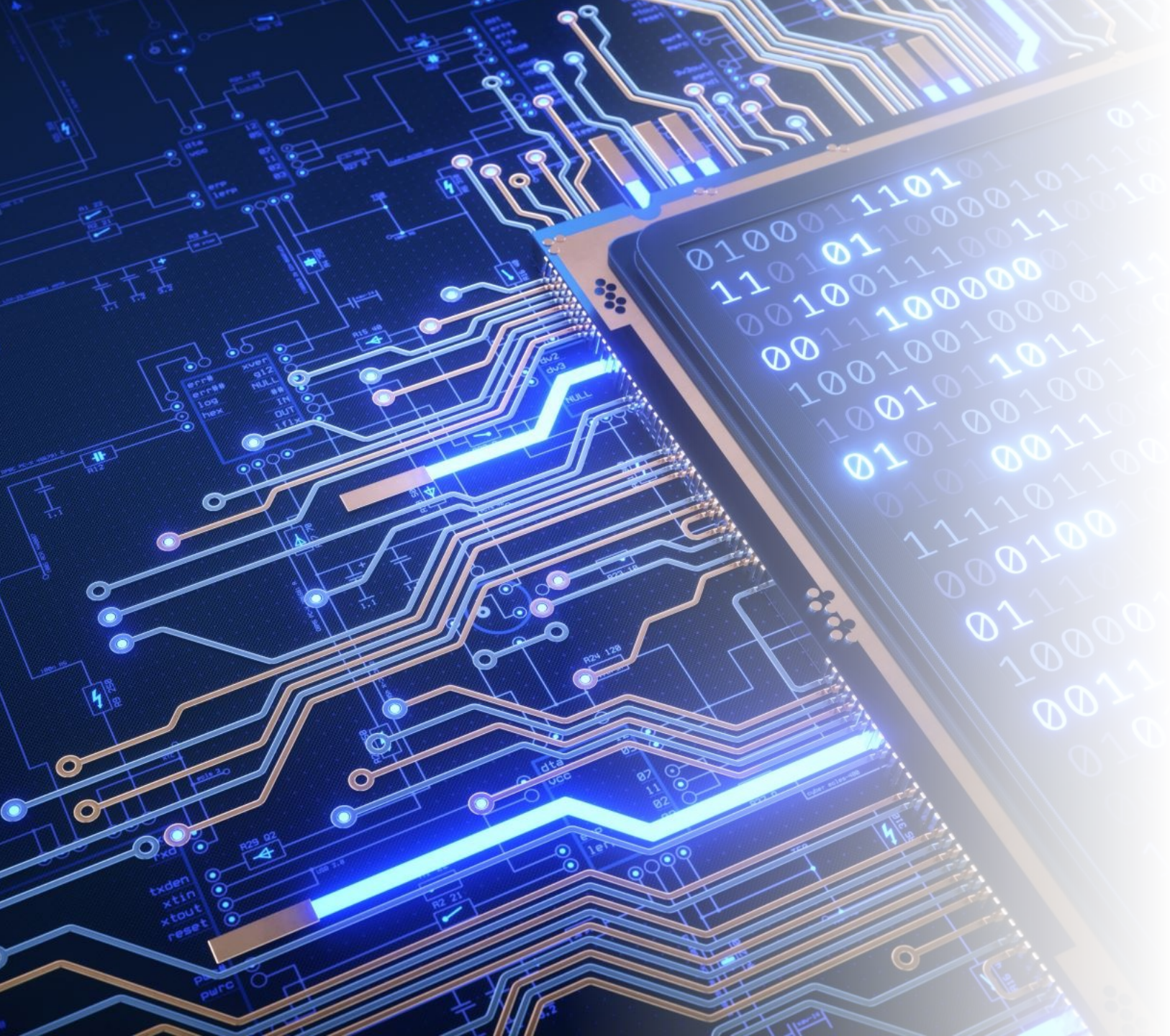


EEP 523: MOBILE APPLICATIONS FOR SENSING AND CONTROL

SUMMER 2021
Lecture 3

Tamara Bonaci

tbonaci@uw.edu

Course Announcements

Today

1. Activity Life Cycle

2. Persistent User Interface

3. Implicit Intents
   a) Take Pictures
   b) Text-to-Speech / Speech-to-Text

4. Fragments

Agenda

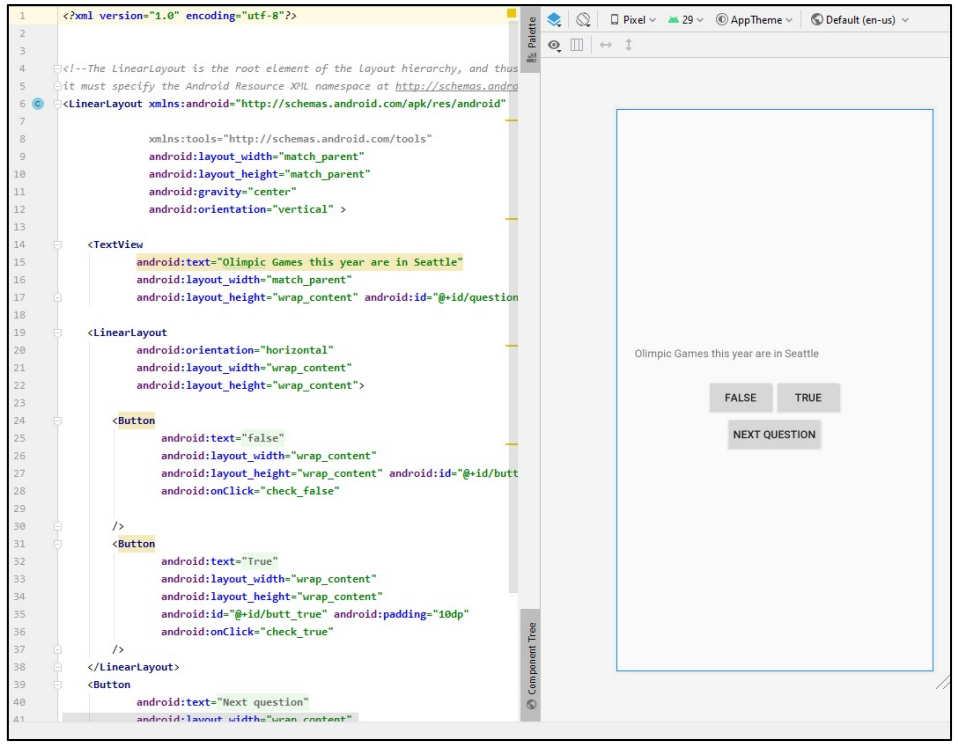
Review: App Basics

The most basic App consists of

an *activity (.kt)* and a *layout (.xml)*

```
class MainActivity : AppCompatActivity() {

    private var mCurrentIndex = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        mCurrentIndex = 1
    }

    private val questionBank = listOf(
        Question("This year the Olympic Games are in Seattle", answer: false),
        Question("First modern Olympic games were in 1796", answer: false),
        Question("This year games there will a total of 40 sports", answer: false),
        Question("The Olympic flag contains 5 colors", answer: true),
        Question("Women were allowed to participate in the Olimpics in 1900", answer: true),
        Question("Skateboarding is an Olympic sport", answer: true),
        Question("Gold medals are made of 50% gold", answer: false),
        Question("Michael Phelps is the athlete with the most Olympic medals", answer: true),
        Question("Men and women do NOT compete against each other in any ...", answer: true),
        Question("Swimming obstacle race was Olympic in one Game", answer: false)
    )


    fun updateQuestion(view: View){
        mCurrentIndex ++
        if (mCurrentIndex > questionBank.size) {
            mCurrentIndex = 1
        }
        val questionTextResID = questionBank[mCurrentIndex].textResId
        questionView.setText(questionTextResID)
    }
```

```
<?xml version="1.0" encoding="utf-8"?>


<!--The LinearLayout is the root element of the layout hierarchy, and thus
it must specify the Android Resource XML namespace at http://schemas.andro
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

              xmlns:tools="http://schemas.android.com/tools"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:gravity="center"
              android:orientation="vertical" >

    <TextView
              android:text="Olimpic Games this year are in Seattle"
              android:layout_width="match_parent"
              android:layout_height="wrap_content" android:id="@+id/question

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

        <Button
                android:text="false"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" android:id="@+id/butt
                android:onClick="check_false"

        />
        <Button
                android:text="True"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:id="@+id/butt_true" android:padding="10dp"
                android:onClick="check_true"
        />
    </LinearLayout>
    <Button
                android:text="Next question"
                android:layout_width="wrap_content"
```



Review: MyActivity.kt

- An **activity** represents a single screen with a User Interface. Basic application startup logic that should happen only once for the entire life of the activity.

- `class MainActivity : AppCompatActivity() {…}` subclass of Android's **Activity** class that provides compatibility support for older versions of Android.

Review: MyActivity.kt

- An **activity** represents a single screen with a User Interface. Basic application startup logic that should happen only once for the entire life of the activity.

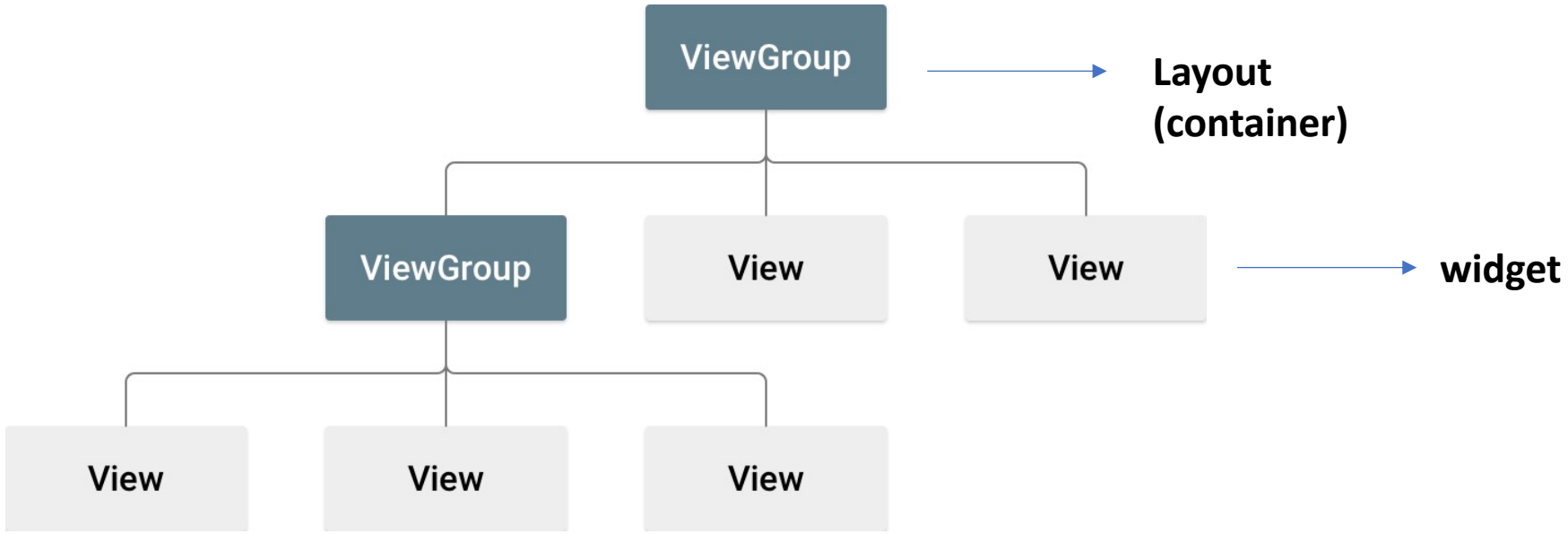
- ***onCreate*() :**
  function to initialize the activity -

Will be covered in Lecture 3
Save data between activity re-initialization

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

→ Set the layout for the activity

Review: Android Graphical User Interface (GUI)



Review: Basic App GUI

1. Define a Layout (**Constraint layout** by default)

2. Add widgets to the Layout and configure them
   a. Drag and drop on editor
   b. Edit **XML** file *`/app/res/layout/activity_main.xml`*

3. Add events to capture user input (press a button)

4. Handle events

Review: Respond to a Button Click (1)

1. **Set *onClick* function in the xml file** */app/res/layout/activity_main.xml*

```
<Button
        android:text="Button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" tools:layout_editor_absoluteY="43dp"
        tools:layout_editor_absoluteX="0dp"
        android:id="@+id/button"
        android:onClick="butt_action"/>
```

+ Implement function in the Kotlin class **file** *app/java/.../MainActivity.kt*

```
fun butt_action( view:View){
    //perform action
}
```

Review: Respond to a Button Click (2)

2. Set *onClick* Listener

```
val btn_click_me = findViewById(R.id.button) as Button
        btn_click_me.setOnClickListener {
            // your code to perform when the user clicks on the button
            butt_action()
        }
```

With Kotlin Android Extensions (View binding)

```
import kotlinx.android.synthetic.main.activity_main.*

val btn_click_me = findViewById(R.id.button) as Button
        button.setOnClickListener {
            // your code to perform when the user clicks on the button
            butt_action()
        }
```

Layouts in Android

EE P 523, Lecture 3

Layouts in Android

You can declare a layout elements in two ways:

1. **Declare UI elements in XML**. (declarative UI)

   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.

Layouts in Android

You can declare a layout in two ways:

1. Declare UI elements in XML

2. **Instantiate layout elements at runtime** (*procedural UI*)

   You can create View and ViewGroup objects (and manipulate their properties) programmatically.

Initiate elements at runtime

e.g.: Draw a button on the center of the screen at runtime

1. Create a new Button variable
2. Define the size, and constraints of the Button. Define where to draw the button on the screen
3. Additional configuration of the button
4. Add listener → event when the user clicks the button
5. Add the button to the main layout

Initiate elements at runtime

Draw a button on the center of the screen at runtime

MainActivity.kt

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

    // 1. Create a new Button variable
    val butt = Button(this)

    //2. Define the size, and constraints of the Button. Define where to draw the button on the screen
    butt.layoutParams = ConstraintLayout.LayoutParams(ConstraintLayout.LayoutParams.WRAP_CONTENT,
                        ConstraintLayout.LayoutParams.WRAP_CONTENT)
    val params = butt.layoutParams as ConstraintLayout.LayoutParams
            params.endToEnd = R.id.main_layout
            params.startToStart = R.id.main_layout
            params.topToTop = R.id.main_layout
            params.bottomToBottom = R.id.main_layout

    //3. Additional configuration of the button
    butt.text = "Click me"
    butt.setBackgroundColor(Color.BLACK)
    butt.setTextColor(Color.WHITE)

    //4. Add listener -- event when the user clicks the button
    butt.setOnClickListener(View.OnClickListener {
        butt.text = "You just clicked me"
    })

    //5. Add the button to the main layout
    main_layout .addView(butt);
}
```

Name of xml file

Equivalent of "parent" in .xml

id of of Layout (Constraint Layout) defined in xml file

Initiate elements at runtime

Draw a button on the center of the screen at runtime

Activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout

        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity"
        android:id="@+id/main_layout">
    <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="I'm a button!"
            android:id="@+id/butt"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
           app:layout_constraintBottom_toBottomOf="parent"  />

</android.support.constraint.ConstraintLayout>
```

Initiate elements at runtime

What would be a different strategy to displaying a button at runtime?

