Android building blocks - Part 1
Activities, Intents, Permissions, Lifecycle and Persistent Storage

Download the full app created in this guide:
https://drive.google.com/file/d/1leLw1Fo7aZfvgL-r2fjwEA_frEfvXS_h/view?usp=sharing

Download the View Binding PDF Guide:
https://drive.google.com/file/d/1-o2TDRApSwzplOvXbrlafuAPaKkaf9lI/view?usp=sharing

Each Android App consist of four components:

1. **Activity** - this is the main android component. Each Activity represents a full screen. Nowadays we us the AndroidX AppCompatActivity which includes more advance features over the basic Activity which it inherits from. If we go deeper into the inheritance tree you will see that Activity also inherits from Context. The Context is an abstract class whose implementation is provided by the Android system when it is creating the Activity. Yes, the system creates the Activity and not us. The context allows us to interact with the system, it allows access to application level resources and classes. We will need it for each Android API class generation including creating other activities and other app components.

2. **Service** - Service is an app component designed to preform non-UI related operations. Like the Activity the Service also inherits from Context and it allows him to fully interact with the android system and so massive operations if needed. The fact that the Service doesn't have UI is sometimes an advantage - think about playing music from the background when the user is out of our app UI and wants the music to continue even when he navigates to other apps. An android App has two states - foreground and idle. Once the app has a foreground activity or a foreground service it considered to be foreground and has unlimited working power. After they move to the background the app will move to idle state and all of its other service and background activities will be killed by the system.

3. **Broadcast Receivers** - Broadcast are transmissions that the Android system can send when events occur. For example when connecting an external device, when connecting to a bluetooth or to wi-fi, when a call or sms arrives or sent, when the battery changes, when the boot completes, when connecting to a power source, when turning on

airplane mode or even when turning the screen on and off. In short for every external Android OS event sends a special broadcast. The Broast Receiver is a component that we can register to receive these broadcasts and do something with them. For example, when we detect that the boot completed we can start a service which let the user know the current weather or checks messages on our server or much more. The Broadcast Receiver doesn't inherit from the Context but receive a limited one by the system when the broadcast it was registered to is sent. This limited context doesn't allow the receiver to do any long term operations but instead it will be finished by the system after 5 seconds if it didn't finish before.

4. **Content Provider** - A Content provider is an app comment designed to provide data to other apps. All of the internal system database are arranged in a Sqlite databases and through the Content Resolver the system provides an interface for us to access their stored data like the address book, the content of the SD card, the calendar and other data stored locally on the phone.

Launching the app - order of events:

1. Everything starts in the **Android Manifest xml** file. In that file we set the app name, icon and other initial settings, the required device features and needed permissions but the most important thing, we declare the existence of all of our apps components and their capabilities. When the user installs our app the Android system creates a single instance from each component defined in the Manifest in the JVM Class Loader and uses this instance when it needs to create the component at runtime. It can do so either when the component is asked for explicitly by its name or when his declared capability is needed. This is the situation when the app launches: When the system detects a press on our app icon it sends the MAIN action to our package - the Activity which is registered to that action will be automatically created by the system. Please notice the exported attribute set to "true". This means that this component can be created by the system when its registered action occur.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="il.co.syntax.androidbuildingblocks">

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

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

</manifest>
```

2. When The Android OS creates an instance of the designated Activity it Automatically calls its **onCreate()** Lifecycle event function. By overriding this function we get our first entry point to the activity creation. Please note that after the super() call you can see the **setContentView()** function - this function receives the id of the initial xml layout file and inflate (inflation is creating objects from the static xml list) all of its views and subviews and populate them on the screen.

```
class MainActivity : AppCompatActivity() {

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


    }
}
```

3. After the initial layout has been created we can do additional

customizations like attaching listeners to buttons, play background music reading data from internal or external storage and populate a list with it and much more.

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

    val btn = findViewById<Button>(R.id.dial_button)
    btn.setOnClickListener { it: View!

        val number = findViewById<TextInputLayout>(R.id.phoneNumberInputLayout).editText?.text.toString()

        Toast.makeText( context: this,number,Toast.LENGTH_SHORT).show()
    }

}
```

But since we use view binding we add this line to the app Gradle

```
buildFeatures {
    viewBinding true
}
```

And our **onCreate** will look like this:

