

# ANDROID - Lab 1

## Lesson goals:

- Get familiar with Android Studio and the Virtual Devices
- Create and customize some basic views
- Introduction to activities
- Create a basic navigation between two different screens
- Display a simple dialog

## 1. Introduction to Android Studio

First of all you should download Android Studio, unless you already did, from here:

<https://developer.android.com/studio>

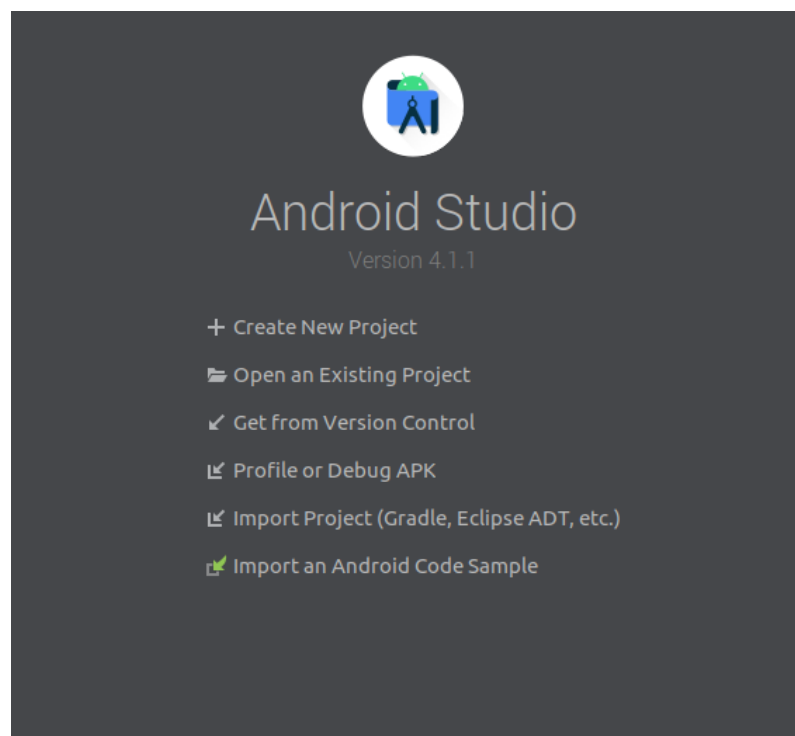


Android Studio provides the fastest tools for building apps on every type of Android device.

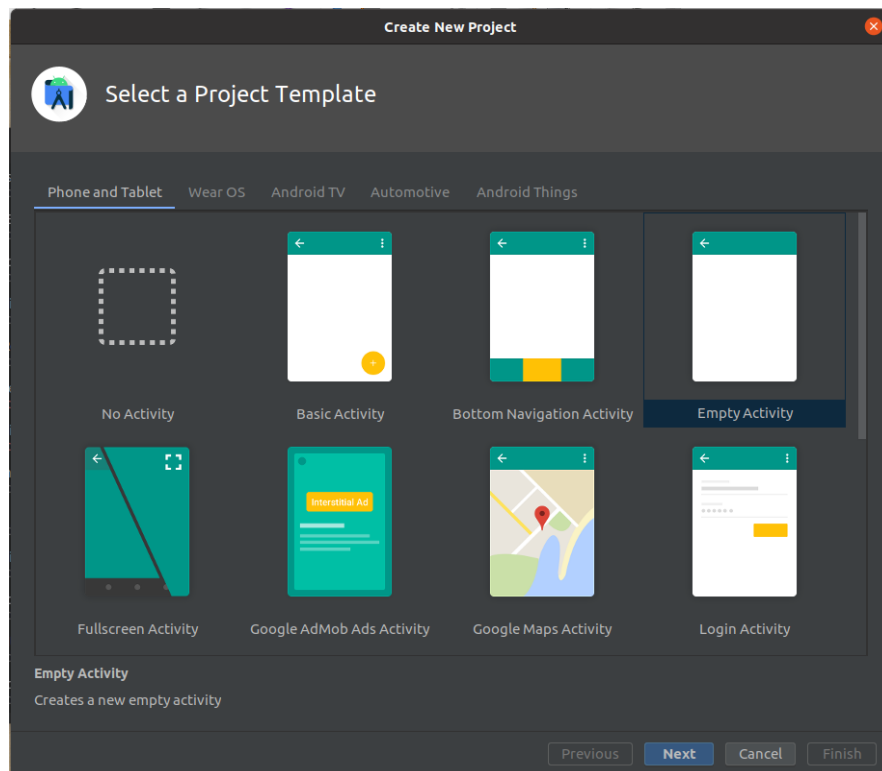
DOWNLOAD ANDROID STUDIO

4.1.2 for Linux 64-bit (882 MiB)