-> Create the button on the xml, and set it to be invisible.
-> then make it visible in the kotlin file when needed.

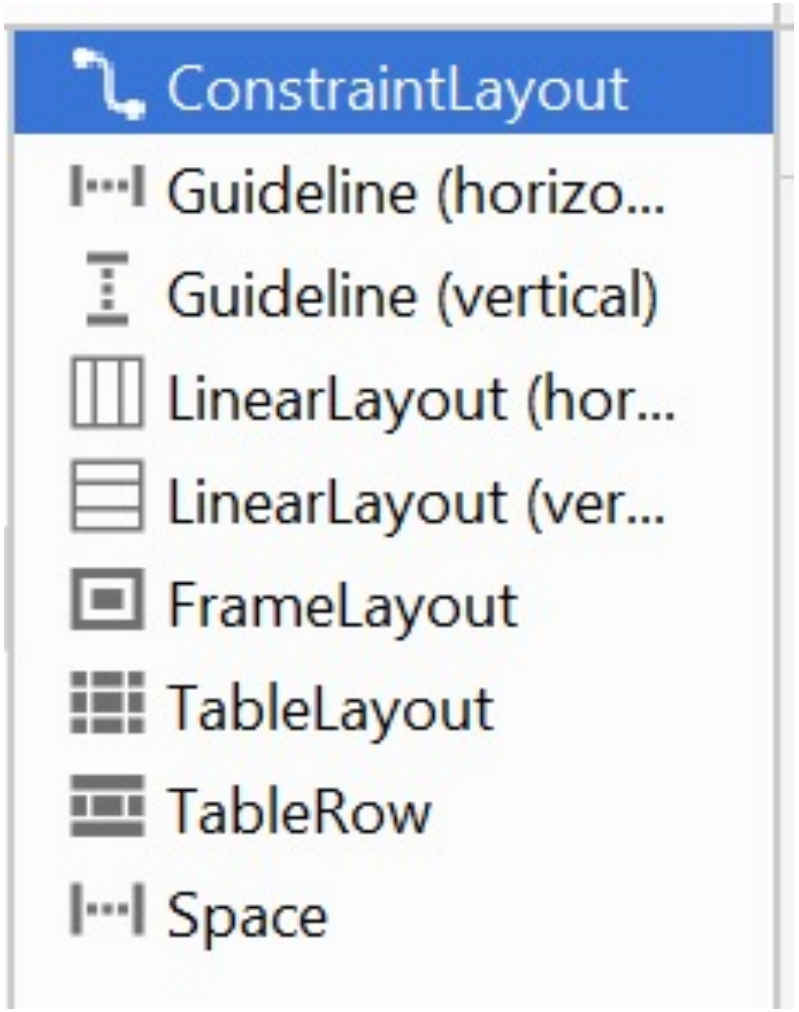
In xml file

```
android:visibility="gone"
```

In the kotlin Class:

```
butty.setVisibility(View.VISIBLE)
```

Common Layouts



Common Layouts

Linear Layout

Organizes its children into **a single horizontal or vertical row**

Scrollbar if length of windows exceeds length of the screen

Relative Layout

Location of child objects **relative to each other or to the parent**

Web View

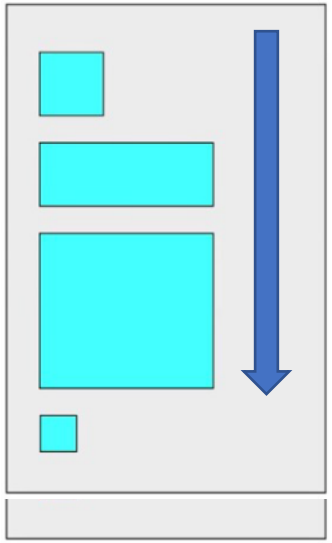
Displays **web pages**

Linear Layout

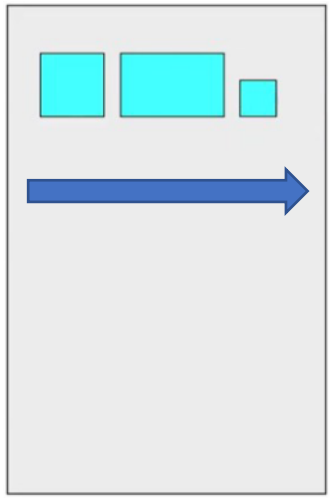
ViewGroup (*container*) that aligns all children

in a single direction

Vertically



Horizontally



All children are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding).

Linear Layout: Gravity

Alignment direction that widgets are pulled

– top, bottom, left, right, center

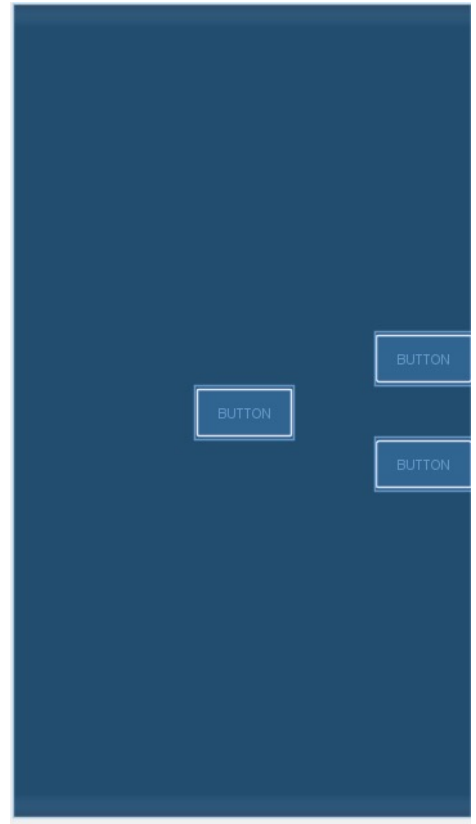
– combine multiple with |

– *gravity* vs *layout_gravity*

In the xml file:

- *android:gravity*
  set gravity on the layout (container) to adjust all widgets (eg. Buttons)

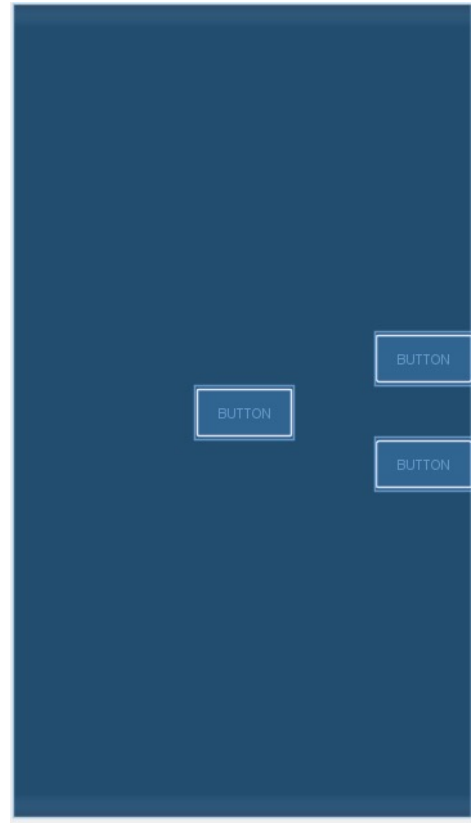
- *android:layout_gravity:*
  adjust individual widgets within the layout



Linear Layout: Gravity

Alignment direction that widgets are pulled

```
<?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"
              android:gravity="center|right">
    <Button
            android:text="Button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
             android:id="@+id/v_1"     />
    <Button
            android:text="Button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
      android:layout_gravity="center"
             android:id="@+id/v_2“     />

    <Button
            android:text="Button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
             android:id="@+id/v_3"/>

</LinearLayout>
```

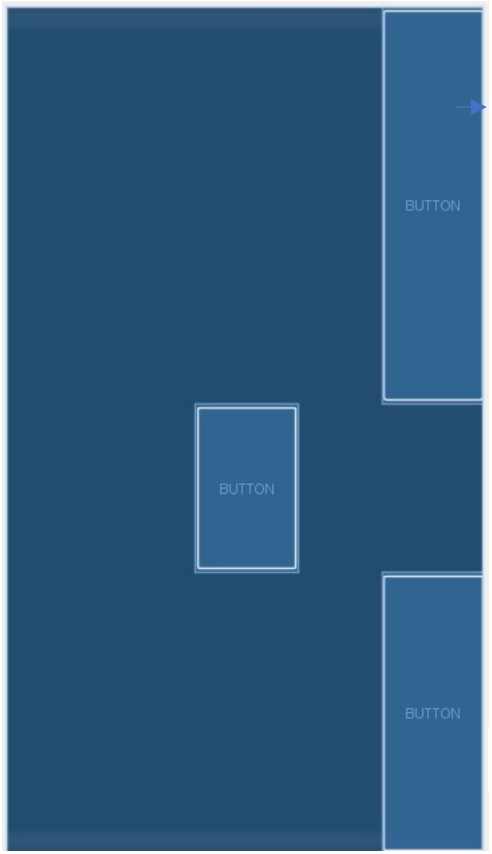


Linear Layout: Weight

Gives elements relative sizes by integers

- widget with weight K gets K/total fraction of total size

Can you guess the weights?



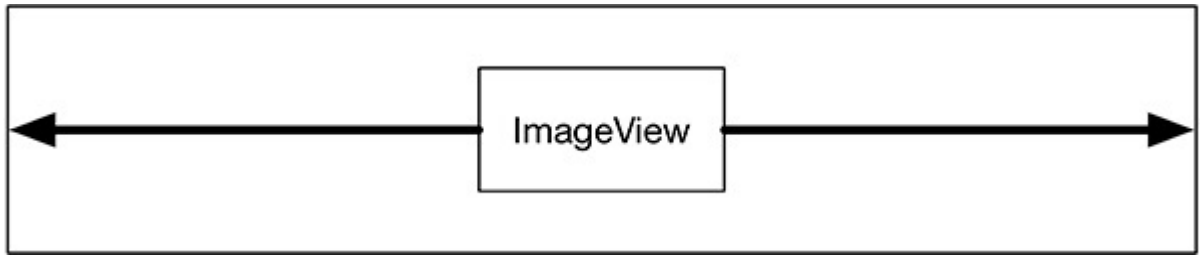

```
android:layout_weight="3"
```

```
android:layout_weight="1"
```

```
android:layout_weight="2"
```

Constraint Layout

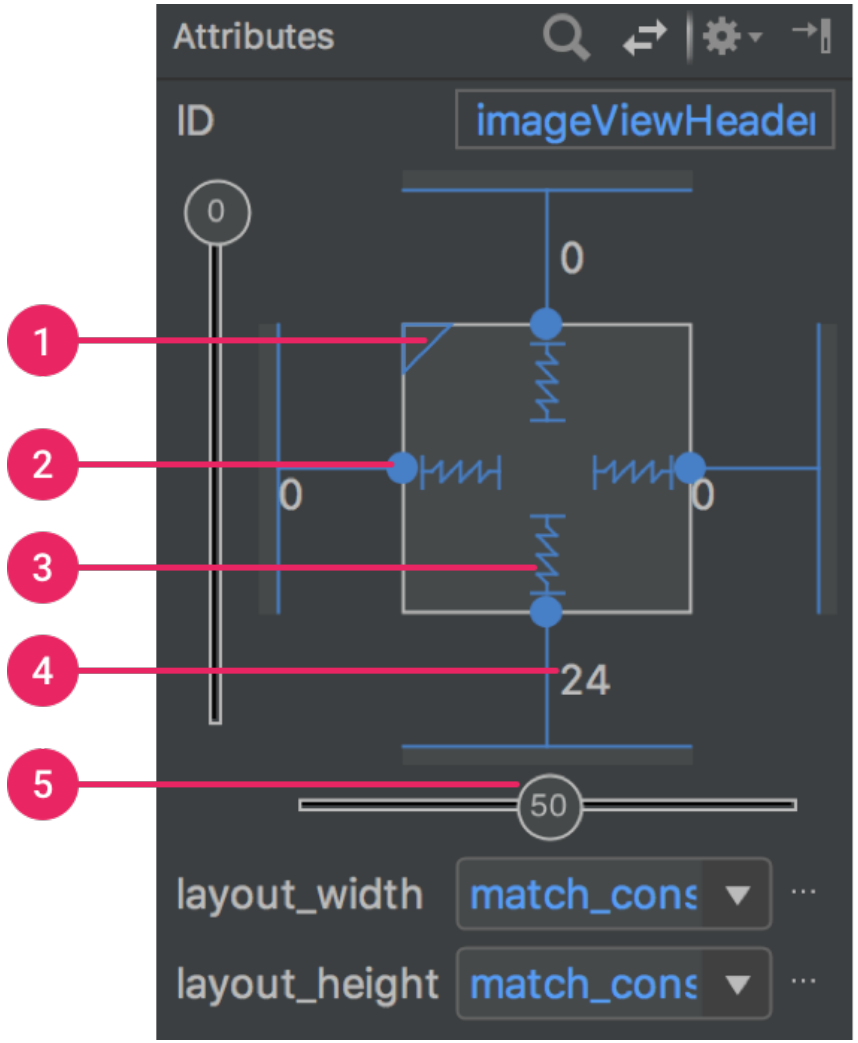
A constraint is like a rubber band: It pulls two things toward each other.



Constraint Layout: Attributes

The **Attributes** window includes controls for

1. size ratio,
2. Delete constraint,
3. height/width mode
4. Margins,
5. constraint bias.



Constraint Layout: Automatic

- Instead of adding constraints to every view as you place them in the layout, you can move each view into the positions you desire, and then click **Infer Constraints** to automatically create constraints.

- **Infer Constraints** scans the layout to determine the most effective set of constraints for all views.

- It makes a best effort to constrain the views to their current positions while allowing flexibility.

- You might need to make some adjustments to be sure the layout responds as you intend for different screen sizes and orientations.

Widgets in Android

EE P 523, Lecture 3

Widgets Attributes (I)

`android:layout_width` and **`android:layout_height`**

The `android:layout_width` and `android:layout_height` attributes are required for almost every type of widget.

They are typically set to either `match_parent` or `wrap_content`:

`match_parent`	view will be as big as its parent
`wrap_content`	view will be as big as its contents require

Widgets Attributes (II)

`android:orientation`

attribute on the two LinearLayout widgets determines whether their children will appear vertically or horizontally.

- The order in which children are defined determines the order in which they appear onscreen.
- In a vertical LinearLayout, the first child defined will appear topmost.
- In a horizontal LinearLayout, the first child defined will be leftmost.(Unless the device is set to a language that runs right to left, such as Arabic or Hebrew. In that case, the first child will be rightmost.)