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

    val binding = ActivityMainBinding.inflate(layoutInflater);
    setContentView(binding.root)

    binding.loginBtn.setOnClickListener {  it: View!
        val name = binding.nameInputLayout.editText?.text.toString()
        Toast.makeText( context: this,name,Toast.LENGTH_SHORT).show()


    }
```

Intents

In the Android OS Intents is all we have :)
Intent are the way to interact with components, they can create them, pass information to them and more. Note the activities usually doesn't have constructors overloading, this is because we don't use constructors calls to create them but rather pass an Intent to the system and let her create the implement the Context for them. Our first entry point to their creation is the

lifecycle event function **onCreate**. When the system receives an Intent she is reading our intentions from it. Our intentions can be either **Explicit** where we are mentioning our desired Component by it's name or **Implicit** where we mention our desired Action string and the system finds the component for us according to what they declare - usually in the Manifest file. If more the one Component can answer that Action the system lets the user pick one and define it as default. The Implicit launch is the case in the app launch - the MainActivity declare himself to answer the action MAIN in the manifest. When the user installs the app the system creates the activity in the class loader it maps it to that action. When later the user press the icon, the system sends an Implicit intent with Main action and because he can answer it and it creates him.

Explicit Intent

So If we want to use the Explicit intent and start our own LoginActivity we first must create it. We have the short way: File->New->Activity->Empty Activity and give the Kotlin and the XML files a name and that's it. by doing this Android studio does allot for us: First it creates a new Kotlin class that extend AppCompatActivity and override the onCreate, then it creates a template xml file and inflate it in the previously overriden onCreate function. It also add the Activity to the Manifest XML file. So basically it is quite nice.

Please note that the default value this activity has for the **exported** attribute in the Manifest file is false, meaning this activity can be created only explicitly by mentioning of his name, He doesn't have any <intent filter> and won't be initiated by an ACTION like the MainActivity.

```
<activity
    android:name=".LoginActivity"
    android:exported="false" />
```

So to initiate it explicitly! create an Intent and use the context's **startActivity()** function:

```
val intent = Intent( packageContext: this,LoginActivity::class.java)
startActivity(intent)
```

Passing Data

what about the name and other information it needs?
Since we don't have a constructor we use the Intent to pass data upon creation. Each Intent contains a Bundle in his extra field. A **Bundle** is basically an HashMap where the key is a String and the value can be String, Int, Double, Float, array of them and any object that implements either the Java's

Serializable interface or its Android Parcelable implementation. We add this extras to the intent using it's **putExtra()** function and when the system creates the new activity it saves this Intent as his property and we use and the **getExtra()** with the same key:

In the calling Activity:

```
val intent = Intent( packageContext: this,LoginActivity::class.java).apply {
    putExtra("user_name",name)
}
startActivity(intent)
```

And in the newly created Activity:

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

        val binding = ActivityLoginBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.textView.text = intent.getStringExtra( name: "user_name")
    }
}
```

And that's it.

Tasks & Back Stack

Activities in the system are managed as an *activity stack*. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.

Your Activity is placed on top of your Task. By pressing the back button you kill this activity (like calling finish() from within the activity) and pops it from your back stack. By pressing the home button you take all of your Activity's Task and put them all in the background (after a while the OS will kill them if you won't return to them) and you can bring the task to the front along with all go the activities in it.

One thing you must understand regarding the Activity task is that each intent is creating a new activity instance. If, for example, from activity A you open B and then A again a new instance of activity A will be created. If you want to bring and existing Activity instance forward you need to change the Activity's **launchMode** attribute in this the Manifest file from standard to: **singleTop -** meaning if the activity already present in the top of the stack (it is in the front) it won't be recreated, **singleTask -** the system creates a new task just for the

activity but if an there is already an instance of that activity somewhere in that task the system bring him forward and routes the intent to it. Because it is not created, the already existing instance still holds the old intent in its Intent property. If you want to update this field with the new Intent, you need to override the **onNewIntent(intent: Intent)** function. And the last one **singleInstance** which is the same as before except that the system doesn't launch any other activities into the new task created for that activity (in case we wasn't already present).