Now let's get started. Open Android Studio and **Create New Project**



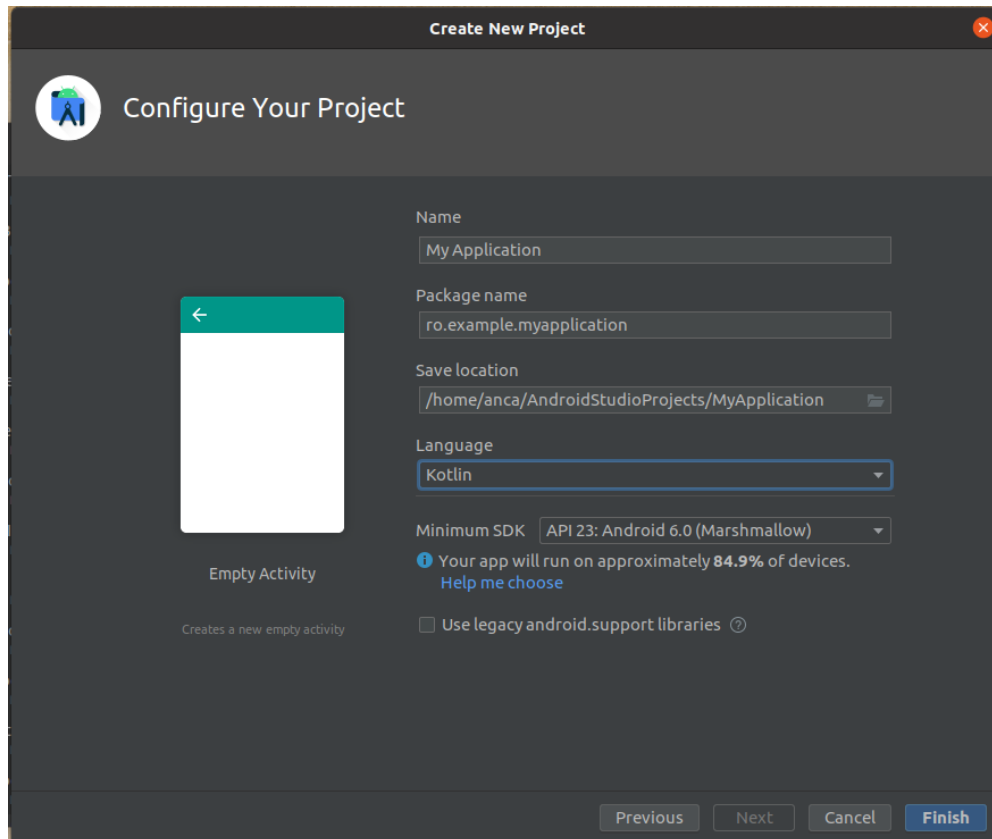
Next you will see that Android Studio gives you a bunch of templates to start your app.