`android:visibility="visible“`

- set widgets to be “visible” or “invisible”
- you can set the initial (default) value in the xml fine and
  then change the value of this attribute from the Activity as a response to an event

Widgets Attributes (III)

`android:text`

The TextView and Button widgets have `android:text` attributes.

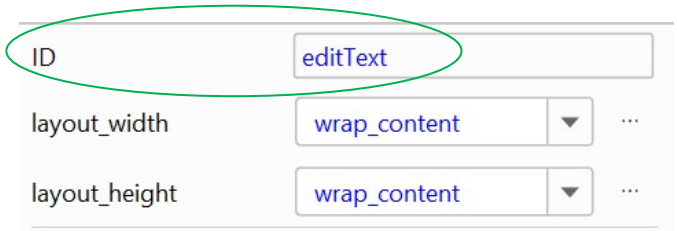
- This attribute tells the widget what text to display.

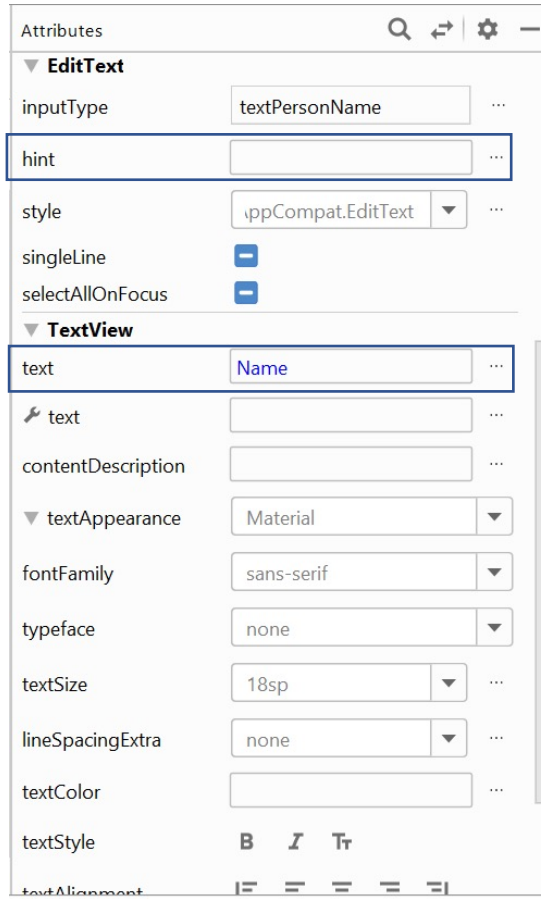
- Notice that the values of these attributes are not literal strings.
- They are references to string resources, as denoted by the @string/ syntax.

- A **string resource** is a string that lives in a separate XML file called a strings file.

- You can give a widget a hardcoded string, like android:text="True", but it is usually not a good idea. Placing strings into a separate file and then referencing them is better because it makes localization (which you will learn later in the course) easy.

Widgets: EditText





Example to read the text entered in a EditText from a user:

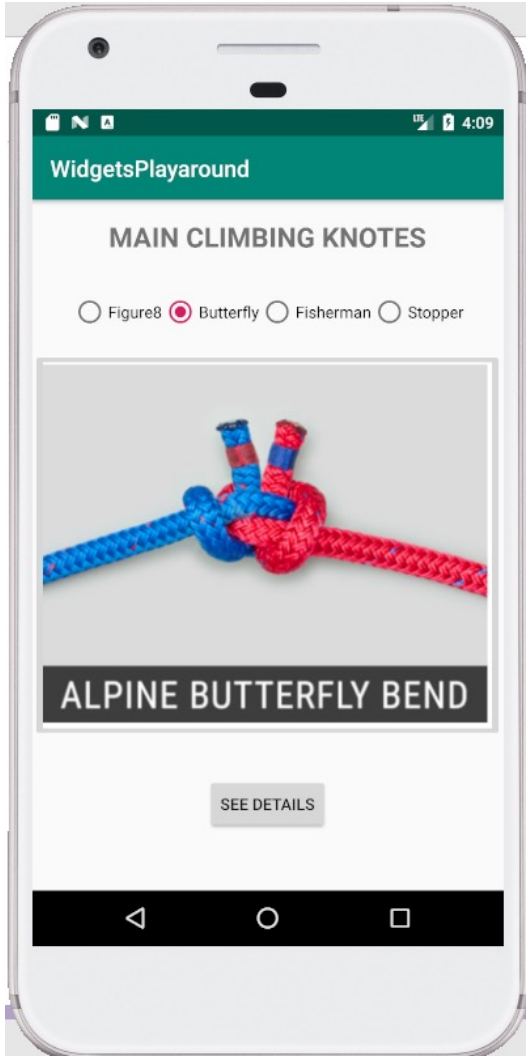
2 alternatives to refer to control the widget form the activity.

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

equivalent

```
import kotlinx.android.synthetic.main.activity_main.*

val message = editText.text.toString()
```

Widgets: RadioGroup




```
private fun updateKnoteImage(view:View) {
    val id = when (view) {
        figure8 -> R.drawable.figure8
        butterfly -> R.drawable.butterfly
        fisherman -> R.drawable.fisherman
        stopper -> R.drawable.stopper
        else
             -> R.drawable.figure8
    }
    knote.setImageResource(id)
}
```

When expression

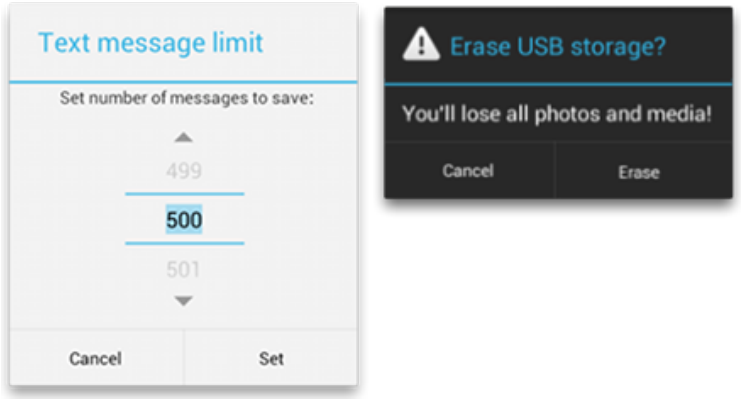
Replaces the ***switch*** operator of Java

In the simplest form it looks like this

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

Dialogs

- A dialog is a small window that prompts the user to make a decision or enter additional information.



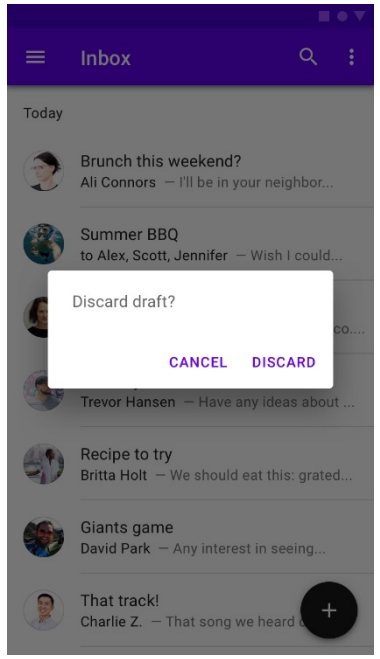
- A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed

- Dialogs are purposefully interruptive, so they should be used sparingly.

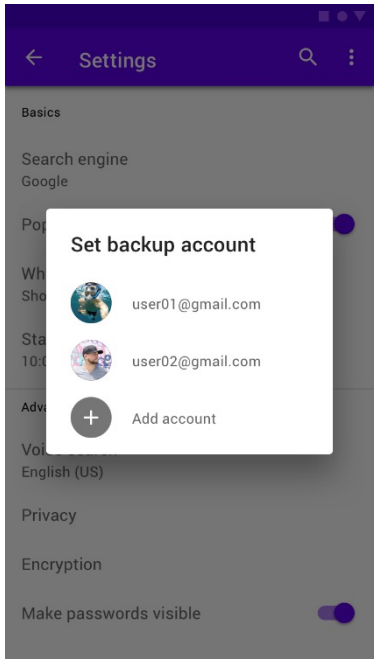
- Dialogs should be used for:
  - Errors that block an app's normal operation
  - Critical information that requires a specific user task, decision, or acknowledgement

Types of Dialogs

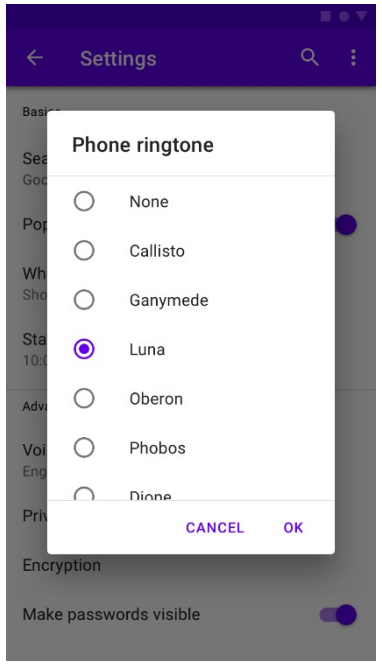
Alert



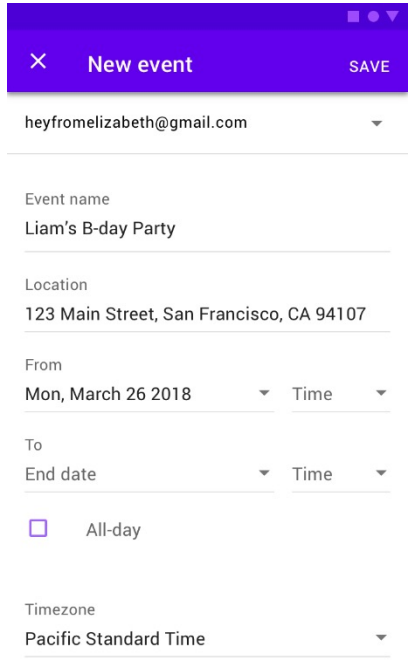
Simple



Confirmation



Full Screen



AlertDialog

There are three different action buttons you can add:

- **Positive**
  You should use this to accept and continue with the action (the "OK" action).

- **Negative**

You should use this to cancel the action.

- **Neutral**
  You should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel.
  It appears between the positive and negative buttons.
  For example, the action might be "Remind me later."

You can add only one of each button type to an AlertDialog. That is, you cannot have more than one "positive" button.

Build an Alert Dialog

1. Create a dialog in your activity class with dialog *builder*
2. The builder has many *set* methods to customize the dialog
3. When ready, *create()* the dialog and *show()* it
4. You can attach listener to the buttons

```
//make a dialog

val mDialog = AlertDialog.Builder(this)
mDialog.setTitle("This is the title")
mDialog.setMessage("This is a Dialog!")
mDialog.setPositiveButton("OK"){_, _ ->} //you can attach a
listener to respond to the OK button
mDialog.show()
```

Using String Resources

Define the string in the *resources/Strings* directory

```
<resources>
    <string name="app_name">myKnotes</string>
    <string name="initial_info">Click see details to add your notes here</string>
</resources>
```

Create a widget/element to write the string

```
<TextView
    android:id="@+id/user_info"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content“
           …
/>
```

Recover the string defined in the Resources directory

```
user_info.text=getString(R.string.initial_info)
```

Using String Resources

Define the string in the *resources/Strings* directory

```
<resources>
    <string name="app_name">myKnotes</string>
    <string name="initial_info">Click see details to add your notes here</string>
</resources>
```

Create a widget/element to write the string

```
<TextView
    android:id="@+id/user_info"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content“
           …
/>
```

Recover the string defined in the Resources directory

```
user_info.text=getString(R.string.initial_info)
```

Multiple Activities

EE P 523, Lecture 3

Multiple Activities

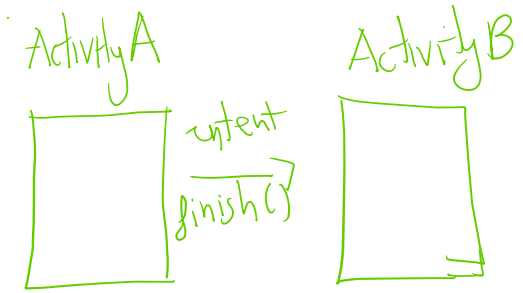
Many apps have **multiple activities**.

- An activity A can launch another activity B in response to an event.
- The activity A can pass data to B.
- The second activity B can send data back to A when it is done.

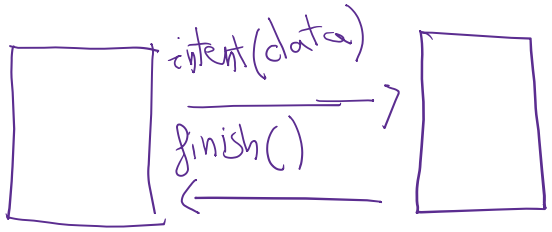
Multiple Activities

We are going to analyze 3 scenarios:

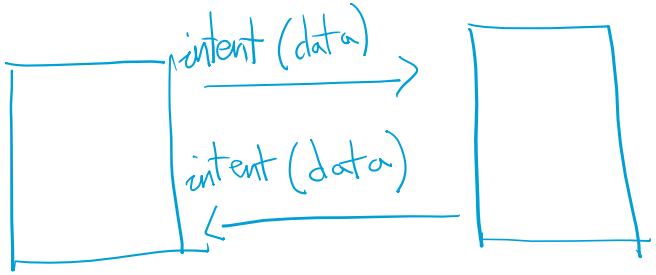
SCENARIO 1



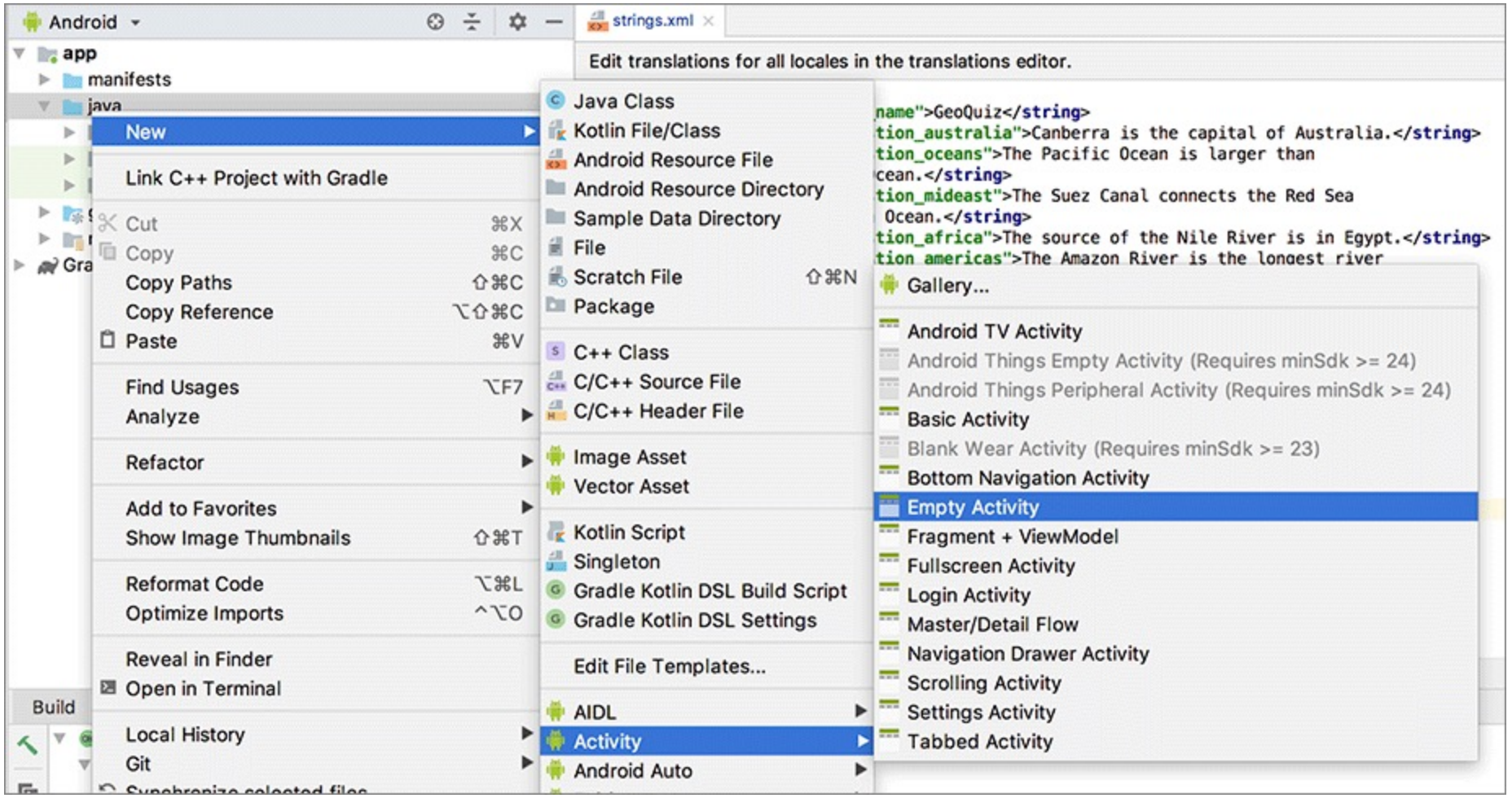
SCENARIO 2



SCENARIO 3

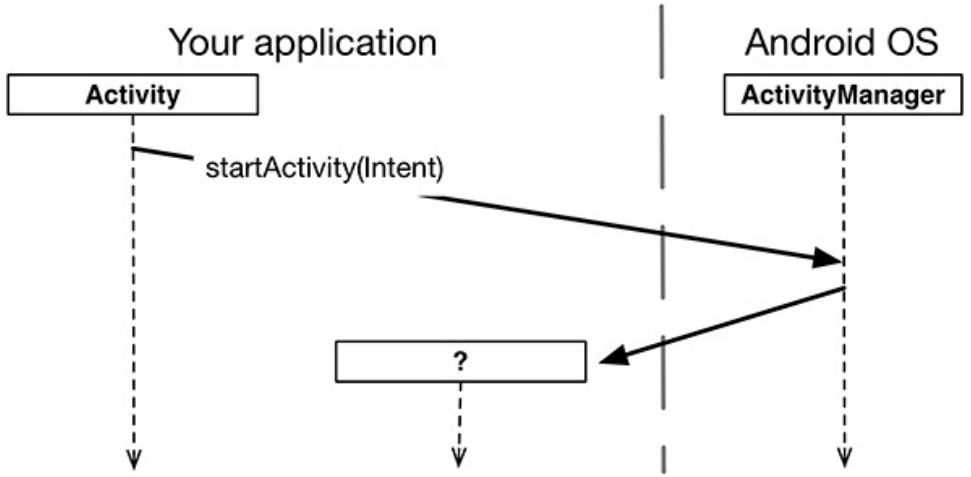


Starting an Activity



Start an activity: Intents

- The simplest way one activity can start another is with the `startActivity(Intent)` function.



- You might have guessed that `startActivity(Intent)` is a static function that you call on the Activity subclass that you want to start. But it is not.

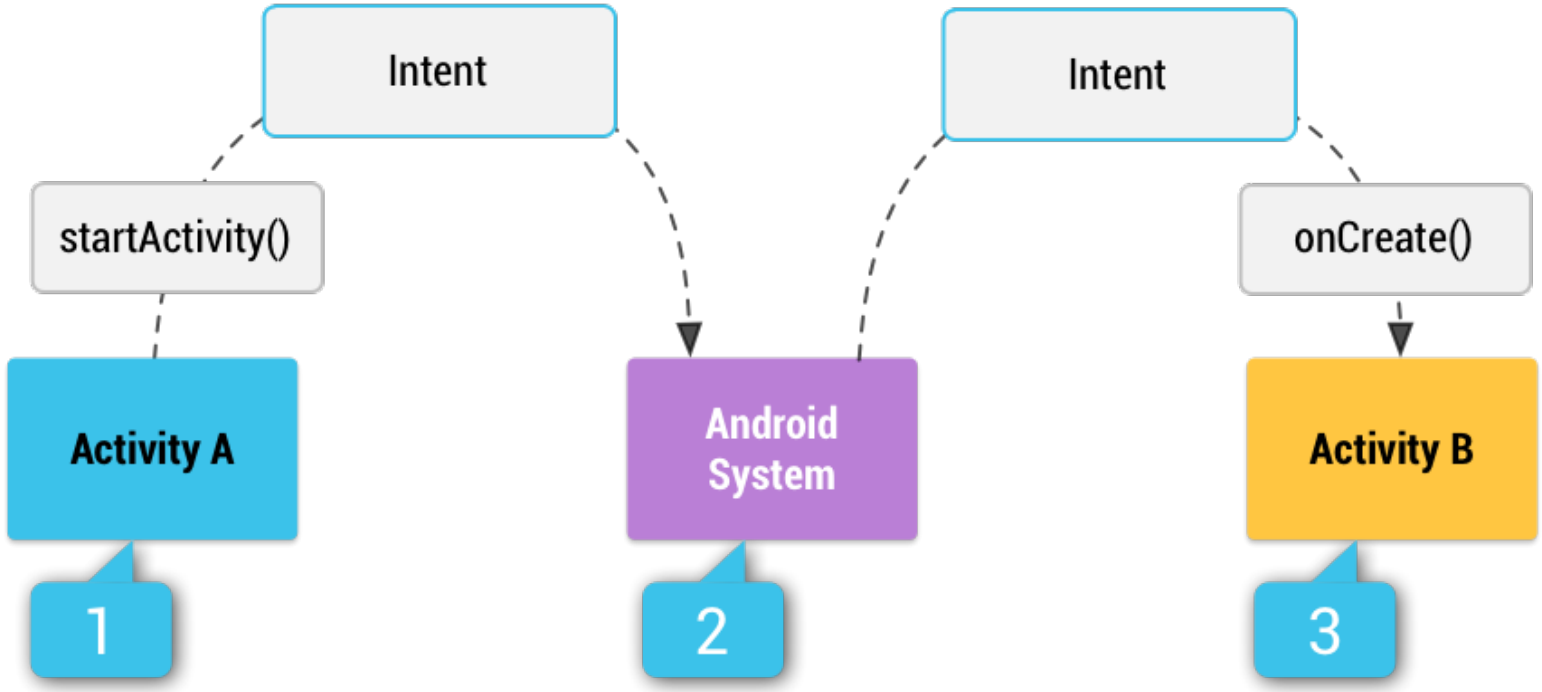
- When an activity calls `startActivity(Intent)`, this call is sent to the OS. In particular, it is sent to a part of the OS called the ActivityManager. The ActivityManager then creates the Activity instance and calls its on`Create(Bundle?)` function.

Multiple Activities: Intents

An ***Intent*** is a messaging object you can use to request an action from another app component.

- **Explicit intents**
  - Specify which application will satisfy the intent
  - 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.
  - Example: show the user a location on a map: use an implicit intent to request that another capable app show a specified location on a map.

Implicit Intent

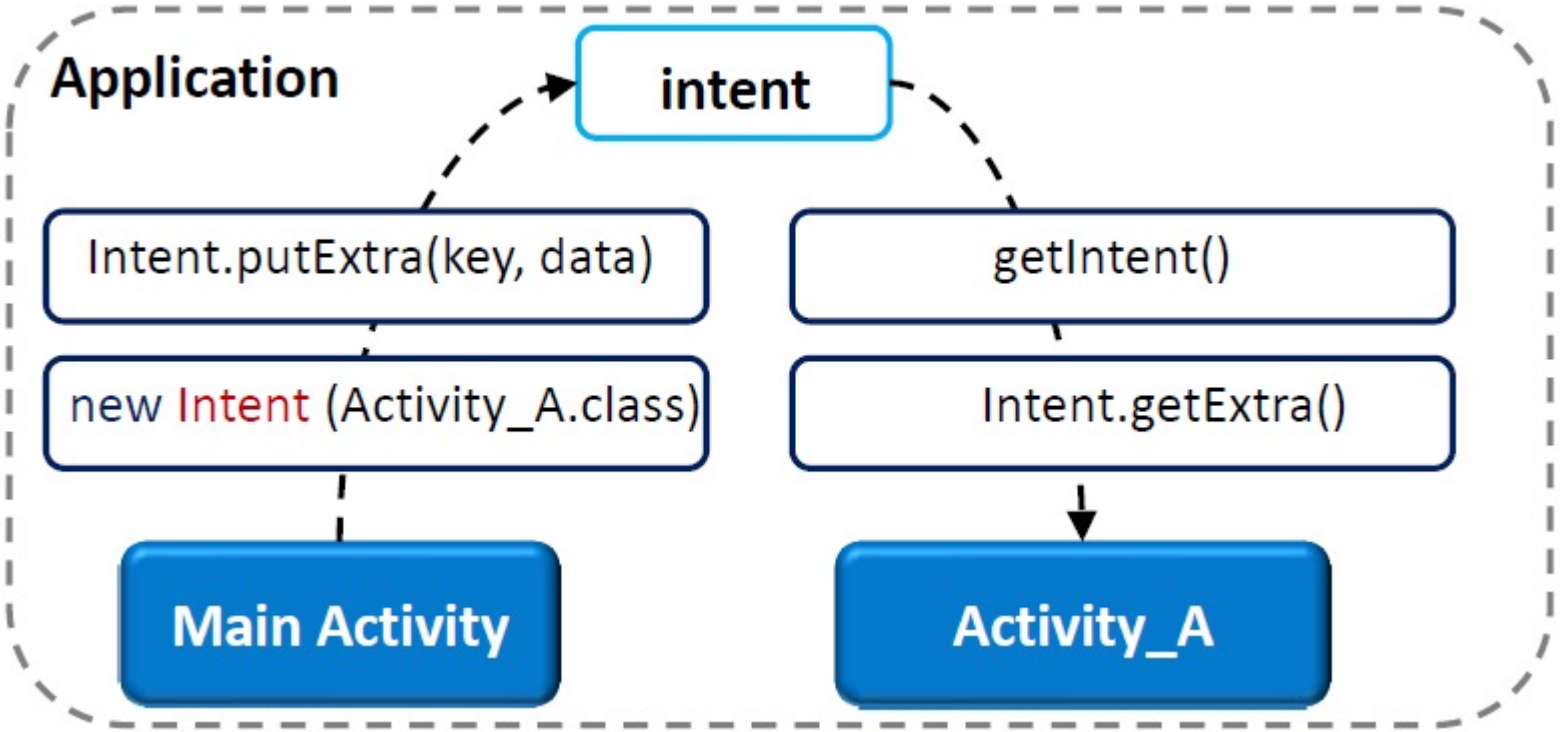


Activity A creates an Intent with an action description and passes it to startActivity().

The Android System searches all apps for an **intent filter** that matches the intent.

Match found: system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.

Explicit Intent



Intent.putExtra(…) comes in many flavors, but it always has two arguments.

- key: always a String key,
- data: type will vary. It returns the Intent itself, so you can chain multiple calls if you need to.

Starting a Second Activity

MainActivity.kt


```
const val KEY = “myMessage" //any String that you define

fun sendMessage(view: View) {
        val intent = Intent(this, SecondActivity::class.java)
        startActivity(intent)

        }

Wee need to pass the text entered by the user to the second activity!
```

1. Create an intent

2. Launch the intent

Passing Data to a Second Activity

MainActivity.kt


```
const val KEY = “myMessage" //any String that you define

fun sendMessage(view: View) {
        val editText = findViewById<EditText>(R.id.editText)
        val message = editText.text.toString()
        val intent = Intent(this, SecondActivity::class.java)
        intent.putExtra(KEY, message)

        startActivity(intent)
}
```

2. Put data into the intent

3. Launch the intent

Passing Multiple Data Points to a Second Activity

```
const val KEY = “myMessage" //any String that you define

fun sendMessage(view: View) {
        val editText = findViewById<EditText>(R.id.editText)
        val message = editText.text.toString()
        val intent = Intent(this, SecondActivity::class.java)
        intent.putExtra(KEY1, message1)
        intent.putExtra(KEY1, message2)
    intent.putExtra(KEY3, message3)


        startActivity(intent)

}
```

Passing data in Between Activities

Extras

- arbitrary data that the calling activity can include with an intent.

- You can think of them like constructor arguments, even though you cannot use a custom constructor with an activity subclass.