For further reading and some nice drawings:
https://developer.android.com/guide/components/activities/tasks-and-back-stack

Implicit Intent

Now let's say we want to open an address on a Map, send an email, dial a number, open a browser to a specific site, take a photo, record a video or any other action which we want to preform but **don't really care who will perform it.** For this we have the Implicit Intent. In the Implicit Intent we set the Action String(it can be either one of the system fixed actions or our own custom one if you want - not common) and according to all of the installed d components and their declared abilities - declared with **<intent-filter>** - will be populated for the user to choose from. Please note that when using the Action string we usually set the extra data with the **setData(uri)** function. This functions accepts a URI that corresponds with the Action. For example when using ACTION_DIAL or ACTION_CALL the data is a phone number URI (starts with tel:):

```
binding.dialButton.setOnClickListener { it: View!
    val number = binding.phoneNumberInputLayout.editText?.text.toString()
    Toast.makeText( context: this,number,Toast.LENGTH_SHORT).show()

    val intent = Intent(Intent.ACTION_DIAL).apply { this: Intent
        data = Uri.parse( uriString: "tel:$number")
    }
    startActivity(intent)
}
```

Run the code. A Dialer with the phone number appears. Nice.
Try changing the ACTION_DIAL to ACTION_CALL. What happens?
Yes, the app crashed! this is because the later action try to actually preform the call while the former just showed a dialer and allowed the user to initiate the call (the first time the system dialer ran it also asked for the permission).

Before moving forward to the permissions please read here a a list of common intent action and their corresponding intent filters - IMPORTANT

https://developer.android.com/guide/components/intents-common

Permissions

Runtime vs install time permissions

First we must understand that before android 6.0 (marshmallow - api version 23) all permission were install time, meaning that all we had to do is to add the required permission to the Manifest file like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="il.co.syntax.androidbuildingblocks">

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

    <application
        android:allowBackup="true"
```

Toady we still need to do it, but for some permissions this is not enough.

In the old way, when the user installed the app he was given two options either to install the app and accept all the permissions without the ability to accept one and deny the other, to revoke them at a later time or even to understand exactly when they are using them or simply not installing the app.

Today for some permissions this is still the case, these permissions are mostly what I call background use permission (like getting boot and bluetooth or wifi connections that happen usually when we don't have UI present), and what Android defines as not **dangerous** - but **normal** permission. you can find the full list of them here: https://stackoverflow.com/a/36937109/2826409 (normal means you should only declare in Manifest file even after Android 6 and dangerous is what we are going to discuss here).

But for the most common permissions like calling, location, recording, reading contacts, and more. We must switch to the Runtime permission mechanism and beside writing them in the Manifest like before we must also present a Pop up window at runtime and specifically ask for them, just like in the iOS model - meaning we have to specifically ask for them when we need them and the user must grant us each requested permission. He can later revoke his approval and he can allow one while denying the other. A good practice is to ask for the permission only when we need it.

And this is how we do it:
First let's move the call execution to a separate function.

```
private fun call() {
    val number = binding.phoneNumberInputLayout.editText?.text.toString()
    val intent = Intent(Intent.ACTION_CALL).apply { this: Intent
        data = Uri.parse( uriString: "tel:$number")
    }
    startActivity(intent)
}
```

Now as a part of the new Launcher API that will be discussed later on we need to create the Permission request Launcher with the basic RequestPermission or RequestMultiplePermission Contract and provide a callback which upon approval will initiate the call:

```
class MainActivity : AppCompatActivity() {

    lateinit var binding : ActivityMainBinding

    val callPermissionLauncher : ActivityResultLauncher<String>  =
        registerForActivityResult(ActivityResultContracts.RequestPermission(), ActivityResultCallback { it: Boolean!
            if(it)
                call()
            else
                Toast.makeText( context: this, text: "can't call without permission",Toast.LENGTH_SHORT).show()
        })
```

When the user presses the call button we check if we already got the permission and if not we initiate the previously created launcher supplying it with the permission it needs to ask for. The system remembers the user approval but he can always revoke it and that is why before preforming the operation we must always check if we have the permission.
(please note that we use the **AppCompat** functions in order to support Android version earlier the 6.0)

```
binding.callBtn.setOnClickListener { it: View!

    if(ActivityCompat.checkSelfPermission( context: this,Manifest.permission.CALL_PHONE)
            != PackageManager.PERMISSION_GRANTED)
                callPermissionLauncher.launch(Manifest.permission.CALL_PHONE)
    else call()