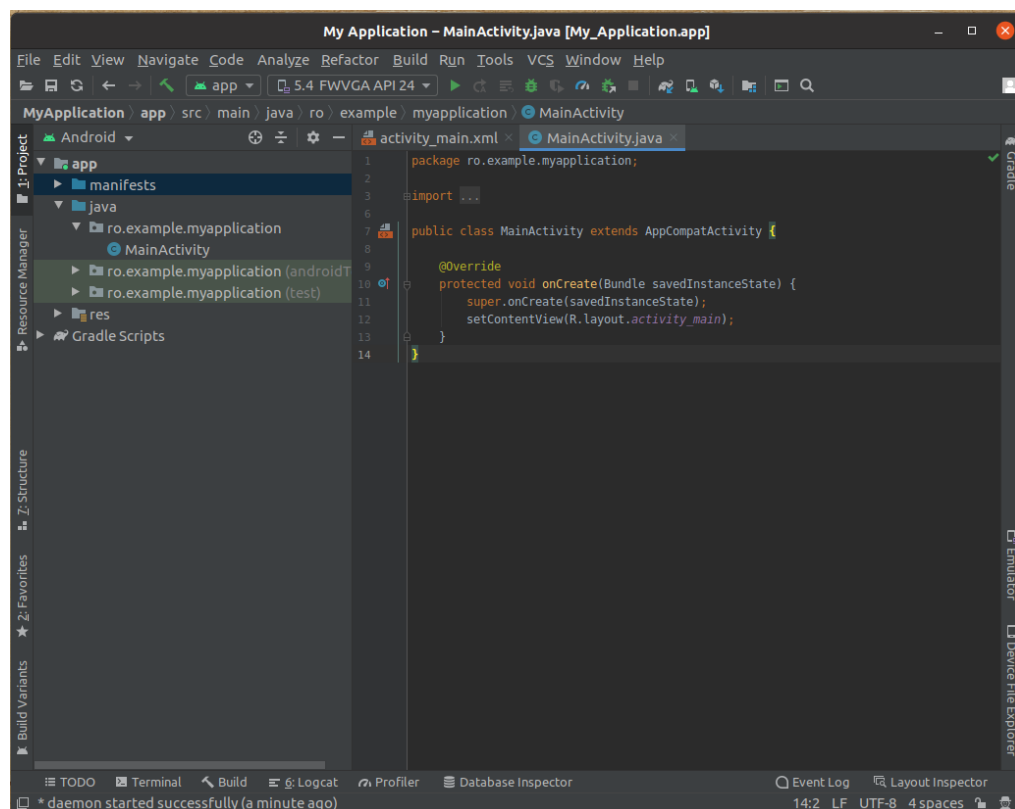
Choose **Empty Activity** for now, then your project needs a configuration

- Set a **Name** to your application
- Enter the **Package name**, [com.example.yourappname]
- If you'd like to place the project in a different folder, change its **Save** location.
- Select either **Java** or **Kotlin** from the **Language** drop-down menu.
- Select the lowest version of Android your app will support in the **Minimum SDK** field.
- Leave the other options as they are.

Click **Finish**.



This is the way your project should look right now: (the Main Window)



## Project file structure:

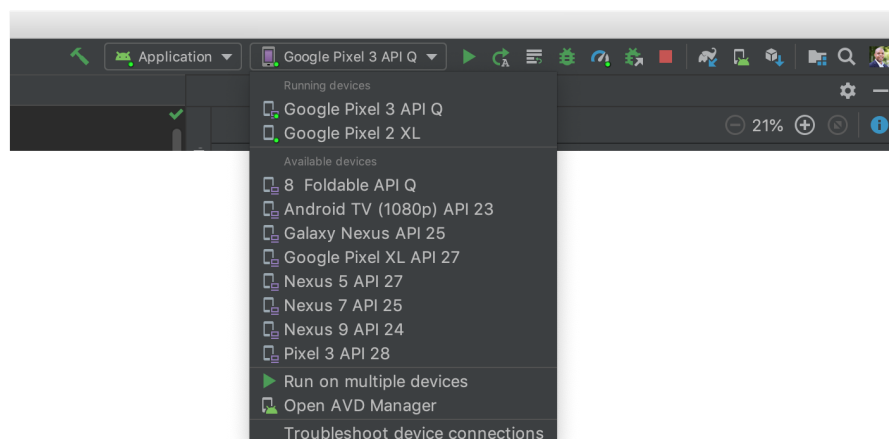
- Manifest (**app > manifests > AndroidManifest.xml**)
  - It contains the structure of your application (activities, services, broadcast receivers and content providers must be declared in here)
  - For more:  
<https://developer.android.com/guide/topics/manifest/manifest-intro>
- Main Activity (**app > java > com.example.myapplication > MainActivity**)
  - The entry point of your app. The system launches an instance of this [Activity](#) and loads its layout.
- Resources (**app > res**)
  - Contains files for each resource your app needs: drawables, layout, values etc.
  - activity\_main.xml is the SML file of your Main Activity user interface (UI), Right now it contains an [TextView](#) element with "Hello World!" text
- Gradle (**Gradle Scripts > build.gradle**)
  - There are two files with this name: Project and Module.
  - Module file contains the dependencies you need for your project and also some details about your app (minSdkVersion, applicationId, versionCode etc)

## Run the app

Now let's run the app for the first time. You can now run the app on a real device or an emulator.