- The OS forwards the intent to the recipient activity, which can then access the extras and retrieve the data,

```
Intent.putExtra(…)
```

Receiving and Extracting Data


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

    setContentView(R.layout.activity_second)

     //To retrieve an Integer
      val received = intent.getIntExtra("myMessage")
     //To retrieve a String
      val received = intent.getStringExtra("myMessage")
```

Getting Resuls Back to the First Activity

Getting Resuls Back to the First Activity - Summary

1. ActivityA starts intent with startActivItyForResult() function

```
startActivityForResult(intent, requestCode)
```

2. ActivityB: receive data, and send other data to ActivityA using setResult()

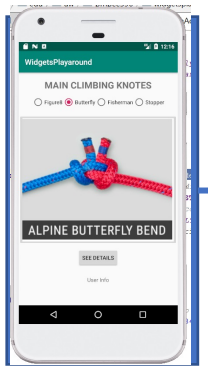
```
setResult(Activity.RESULT_OK, intent)
```

3. Activity3: receives data from ActivityB following these steps
   i. Override the function onActivityResult()
   ii. Check the requestCode
   iii. Check the request from previous activity was successful
   iv. Retrieves data from the intent

```
override fun onActivityResult(requestCode:Int, resultCode:Int, data: Intent?) {
    // Check which request we're responding to
    if (requestCode == PICK_KNOTE_REQUEST) {
        // Make sure the request was successful
        if (resultCode == Activity.RESULT_OK) {
            val data_from_ActivityB= data!!.getStringExtra("KEY_D")
}
    }
```

Explicit Intent: Getting Results from an Activity

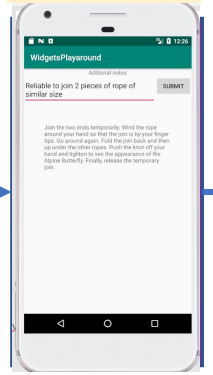
ActivityA.kt




```
intent.putExtra("KEY", message)
startActivityForResult(intent, request_code)
```

1. In response to some event: *start* ***ActivityB*** and expect some data back

ActivityB.kt



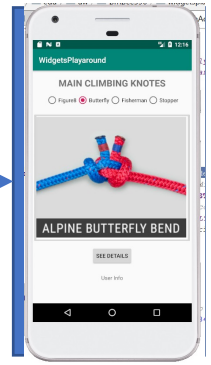

```
intent.putExtra("KEY_B", message)
setResult(Activity.RESULT_OK, intent)
finish()
```

2. Receive data from ***ActivityA***

```
intent.getIntExtra("KEY")
        Or
intent.getStringExtra("KEY")
```

3. Do something with it
4. Pass data to ***ActivityA***
5. Return to ***ActivityA.***

ActivityA.kt



Receive data from Activity B

```
fun onActivityResult(…) {
   intent.getIntExtra("KEY_B")
     ……
 }
```

Activities: Manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.uw.pmpee590.exampleapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

Launching main activity

How do we tell Android which activity to run/display when the App is first launched?

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="edu.uw.pmpee590.myapplication">

    <application
            android:allowBackup="true"
            android:icon="@mipmap/ic_launcher"
            android:label="My Application"
            android:roundIcon="@mipmap/ic_launcher_round"
            android:supportsRtl="true"
            android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

Using the Camera

EE P 523, Lecture 3

Using the Camera: Implicit Intent

1) **Implicit Intent**
   Delegating the work to another camera app on the device

   You want to take photos with minimal fuss, not reinvent the camera.
   Happily, most Android-powered devices already have at least one camera application installed.

```
val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
startActivityForResult(takePictureIntent,REQUEST_TAKE_PHOTO)
```

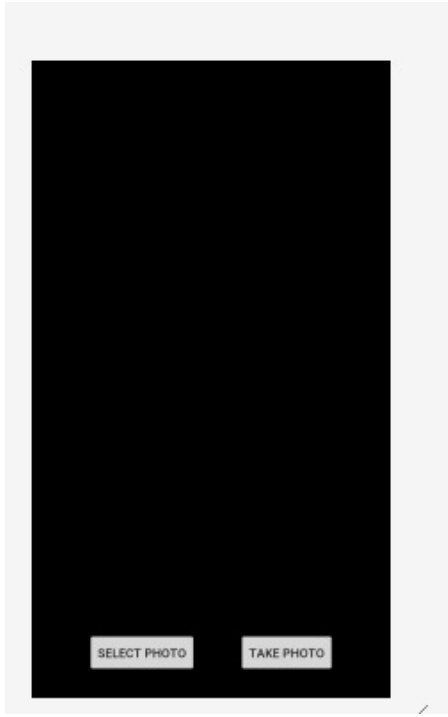
2) **Custom Camera control**
   Control the camera hardware directly using the framework APIs.
   Build a specialized camera application or something fully integrated in your app UI.

Implicit Intent: Take Pictures

1. **Ask the user to give permissions to use camera and save pictures**

in the onCreate function – **details in the takePicture App on Canvas**

```
getRuntimePermissions()
```

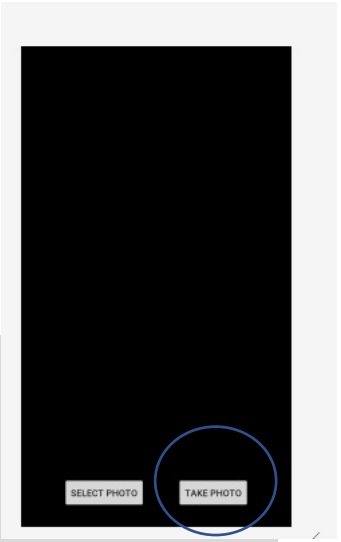


AND in the manifest file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.uw.eep523.takepicture">

    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

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

Implicit Intent: Take Pictures

2.A) Create an implicit intent to launch the CAMERA and take picture




```
fun startCameraIntentForResult(view:View) {
    // Clean up last time's image
    imageUri = null
    val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

        takePictureIntent.resolveActivity(packageManager)?.let {
         val values = ContentValues()
         values.put(MediaStore.Images.Media.TITLE, "New Picture")
         values.put(MediaStore.Images.Media.DESCRIPTION, "From Camera")
         imageUri = contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values)
         takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri)
         startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
    }
}
```

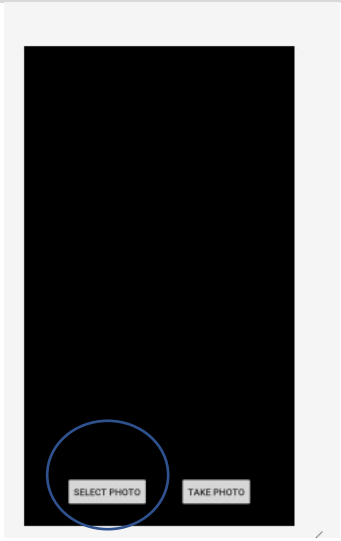
```
private const val REQUEST_IMAGE_CAPTURE = 1001
```

Implicit Intent: Take Pictures

2.B) Create an implicit intent CHOOSE picture from the phone gallery

```
fun startChooseImageIntentForResult(view:View) {
    val intent = Intent()
    intent.type = "image/*"
    intent.action = Intent.ACTION_GET_CONTENT
    startActivityForResult(Intent.createChooser(intent, "Select Picture"), REQUEST_CHOOSE_IMAGE)
}
```

```
private const val REQUEST_CHOOSE_IMAGE = 1002
```



Implicit Intent: Take Pictures

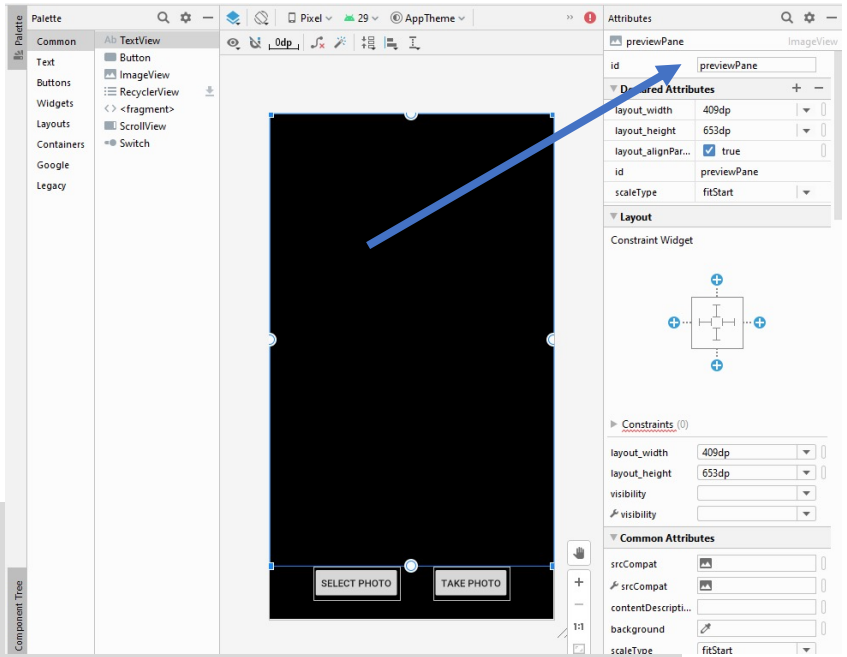
3) Receive data from Implicit Intent: New picture OR picture form gallery

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        tryReloadAndDetectInImage()
    } else if (requestCode == REQUEST_CHOOSE_IMAGE && resultCode == Activity.RESULT_OK) {
        // In this case, imageUri is returned by the chooser, save it.
        imageUri = data!!.data
        tryReloadAndDetectInImage()
    }
}
```

```
private const val REQUEST_IMAGE_CAPTURE = 1001
private const val REQUEST_CHOOSE_IMAGE = 1002
```

Implicit Intent: Take Pictures

4) Display picture on am ImageView widget




```
private fun tryReloadAndDetectInImage() {
    try {
        if (imageUri == null) {
            return
        }
        val imageBitmap = if (Build.VERSION.SDK_INT < 29) {
            MediaStore.Images.Media.getBitmap(contentResolver, imageUri)
       } else {
            val source = ImageDecoder.createSource(contentResolver, imageUri!!)
            ImageDecoder.decodeBitmap(source)
        }
        previewPane?.setImageBitmap(imageBitmap) //previewPane is the ImageView from the
                                                 layout

    } catch (e: IOException) {
        Log.e(TAG, "Error retrieving saved image")
    }
}
```

Files and Storage

EE P 523, Lecture 3

Files and Storage

Android provides several options for you to save your app data.

- **Internal file storage**: Store app-private files on the device file system.
- **External file storage**: Store files on the shared external file system. This is usually for shared user files, such as photos.
- **Shared preferences:** Store private primitive data in key-value pairs.
- **Databases**: Store structured data in a private database.

Files and Storage

Internal storage:

- It's always available.
- Files saved here are accessible by only your app.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

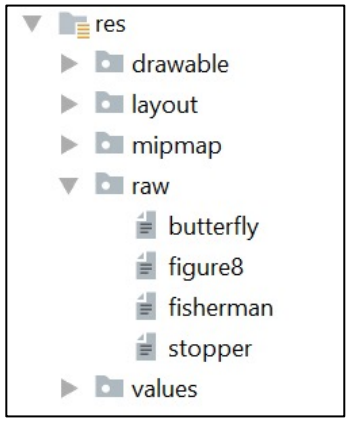
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from

Internal Storage

- Create a new resource directory **-> raw**
- Create resource file under ***raw/file_name***




```
//Read all text in the file butterfly.text

val textFiledID = resources.getIdentifier("butterfly", "raw", packageName)
val fileRead= resources.openRawResource(textFiledID).bufferedReader().readText()
```

External Storage

If your app needs to read/write the device's external storage, you must explicitly request **permission** to do so in your app's **AndroidManifest.xml** file.

- On install, the user will be prompted to confirm your app permissions.

```
<manifest ...>
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
...
</manifest>
```

https://developer.android.com/training/data-storage/files.html#kotlin

Files and Stream Objects

File - **Objects that represent a file or directory.**

– methods: `canRead, canWrite, create, delete, exists, getName, getParent, getPath, isFile, isDirectory, lastModified, length, listFiles, mkdir, mkdirs, renameTo`

InputStream, OutputStream - **flows of data bytes from/to a source or destination**

– Could come from a file, network, database, memory, ...
– Normally not directly used; reading/writing a byte at a time from the input.
– Instead, a stream is often passed as parameter to other objects like
- methods/properties *bufferedReader* to do the actual reading / writing.

Readers and Scanners

File and **InputStream** objects are not usually used directly.
Instead, you wrap them In a reader or scanner

BufferedReader – I/O object for reading a line at a time
- methods/properties: `readLine, ready, lineSequence, close`

Scanner – I/O object for reading lines or tokens at a time
- methods/properties: `readLine, hasNextDouble, hasNextInt, hasNextLine, next, nextDobule, nextIntnextLine`

Internal Storage

An activity has methods that you can call to read/write files

Method	Description
Resources.openRawResource(R.raw.id)	Read an input file from res/raw
openFileOutput(openFileOutput("name", mode)	Opens a file for writing (as an OutputStream)
openFileInput(openFileInput("name", mode)	Opens a file previously created by openFileOutput for reading (as an OutputStream)
filesDir	Returns a File for an internal directory for your app
cacherDir	Returns a File for a "temp" directory for scrap files

Activities and Activity States

EE P 523, Lecture 3

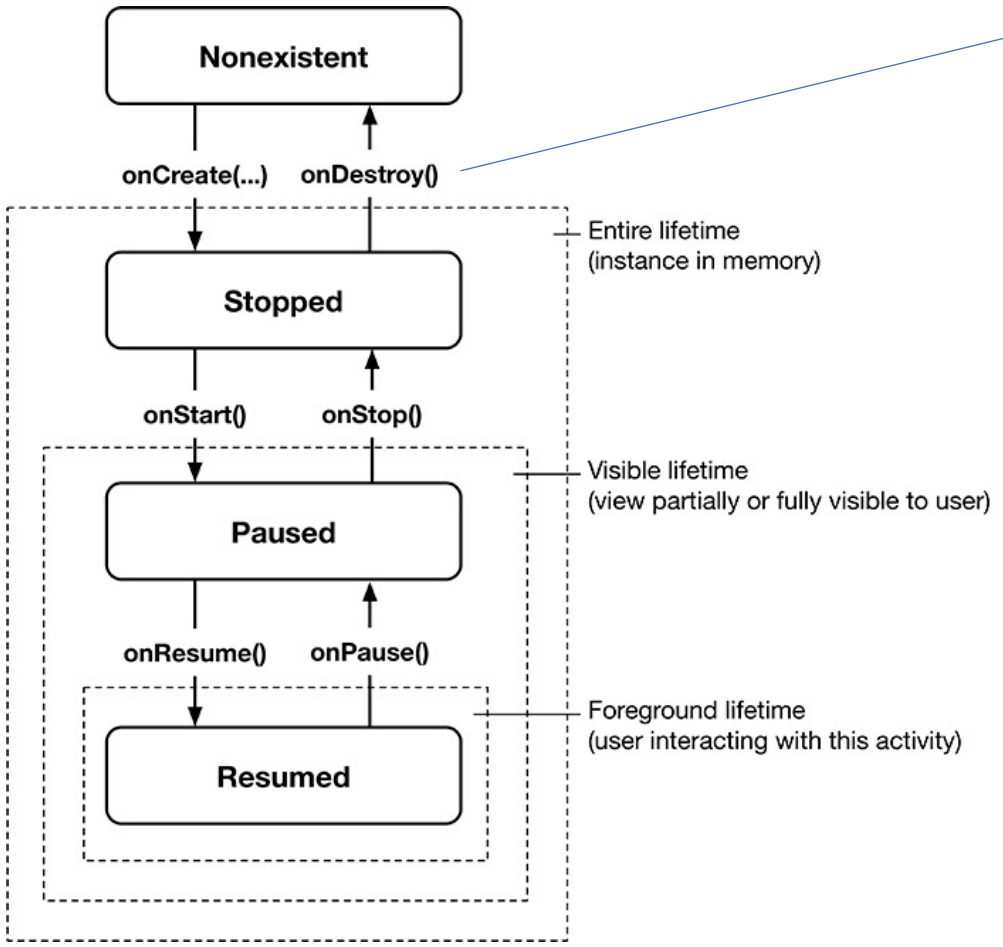
Activity State

An activity can be thought of as being in one of several states:

- **starting**: In process of loading up, but not fully loaded.
- **running**: Done loading and now visible on the screen.
- **paused**: Partially obscured or out of focus, but not shut down.
- **stopped**: No longer active, but still in the device's active memory.
- **destroyed**: Shut down and no longer currently loaded in memory.