}
```

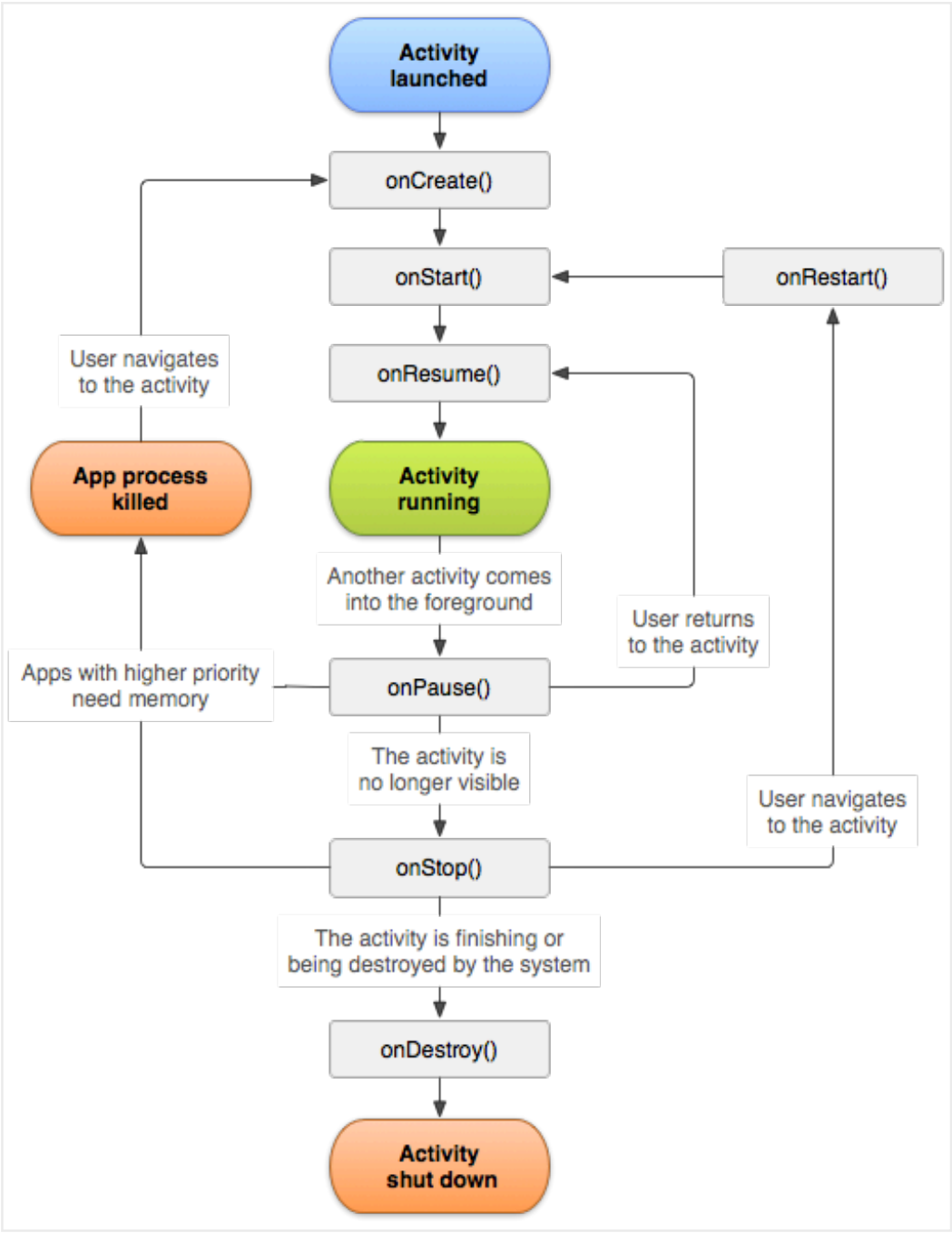
Please note: before asking for the permission Android encourage you to check whether you should show A UI explaining why you need this permission. You can check with the system whether you need to show the rational with the **shouldShowRequestPermissionRationale**() function if it returns true show a dialog explains why you need it if not just go ahead and ask for it.
shouldShowRequestPermissionRationale method returns false only if the user selected *Never ask again* or device policy prohibits the app from having that permission

Activity LifeCycle

An activity has essentially four states:

- If an activity in the foreground of the screen (at the top of the stack), it is *active* or ***running***.
- If an activity has lost focus but is still visible (that is, a new **non-full-sized** window has a focus and it it placed on top of your activity), it is ***paused***. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can't receive interactions from the user.
- If an activity is completely obscured visually by another activity, it is ***stopped***. It still retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
- If an activity is paused or stopped, **the system can drop the activity from memory** by either asking it to finish, or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

The following diagram from the Android Developers shows the important state paths of an Activity. The square rectangles represent callback methods you can implement to perform operations when the Activity moves between states. The colored ovals are major states the Activity can be in.



(Photo from the Android Developers https://developer.android.com/guide/components/activities/activity-lifecycle)

There are three key loops you may be interested in monitoring within your activity:

- The **entire lifetime** of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy().
- The **visible lifetime** of an activity happens between a call to onStart() until a corresponding call to onStop(). During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user.
- The **foreground lifetime** of an activity happens between a call to

onResume() until a corresponding call to onPause(). During this time the activity is in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states. For example when a dialog indicating a new message arrived, a call received, or any other window that is in the foreground even if it is not fully hides out activity.

In other words: When another window hide even a part of our activity the function **onPause** is called - This function is the best place to save user info to persistent storage.
Note: When overriding each function it is very important to call super first.

Lets examine a situation where we move from activity A to activity B what do think the order of events should be? Think about it.

The key is to remember that while **onPause** is called on the first lost of foreground, onStop will only get called when our views are no longer visible, and that will happen only when activity B has the foreground. This is why the order of events will be:
A - onPause()
B - onCreate()
B - onStart()
B - onResume()
A - onStop()

This is also a good reason to save the data on the onPause - if we need it in one of the new activity lifecycle events.

Persistent storage

As we have seen, when the android system calls the **onDestroy**() function all the app memory is deallocated and its resources are freed. So if we need to save some information across the user sessions we can use the lifecycle events to persist data across sessions. Android provides several options for you to save and persist your application data. The solution you choose depends on your specific needs:

- Shared Preferences - Store private primitive data in key-value pairs. As its name suggest this is mainly useful in saving simple user preferences like if it is the first run our not, whether he muted the music, whether he want green or blue background color, his already typed e-mail in an edit text and other basic user user information.
- Internal Storage - Store private data on the device memory, this storage is designed to be the app “sandbox”, its private to your app and will be deleted when the user uninstall your app. It is not limited in size but the user can clear it from the settings. The shared

preferences mentioned above are saved here as well as the ROOM DATABASE we will learn later on, but we can also write to this area directly using Java's streams.

- External Storage - Store public data on the shared external storage this information can be shared with other apps and can be saved even after your app is deleted. The External storage called "external" because you can share it with others, **it is not external to the device but to the app.** It is divided to two sections: The first is for use by our app. Like said it can be shared with others. It will be deleted when the user uninstall the app and writing to it doesn't requires permissions. The second is the external shared by all apps, writing and reading from it requires permissions and data saved there will not be deleted when the user uninstall our app.

shared preference

The SharedPreferences class provides you the easiest way of saving data to the device memory, the data will be saved while you app is installed on the device. We save all the information with the key-value bundle(hash Table) we have seen before - but with less options - only java primitives, String and a set of Strings.

To get a SharedPreferences object for your application, use one of two methods:

- getSharedPreferences() - Use this if you need multiple preferences files identified by name, which you specify with the first parameter (use a constant). Use this method to get a preference file to be used across all activities, meaning a file that can be accessed from anywhere in your app.
- getPreferences() - Use this if you need only one preferences file for your **Activity**. Because this will be the only preferences file for your Activity, you don't need to supply a name.

Here is an example of using the first option in the **onPause** lifecycle event

```
companion object{
    const val PREFS = "details"
}
```

```
override fun onPause() {
    super.onPause()

    getSharedPreferences(PREFS, MODE_PRIVATE).edit().apply { this: SharedPreferences.Editor!
        putString("user_name",binding.nameInputLayout.editText?.text.toString())
        putString("phone",binding.phoneNumberInputLayout.editText?.text.toString())
    }.apply()
}
```

Please note that writing to the file system can be either synchronous or a-synchronous. If you use the **apply()** function on the editor the writing is done later on but if you use the **commit()** function the system holds everything and writes the data to the filesystem now.

When we want to read data from the shared preferences we use the same file name and keys:

```
getSharedPreferences(PREFS, MODE_PRIVATE).apply { this: SharedPreferences!
    binding.nameInputLayout.editText?.setText(getString("user_name",""))
    binding.phoneNumberInputLayout.editText?.setText(getString("phone",""))
}
```

Note the MODE_PRIVATE flag which is our only option - the file is only readable by our app - the other modes WORD_READABLE and WORLD_WRITABLE considered to be dangerous and as of API 17 are deprecated (when google moved to SELinux).