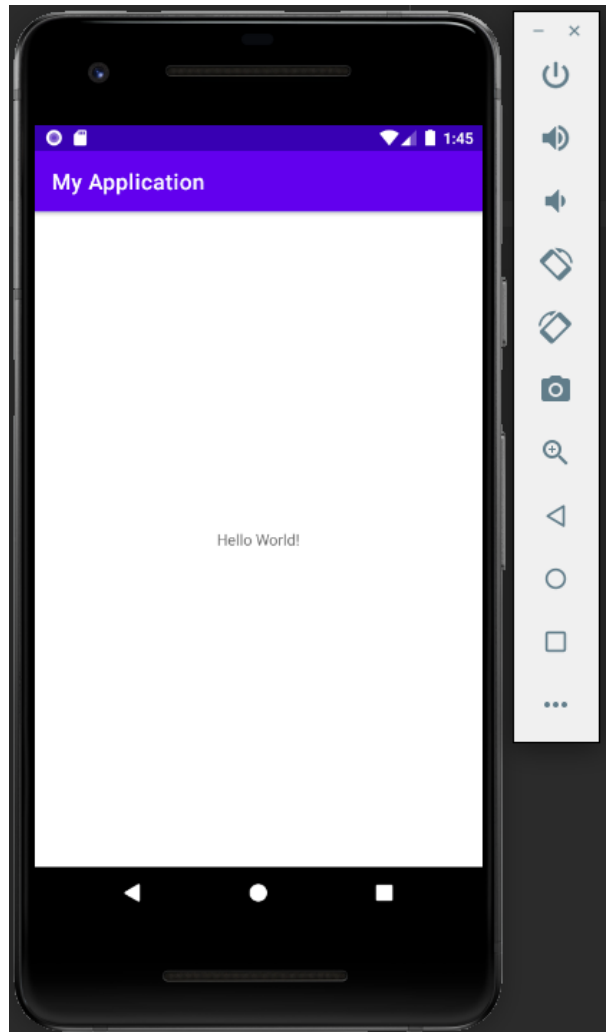
To create an emulator:

1. **Open AVD Manager** and [create an Android Virtual Device \(AVD\)](#).
2. In the toolbar, select your app from the run/debug configurations drop-down menu.
3. From the target device drop-down menu, select the AVD that you want to run your app on.



4. Click **Run**  .

At this point, if your simulator finished starting up, you should be able to see your app:



For more information:

<https://developer.android.com/training/basics/firstapp>

## 2. Activities

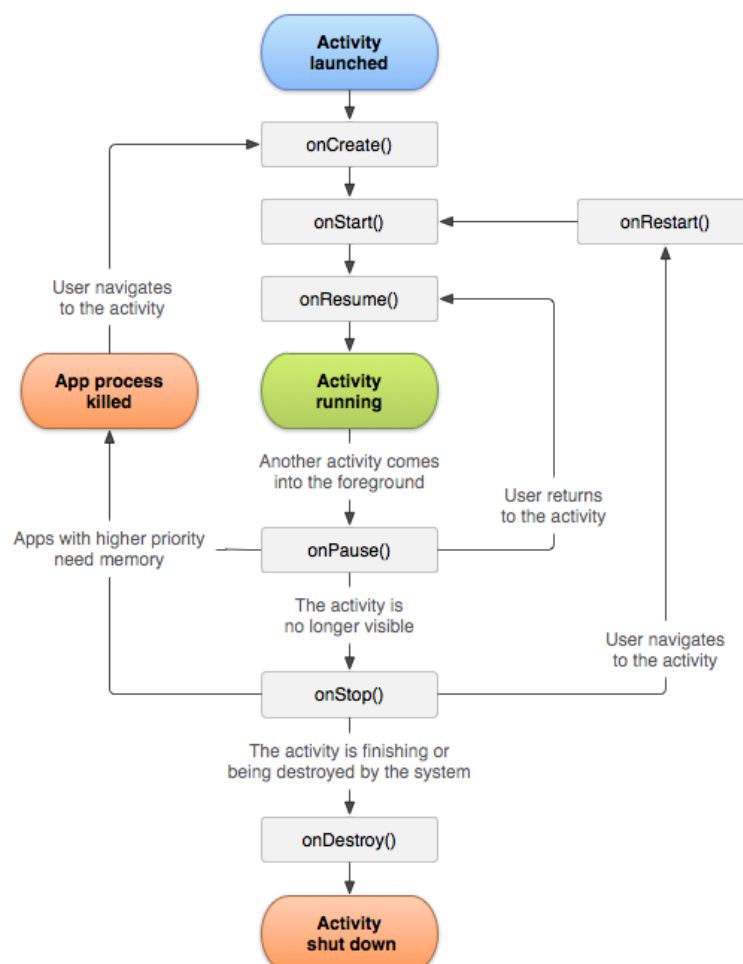
An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface.

Unlike programming paradigms in which apps are launched with a `main()` method, the Android system initiates code in an [Activity](#) instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

For your app to be able to use activities, you must declare the activities, and certain of their attributes, in the manifest. To declare your activity, open your manifest file and add an `<activity>` element as a child of the `<application>` element.

### Activity Lifecycle

Over the course of its lifetime, an activity goes through a number of states. You use a series of callbacks to handle transitions between states.



*For example:* When an activity is first launched, it's *created* and then *started* and then *resumed*. Now the activity is active and the user can interact with it but as soon as the user moves to a different activity or a different application, this activity moves into a *paused* state and then to a *stop* state.

## Activity Lifecycle Callbacks

### **onCreate()**

Your implementation should initialize the essential components of your activity: (create views and bind data to lists). Most importantly, this is where you must call `setContentView()` to define the layout for the activity's user interface.

### **onStart()**

As `onCreate()` exists, the activity becomes visible to the user.

### **onResume()**

The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input.

### **onPause()**

Activity enters a Paused state when it loses focus, this means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

### **onStop()**

The system calls `onStop()` when the activity is no longer visible to the user.

### **onRestart()**

The system invokes this callback when an activity in the Stopped state is about to restart. This callback is always followed by `onStart()`

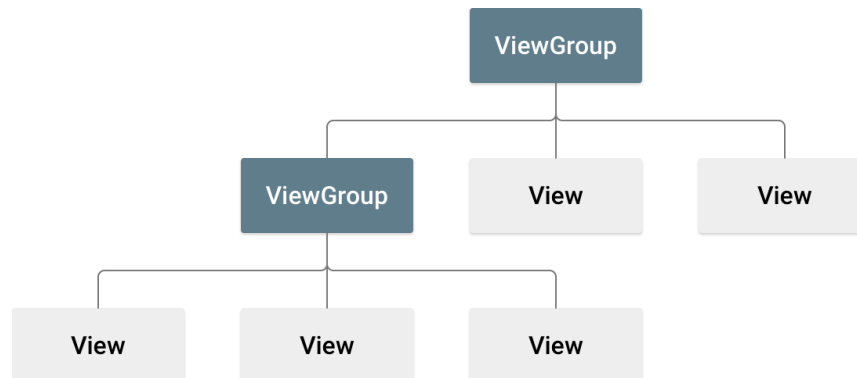
### **onDestroy()**

The system invokes this callback before an activity is destroyed.



### 3. User Interface

The user interface (UI) for an Android app is built as a hierarchy of ViewGroups and Views. ViewGroups are containers that control how their child views are positioned on the screen. And Views are UI components like buttons and text boxes.