- Transitions between these states are represented by **events** that you can listen to in your activity code.
  - onCreate, onPause, onResume, onStop, onDestroy, ...

Activity State



LifeCycle Callbacks

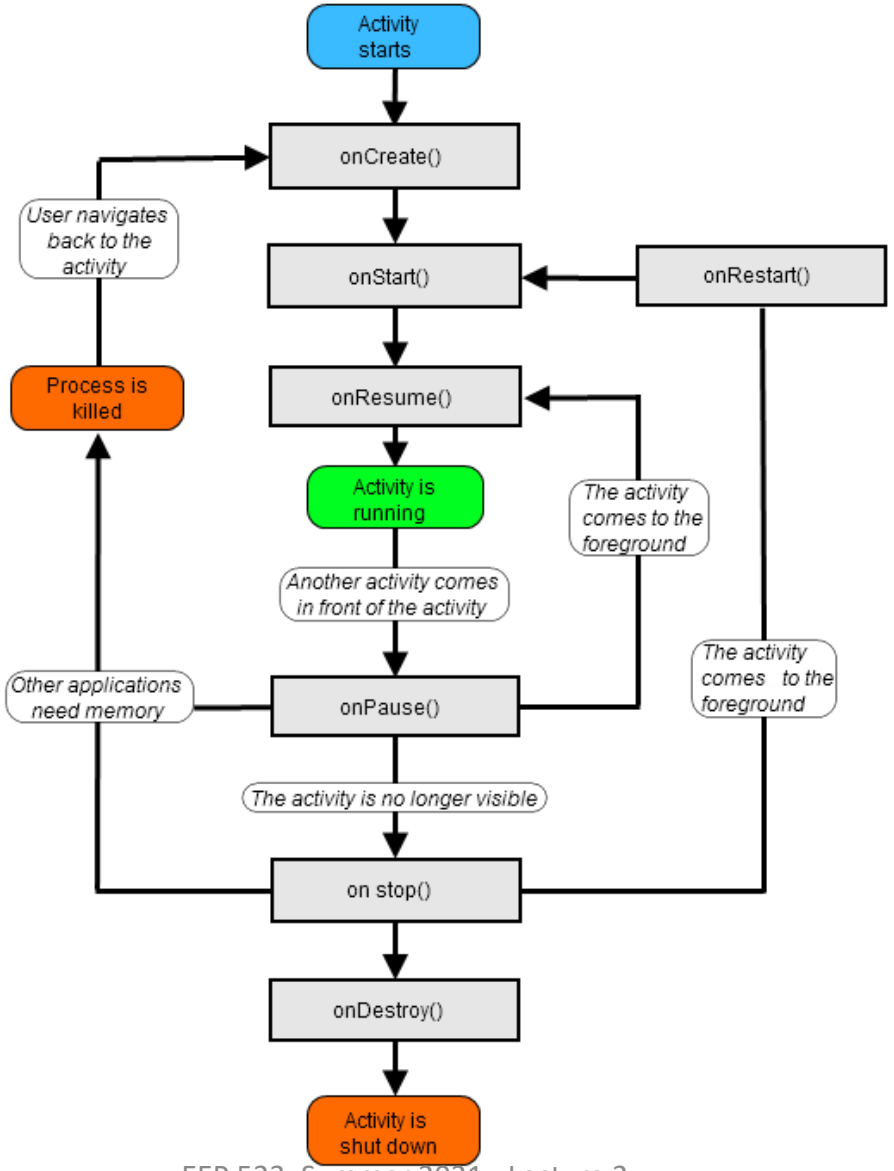
State	In memory?	Visible to user?	In foreground?
nonexistent	no	no	no
stopped	yes	no	no
paused	yes	yes/partially*	no
resumed	yes	yes	yes

Callback functions

- The concept of callbacks is to *inform a class synchronous / asynchronous*
- *if some work in another class is done*.

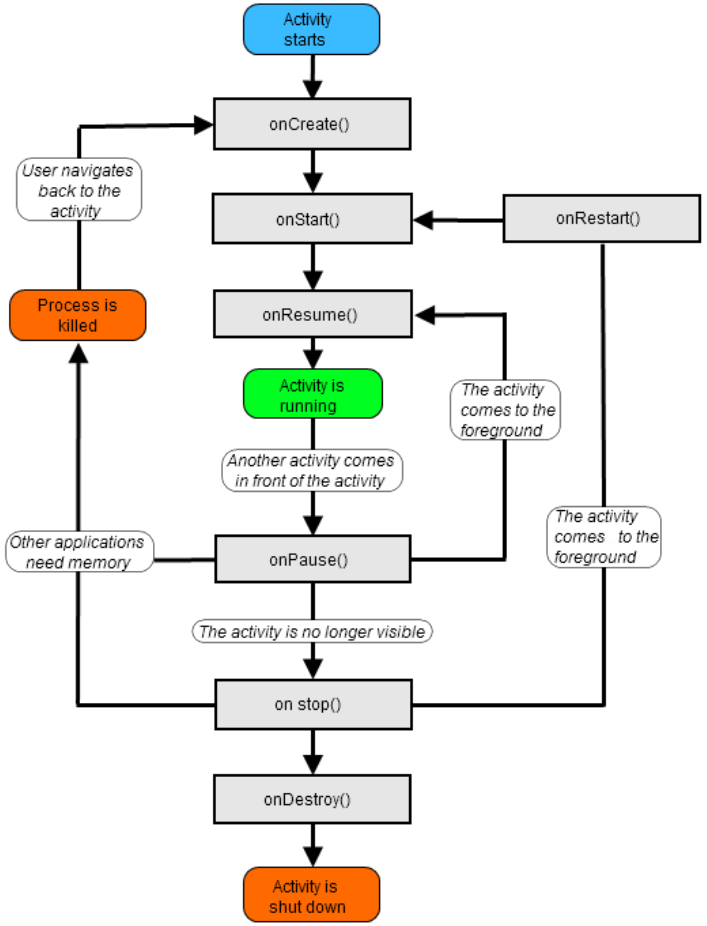
- Some call it the *Hollywood principle*: "Don't call us we call you".

Activity Lifecycle



Activity State Transitions

- Jump between activities in the same app: onPause() / onResume()
- Jump between two apps that are in memory: onStop() / onStart()



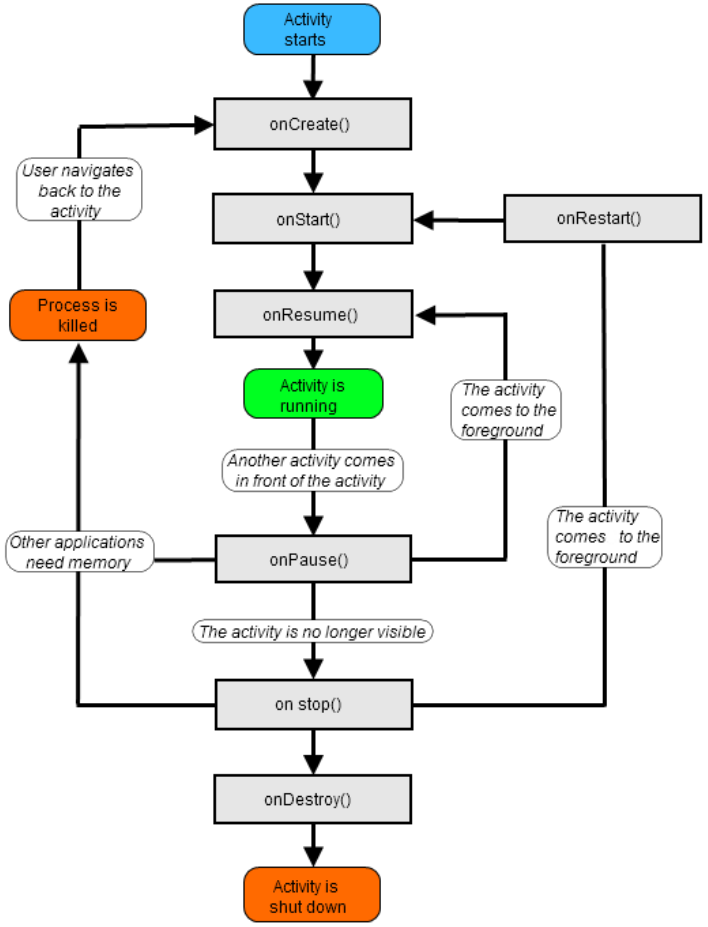
Navigation A -> Home

Home button

onStart()
onResume()
onPause()
onSaveInstanceState()
onStop()

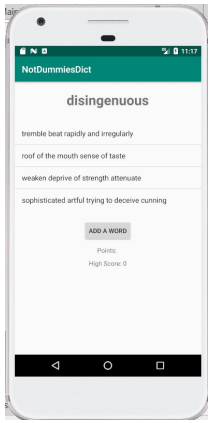
Back button

onReStart()
onStart()
onResume()
onPause()
onStop()
onDestroy()



Navigation A ->B

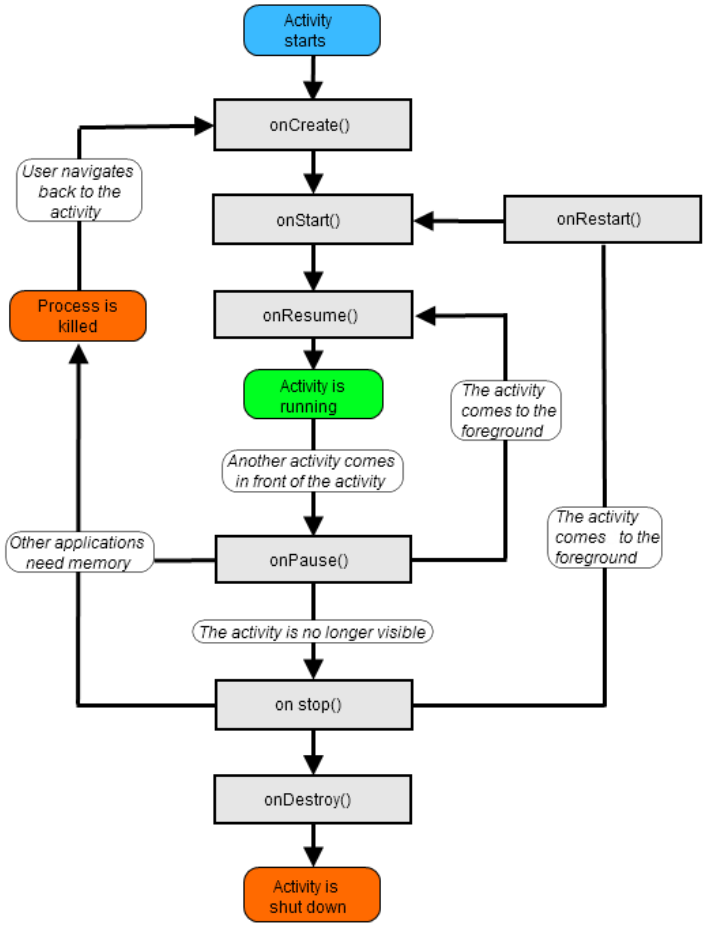
startActivity(intent)



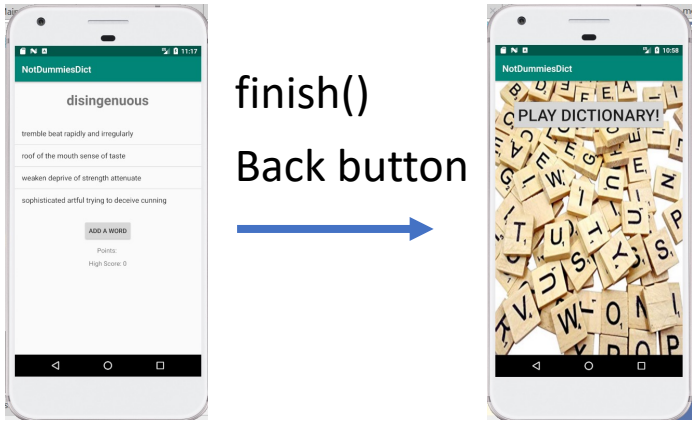
A- onStart()
A- onResume()
A- onPause()

B- onCreate()
B - onStart()
B – onResume()

A - onSaveInstanceState()
A- onStop()



Navigation B ->A



finish()

Back button

B- onPause()

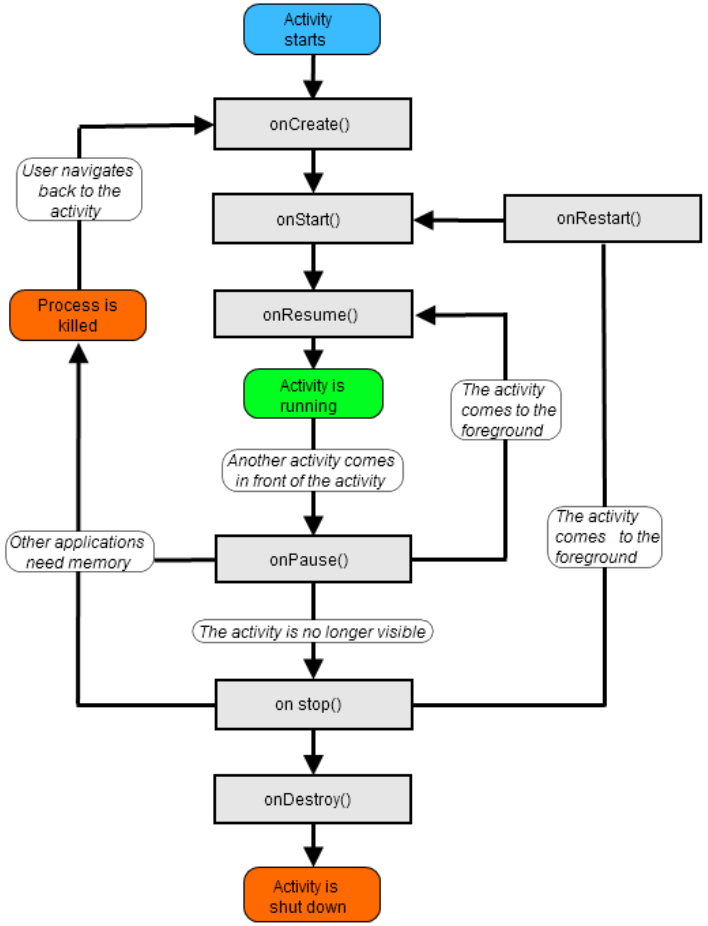
A - onReStart()

A - onStart()

A- onResume()