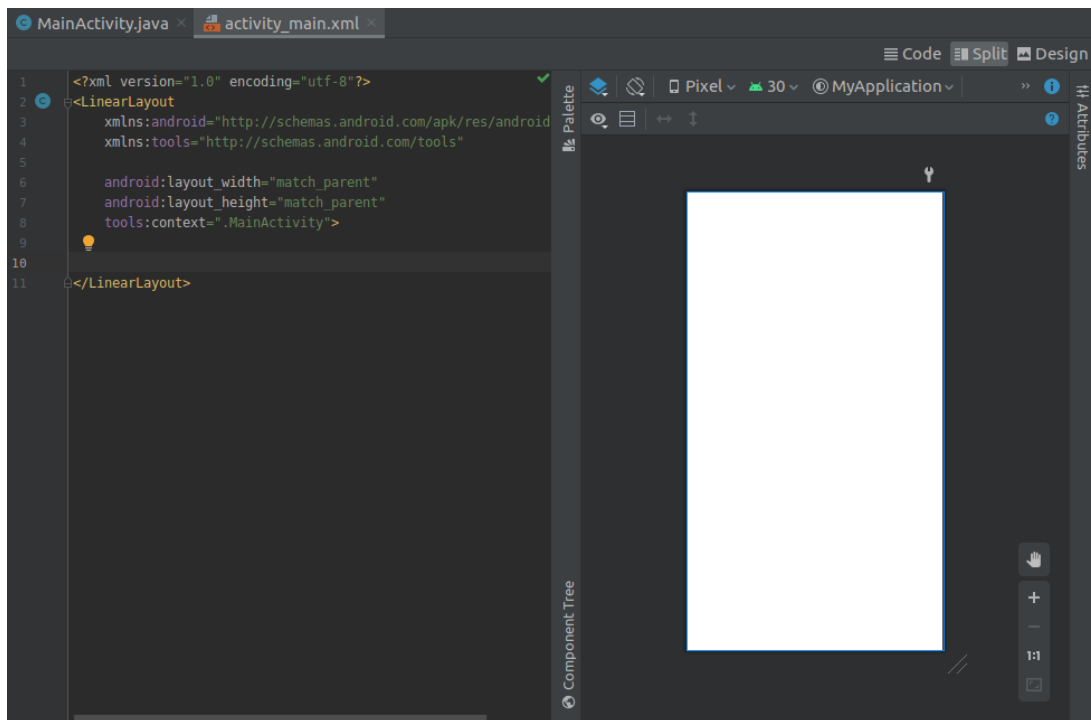
Let's start by opening the activity layout (**app > res > layout > activity\_main.xml**), which is the one Android Studio created from the Empty Activity template.

It contains `ConstraintLayout` as the root tag. `ConstraintLayout` is a `ViewGroup` and it content the `TextView`, at the moment.

For now, change the **`ConstraintLayout`** tag with the **`LinearLayout`**. `LinearLayout` is also a `ViewGroup` and is more simple to use, does not give you the same freedom as the `ConstraintLayout` but is very good for the beginning.

`LinearLayout` is a view group that aligns all children in a single direction, vertically or horizontally. You can specify the layout direction with the `android:orientation` attribute.