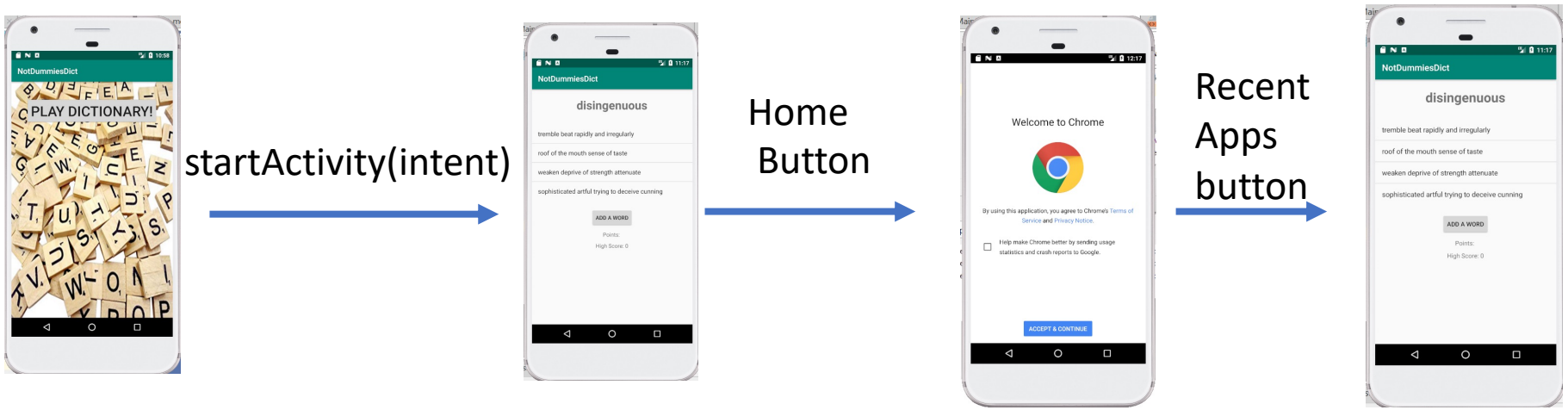
B – onStop()

B - onDestroy



Activity State Transitions

- Jump between two apps that are in memory: onStop() / onStart()



B – onPause()
B – onSavedInstnce()
B – onStop()

B – onRestart()
B – onStart()
B – onResume()

Lost activity state

User-initiated UI state dismissal

The user can completely dismiss an activity by:

- pressing the back button
- swiping the activity off of the Overview (Recents) screen
- navigating up from the activity
- killing the app from Settings screen
- completing some sort of "finishing" activity (which is backed by Activity.finish())

System-initiated UI state dismissal

- When you rotate the device's **orientation** from portrait to landscape

onCreate()

In **onCreate**, you create and set up the activity object, load any static resources like images, layouts, set up menus etc.

- after this, the Activity object exists
- think of this as the "constructor" of the activity

```
class FooActivity: AppCompatActivity() {
...
overide fun onCreate(savedInstanceState :Bundle?) {
    super.onCreate(savedInstanceState); // always call super
     setContentView(R.layout.activity_foo); // set up layout
     any other initialization code; // anything else you need
     }
}
```

onPause()

When **onPause** is called, your activity is still partially visible.
Example: Dialog pops up
May be temporary, or on way to termination.

- **Stop animations** or other actions that consume CPU.
- **Commit unsaved changes** (e.g. draft email).
- – **Release system resources** that affect battery life.

```
override fun onPause() {
      super.onPause(); // always call super
     if (myConnection != null) {
             myConnection.close(); // release resources
         myConnection = null;
      }
      }
}
```

onResume()

When **onResume** is called, your activity is coming out of the Paused state and into the Running state again.

- Also called when activity is first created/loaded!

- **Initialize resources** that you will release in onPause.
- **Start/resume animations** or other ongoing actions that should only run when activity is visible on screen.

```
override fun onResume() {
       super.onResume(); // always call super
     if (myConnection == null) {
      myConnection = new ExampleConnect(); // init.resources
          myConnection.connect();
       }
       }
}
```

onStop()

When **onStop** is called, your activity is no longer visible on the screen:

- User chose another app from **Recent Apps** window.
- User starts a **different activity** in your app.
- User receives a **phone call** while in your app.

Your app might still be running, but that activity is not.

- onPause is always called before onStop.
- onStop performs heavy-duty shutdown tasks like writing to a database.

```
override fun onStop() {
    super.onStop() // always call super
    Log.d("tag", "B- onDestroy()")
}
```

onStart() – onRestart()

- **onStart** is called every time the activity begins.

- **onRestart** is called when activity *was* stopped, but is started again later (all but the first start).
  - Not as commonly used; favor onResume.
  - Re-open any resources that onStop closed.

```
override fun onStart() {
    super.onStop() // always call super
    Log.d("tag", "B- onDestroy()")
}
```

```
override fun onReStart() {
    super.onStop() // always call super
    Log.d("tag", "B- onDestroy()")
}
```

onDestroy()

When **onDestroy** is called, your entire app is being shut down and unloaded from memory.

- Unpredictable exactly when/if it will be called.
- Can be called whenever the system wants to reclaim the memory used by your app.
- Generally favor onPause() or onStop() because they are called in a predictable and timely manner.

```
override fun onDestroy() {
    super.onDestroy() // always call super
    Log.d("tag", "B- onDestroy()")
}
```

Coordinating Activities

- When one activity starts another, they both experience lifecycle transitions
- 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.

Handling Configuration changes

A quick way to retain your activity's GUI state on **rotation** is to set the **configChanges** attribute of the activity in AndroidManifest.xml.

– This doesn't solve the other cases like loading other apps/activities.

```
<activity android:name=".MainActivity"
android:configChanges="orientation|screenSize"
...>
```

Handling Configuration changes

- The "orientation" value prevents restarts when the screen orientation changes.
- The "screenSize" value also prevents restarts when orientation changes, but only for Android 3.2 (API level 13) and above.
- The "screenLayout" value is necessary to detect changes that can be triggered by devices such as foldable phones and convertible Chromebooks.
- The "keyboardHidden" value prevents restarts when the keyboard availability changes.

```
<activity android:name=".MainActivity"
android:configChanges="orientation|screenSize"
...>
```

Handling Configuration changes

```
override fun onConfigurationChanged(newConfig: Configuration) {
    super.onConfigurationChanged(newConfig)

    // Checks the orientation of the screen
    if (newConfig.orientation === Configuration.ORIENTATION_LANDSCAPE) {
        Toast.makeText(this, "landscape", Toast.LENGTH_SHORT).show()

    } else if (newConfig.orientation === Configuration.ORIENTATION_PORTRAIT) {
        Toast.makeText(this, "portrait", Toast.LENGTH_SHORT).show()
    }
}
```

Caution: Handling the configuration change yourself can make it much more difficult to use alternative resources, because the system does not automatically apply them for you. This technique should be considered a last resort when you must avoid restarts due to a configuration change and is not recommended for most applications.

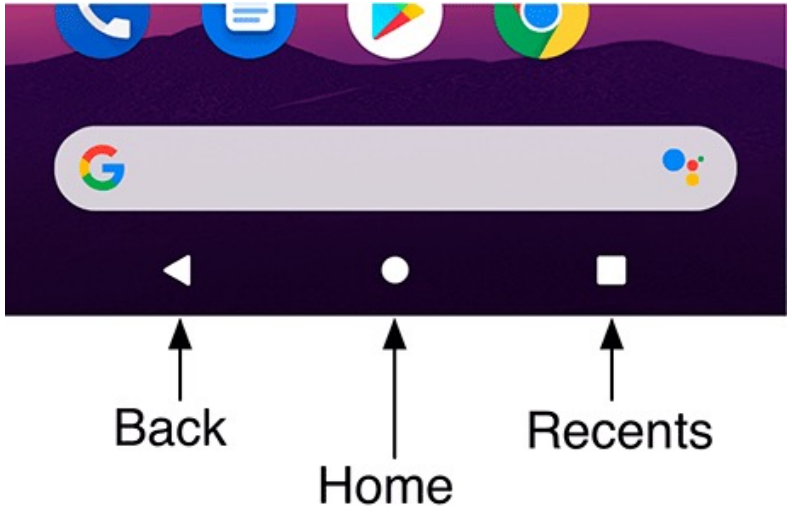
Persisting UI State

EE P 523, Lecture 3


Android Jetpack: ViewModel

Handling Activity State Changes

- ❑ Back bottom
- ❑ Home button
- ❑ Close App
- ❑ Rotate phone



We will learn about each by coding the App ***Pictionary***

Preserving UI State

	ViewModel	Saved instance state	Persistent storage
Storage location	in memory	serialized to disk	on disk or network
Survives configuration change	Yes	Yes	Yes
Survives system-initiated process death	No	Yes	Yes
Survives user complete activity dismissal/onFinish()	No	No	Yes
Data limitations	complex objects are fine, but space is limited by available memory	only for primitive types and simple, small objects such as String	only limited by disk space or cost / time of retrieval from the network resource
Read/write time	quick (memory access only)	slow (requires serialization/deserialization and disk access)	slow (requires disk access or network transaction)

Preserve UI State: *ViewModel*

- A **ViewModel** is related to one particular screen and is a great place to put logic involved in formatting the data to display on that screen.
- A ViewModel holds on to a model object and “decorates” the model – adding functionality to display onscreen that you might not want in the model itself.
- Using a ViewModel aggregates all the data the screen needs in one place, formats the data, and makes it easy to access the end result.
- ViewModel is part of the androidx.lifecycle package,which contains lifecycle-related APIs including lifecycle-aware components.
- Lifecycle-aware components observe the lifecycle of some other component, such as an activity, and take the state of that lifecycle into account.

Preserve UI State: *ViewModel*

1. Add dependencies in *app/build.gradle (Module:app)*

   Your project's dependencies live in a file called *build.gradle* (recall that Gradle is the Android build tool).

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'

   //Lifecyle extenssions
    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'


}
```

Preserve UI State: *ViewModel*

1. Add dependencies in *app/build.gradle (Module:app)*
2. Create your ViewModel Class

```
import androidx.lifecycle.ViewModel

class WordtoGuess: ViewModel() {

    override fun onCleared() {
        super.onCleared()
        Log.d("tag", "ViewModel instance about to be destroyed")
    }

}
```

The **onCleared()** function is called just before a **ViewModel** is destroyed. This is a useful place to perform any cleanup, such as un-observing a data source.

Preserve UI State: *ViewModel*

1. Include in Gradle
2. Create your ViewModel Class
3. Access the ViewModel from your activity

```
private val wordViewModel: WordtoGuess by lazy {
    ViewModelProviders.of(this).get(WordtoGuess::class.java)
}
```

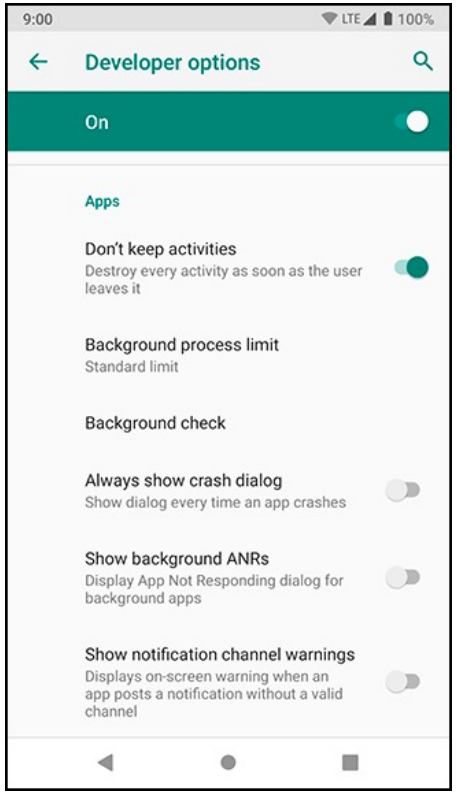
A property defined via by lazy is initialized using the supplied lambda upon first use, unless a value had been previously assigned.
Read more:
https://www.bignerdranch.com/blog/kotlin-when-to-use-lazy-or-lateinit/

Notes:

- Using by lazy allows you to make the `wordViewModel` property a val instead of a var. This is great, because you only need (and want) to grab and store the `WordtoGuess` when the activity instance is created – so `wordViewModel` should only be assigned a value one time.
- More importantly, using by lazy means the `wordViewModel` calculation and assignment will not happen until the first time you access `wordViewModel`. This is good because you cannot safely access a ViewModel until Activity.onCreate(…). If you try to call ViewModelProviders.of(this).get(`WordtoGuess` ::class.java) before Activity.onCreate(…), your app will crash with an IllegalStateException.
- When the activity queries for the WordtoGuess after a configuration change, the instance that was first created is returned. When the activity is finished (such as when the user presses the Back button), the ViewModel-Activity pair is removed from memory.

Preserve UI State: *ViewModel*

What happens if the the OS kills an App?



Will destroy an App when the user click on the HOME button (disabled by default)

Preserve UI State: *save instance state*

When an activity is being destroyed, the event method **onSaveInstanceState** is also called.

- This method should save any "non-persistent" state of the app.
- **non-persistent state**: Stays for now, but lost on shutdown/reboot.

Accepts a **Bundle** parameter storing key/value pairs.

- Bundle is passed back to activity if it is recreated later.

```
override fun onSaveInstanceState(outstate:Bundle?) {
  super.onSaveInstanceState(outState); // always call super
       outState.putInt("name", value);
       outState.putString("name", value);
...
}
```

Preserve UI State: *save instance state*

`onSaveInstanceState(savedInstanceState: Bundle)`

```
override fun onSaveInstanceState(savedInstanceState: Bundle) {
    super.onSaveInstanceState(savedInstanceState)
    Log.i("TAG", "onSaveInstanceState")
    savedInstanceState.putInt("key", points)
}
```

Example: save the number of points

`onRestoreInstanceState(savedInstanceState: Bundle?)`

```
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState)

    // Restore state members from saved instance
    savedInstanceState?.run { points = savedInstanceState.getInt("key", 0)
        points_text.text = "POINTS: $points"
        Log.d("tag", "B- onRestoreInstanceState()")
    }
}
```

Recover the number of points

Saving your classes

By default, your own classes can't be put into a Bundle.

- You can make a class able to be saved by implementing the *(methodless)* java.io.***Serializable*** interface.

```
@Serializable
data class Destination(
        var name : String? = "",
        var country : String? = "",
        var code : Int = 0,
)


val delhi = Destination(name = "Delhi", country = "India", code = 0)

val delhiAsString = JSON.stringify(delhi)
println(delhiAsString) // {"name":"Delhi","country":"India","code":0}
```

Saving your classes

```
class MainActivity : AppCompatActivity() {

  override fun onSaveInstanceState(outstate: Bundle? ) {
    super.onSaveInstanceState(outState);
       val delhi = Destination(name="Delhi",country="India",code=0);
       outState.putSerializable(“delhi", delhi);
}
```

Saving object to file

```
val delhi = Destination(name = "Delhi", country = "India", code = 0)
```

Write object

```
val fos = FileOutputStream(“myfilename.tmp”)
val oos = ObjectOutputStream(fos)
oos.writeObject(delhi)
oos.close()
fos.close()
```

Loading (w/o exception handling code):

```
val fis = FileInputStream(fileName)
val ois = ObjectInputStream(fis)

val result:Destination = ois.readObject()
ois.close()
val mName = result.getOne
```

Preserve UI State: *sharedPreferences*

How to save the data even after closing the App?

Preserve UI State: *sharedPreferences*

- Save relatively small collection of key-values
- **SharedPreferences** object points to a file containing key-value pairs and provides simple methods to read and write them.
- Each **SharedPreferences** file is managed by the framework and can be private or shared.

Example: save the highest score

```
val prefs = getSharedPreferences("myprefs", Context.MODE_PRIVATE)
val prefsEditor = prefs.edit()

prefsEditor.putInt("highScore", highScore)
prefsEditor.apply()
```

Preserve UI State: *sharedPreferences*

```
if(points>highestScore){
    val prefs = getSharedPreferences("myprefs", Context.MODE_PRIVATE)
    val prefsEditor = prefs.edit()
    prefsEditor.putInt("highScore", points)
    prefsEditor.apply()
    max_text.text = "High Score: $points"  // display the score in a TextView


}
```

Example: save highest score

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

    // Load the high score
    val prefs = getSharedPreferences("myprefs", Context.MODE_PRIVATE)
    highestScore = prefs.getInt("highScore", 0)
    max_text.text = "High Score: $highestScore”  // display the score in a TextView
```

Recover the highest score

Intents - 2

EE P 523, Lecture 3

Explicit intents

Passing multiple Basic Types

a) Multiple calls to putExtra

```
In your ActivityA.kt
val i = Intent(this, KnotesDetails::class.java).putExtra("KEY1", value1)
i.putExtra("KEY2",value2)
i.putExtra("KEY3, value3)
   ...
And then in your ActivityB.kt
val id = intent.getIntExtra("KEY1",default)
val id2 = intent.getIntExtra("KEY2",default)
val id3 = intent.getIntExtra("KEY3",default)
```

b) Passing an array

```
In your ActivityA.kt
private val COUNTRIES = arrayOf("Australia", "Brazil")
val i = Intent(this, ActivityB::class.java).putExtra("KEY", COUNTRIES)

In your ActivityB.kt
val country_list= intent.getStringArrayExtra("KEY",default)
country_first= country_list[0]
```

c) Passing an object

Explicit intents : passing objects

1. Create class that extends ***Parcelable***

```
import android.os.Parcelable
import kotlinx.android.parcel.Parcelize

@Parcelize class Book(val title: String, val author: String, val year: Int) : Parcelable
```

2. Passing the *Parcelable*

```
val myBook = Book(“myTitle”, ”myAuthor”, 2019)

val intent = Intent(this, AnotherActivity::class.java)
intent.putExtra("extra_item", myBook)
```

3. Retrieving the Parcelable

```
val item = intent.getParcelableExtra<Book>("extra_item")
// Do something with the item (example: set Item title and price)
val receivedTitle = item.title
```

Intents: Implicit or Explicit ?

Explicit: Start known/explicit Activity

```
val i = Intent(this, KnotesDetails::class.java)
i.putExtra("KEY", mKnoteID)
startActivityForResult(i, PICK_KNOTE_REQUEST)
```

This (current activity) ➡ KnotesDetails activity

Implicit: Start an action

```
 Intent(MediaStore.ACTION_IMAGE_CAPTURE).also { takePictureIntent ->
        takePictureIntent.resolveActivity(packageManager)?.also {
            startActivityForResult(takePictureIntent,
REQUEST_IMAGE_CAPTURE)
```

Ccurrent activity ➡ ACTION_IMAGE_CAPTURE

Implicit intents

- An intent allows you to start an activity in another app by describing a simple action you'd like to perform (such as "view a map" or "take a picture") in an Intent object.

- This type of intent is called an *implicit* intent because it does not specify the app component to start, but instead specifies an ***action*** and provides some *data* with which to perform the action.

- When you call startActivity() or startActivityForResult() and pass it an implicit intent, the systemresolves the intent to an app that can handle the intent and starts its corresponding Activity.

If there's more than one app that can handle the intent, the system presents the user with a dialog to pick which app to use.

Common *Implicit* intents *(link)*

Main App	Actions
Alarm clock	https://developer.android.com/guide/components/intents-common#Clock
Calendar	https://developer.android.com/guide/components/intents-common#Calendar
Camera	https://developer.android.com/guide/components/intents-common#Camera
Contacts	https://developer.android.com/guide/components/intents-common#Contacts
Email	https://developer.android.com/guide/components/intents-common#Email
File Storge	https://developer.android.com/guide/components/intents-common#Storage
Maps	https://developer.android.com/guide/components/intents-common#Maps
Music or Video	https://developer.android.com/guide/components/intents-common#Music
New note	https://developer.android.com/guide/components/intents-common#NewNote
Phone	https://developer.android.com/guide/components/intents-common#Phone
Search	https://developer.android.com/guide/components/intents-common#Search
Settings	https://developer.android.com/guide/components/intents-common#Settings
Text messaging	https://developer.android.com/guide/components/intents-common#Messaging
Browser	https://developer.android.com/guide/components/intents-common#Browser

Implicit intents: Take Pictures

Example App "*TakePicture*" on Canvas

1. **Get permissions**
   a) Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.uw.eep523.facedetectorpicture">
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

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

   b) At runtime

```
getRuntimePermissions()
if (!allPermissionsGranted()) {
    getRuntimePermissions()
}
```

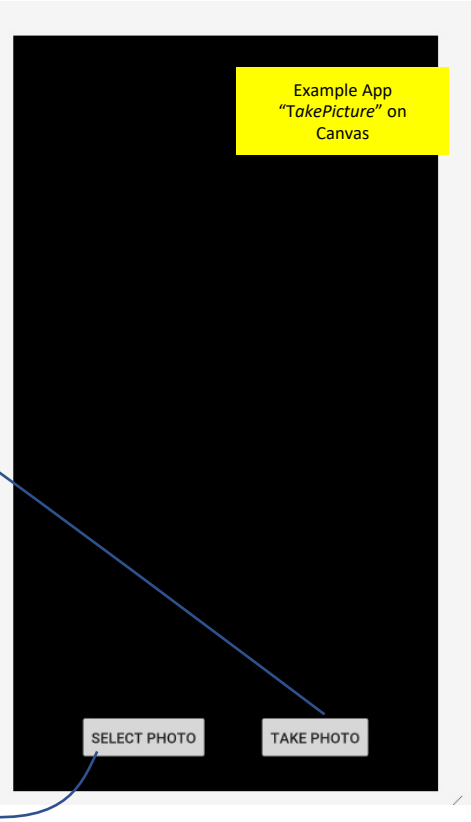
```
private fun getRequiredPermissions(): Array<String?> {
    return try {
        val info = this.packageManager.getPackageInfo(this.packageName, PackageManager.GET_PERMISSIONS)
        val ps = info.requestedPermissions
        if (ps != null && ps.isNotEmpty()) {
            ps
        } else {
            arrayOfNulls(0)
        }
    } catch (e: Exception) {
        arrayOfNulls(0)
    }
}
```

Implicit intents: Take Pictures

2. Launch the intent

```
fun startCameraIntentForResult(view:View) {
    // Clean up last time's image
    imageUri = null
    previewPane?.setImageBitmap(null)

    val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
    takePictureIntent.resolveActivity(packageManager)?.let {
        val values = ContentValues()
        values.put(MediaStore.Images.Media.TITLE, "New Picture")
        values.put(MediaStore.Images.Media.DESCRIPTION, "From Camera")
        imageUri = contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values)
        takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri)
        startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
    }
}
```

Take picture and add to gallery




```
fun startChooseImageIntentForResult(view:View) {
    val intent = Intent()
    intent.type = "image/*"
    intent.action = Intent.ACTION_GET_CONTENT
    startActivityForResult(Intent.createChooser(intent, "Select Picture"), REQUEST_CHOOSE_IMAGE)
}
```

Choose picture from gallery

Implicit intents: Take Pictures

3. Recover the image from the intent: image is recovered as a Bitmap

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        tryReloadAndDisplayImage()
    } else if (requestCode == REQUEST_CHOOSE_IMAGE && resultCode == Activity.RESULT_OK) {
        // In this case, imageUri is returned by the chooser, save it.
        imageUri = data!!.data
        tryReloadImage()
    }
}
```

```
private fun tryReloadImage() {
        val imageBitmap = if (Build.VERSION.SDK_INT < 29) {
            MediaStore.Images.Media.getBitmap(contentResolver, imageUri)
        } else {
            val source = ImageDecoder.createSource(contentResolver, imageUri!!)
            ImageDecoder.decodeBitmap(source)
        }
    }
}
```

Implicit intents: Take Pictures

4. Display the Image in a widget/view *(you will need both for assignment 2)*

a) In an **ImageView**: `imageViewID?.setImageBitmap(imageBitmap)`

b) In a **DrawView** : `drawView?.background= BitmapDrawable(getResources(), imageBitmap);`

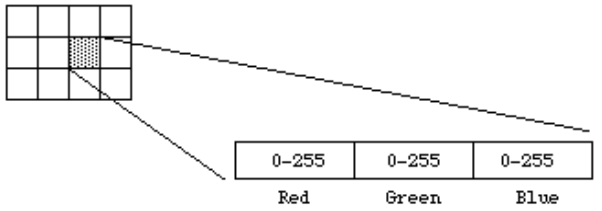
Image from
the camera or gallery

Bitmaps and Drawables

Bitmap

Regular rectangular mesh of cells called pixels, each pixel containing a colour value. They are characterized by only two parameters, the number of pixels and the information content (color depth) per pixel.

24 bit RGB
8 bits allocated to each red, green, and blue component. In each component the value of 0 refers to no contribution of that colour, 255 refers to fully saturated contribution of that colour. Since each component has 256 different states there are a total of 16777216 possible colours.



Drawable

General abstraction for "something that can be drawn."

- The Drawable class provides a generic API for dealing with an underlying visual resource that may take a variety of forms.
- Unlike a View, a Drawable does not have any facility to receive events or otherwise interact with the user.

Bitmap

Using a BitmapFactory, you can create bitmaps in 3 common ways:

- from a resource – you can create a bitmap from an image (such as .png or .jpg) located in your drawable folder.

```
val pict = BitmapFactory.decodeResource(resources,R.drawable.bored)
```

- file

- InputStream (such as an image from the camera)

Drawables (/res)

Supported file types:

- PNG (preferred),
- JPG (acceptable)
- GIF (discouraged).

Supporting Bitmaps

- Bitmaps can very easily exhaust an app's memory budget.
    - For example, the camera on the Pixel phone takes photos of up to 4048x3036 pixels (12 megapixels).

- If the bitmap configuration used is ARGB_8888, the default for Android 2.3 (API level 9) and higher, loading a single photo into memory takes about 48MB of memory (4048*3036*4 bytes).

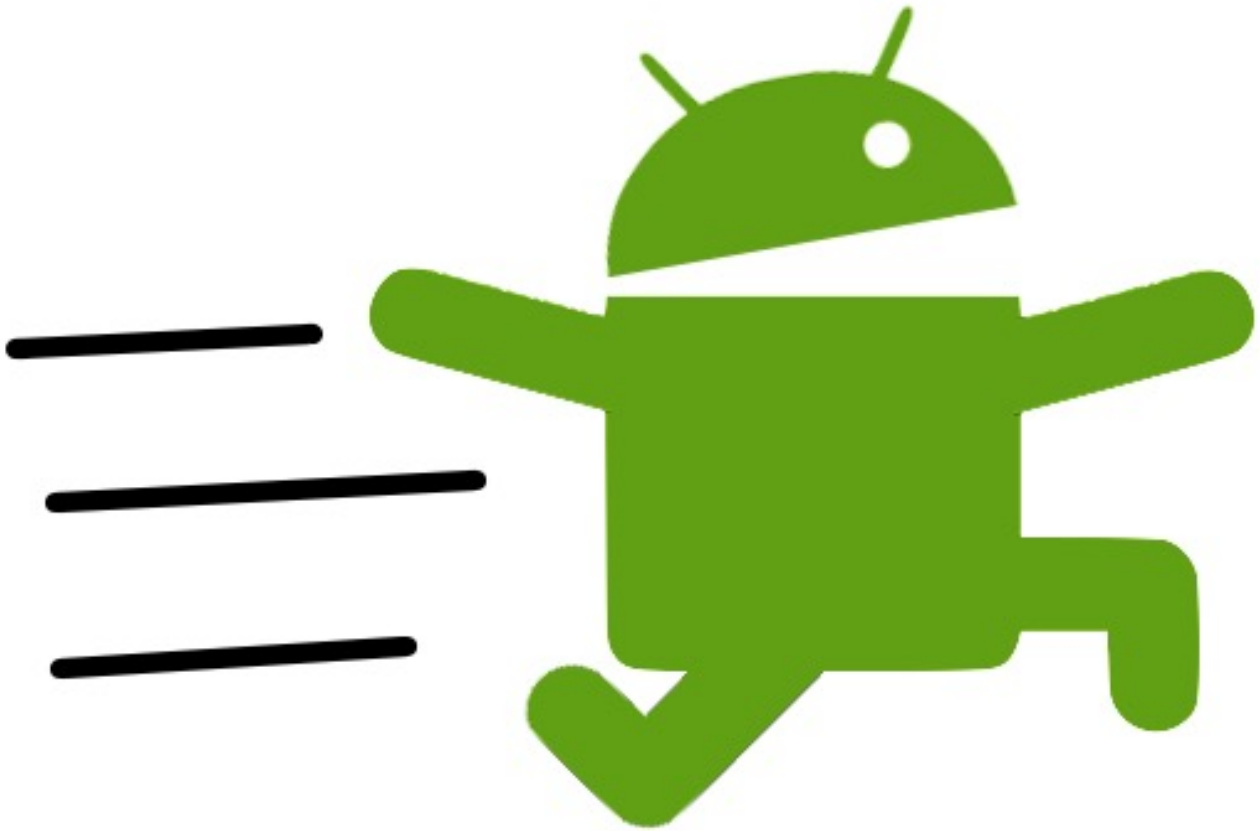
- Loading bitmaps on the UI thread can degrade your app's performance, causing slow responsiveness.
  It is therefore important to manage threading appropriately when working with bitmaps.

Tips for dealing with large Bitmaps in your App

1. Use an external library to load images in the most optimized manner.
   These libraries already follow best practices for loading images.
   - Picasso (link)
   - Glide (link)

2. Scale down large bitmaps
   Kotlin example to scale your bitmap

   https://developer.android.com/topic/performance/graphics/load-bitmap#load-bitmap



Your Questions

?