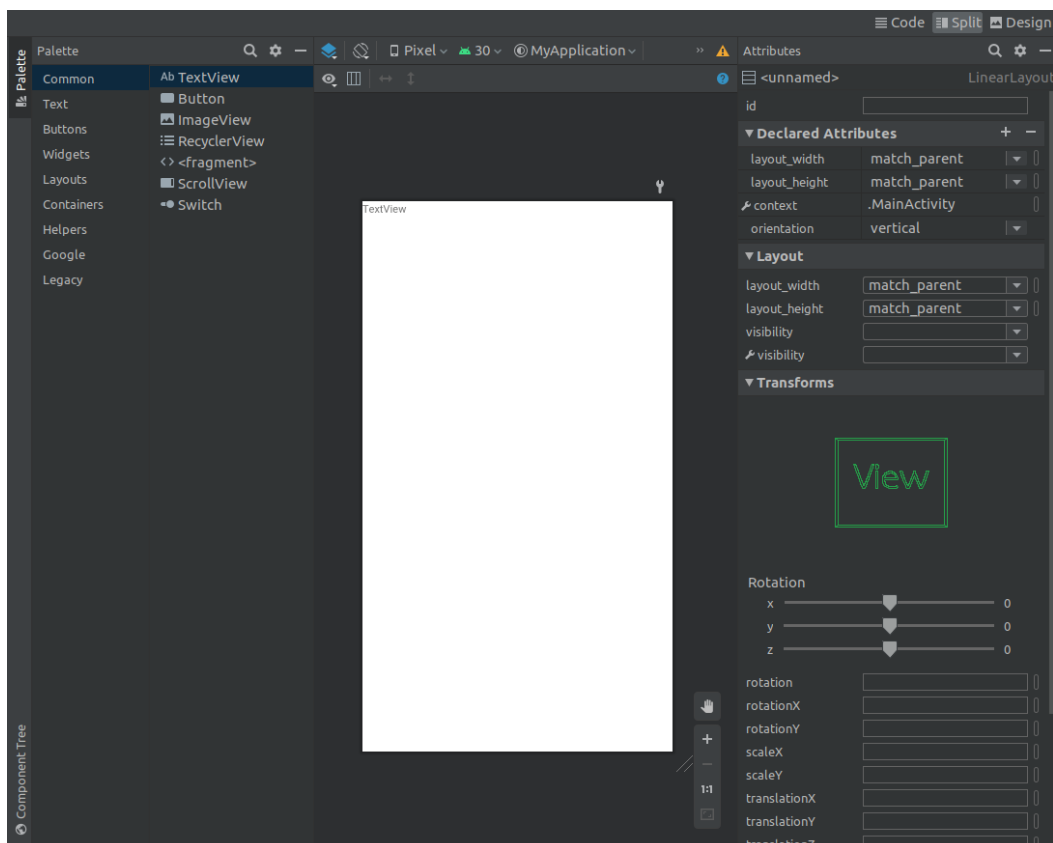
All children of a `LinearLayout` are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). A `LinearLayout` respects *margins* between children and the *gravity* (right, center, or left alignment) of each child.



This is how your screen should look like in the Split mode (right corner).

Clicking on the **Palette** button will reveal lots of Views and ViewGroups. You can drag and drop any of those elements into your design and Android Studio will add it to your xml.

Try and add a `TextView` to your layout. On the right side of the design screen you can open the **Attributes** screen that holds all the attributes of the selected View or ViewGroup.



Your TextView right now has the width as it's parent and the height as it needs to hold the text.

Now you can either work in the xml file or change the attributes in the Attributes screen. Change the text of the TextView to "Hello World! And try to align it in the middle of its parent.

There are two type of gravity:

- android:gravity (center the content in the View)
- android:layout\_gravity (center the View in the ViewGroup)


To understand this fully, change the width of the TextView to 100dp (*density-independent pixels*) and apply these two attributes one after the other to see how it affects your layout.

However, none of these center your text in the center of the screen, it only center vertically. This is caused by the LinearLayout, which gives your View only a row in the layout. So if you want your text to be centered in the middle, you either make its height to *match\_parent*, or tell the LinearLayout to center its children by adding the gravity to its attributes.



## Strings

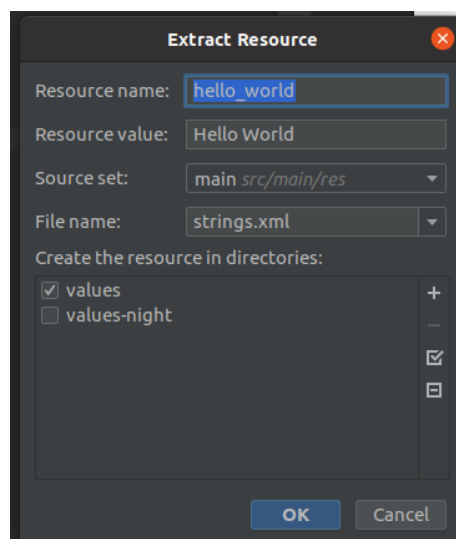
Once you are back in the layout, you can see the text attribute has a warning on it, which says your text should not be hardcoded and you should use string resources.



```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="100dp"
    android:text="Hello World" />
```

You have two options:

1. Go to **app > res > values > strings.xml** and open it  
This is the file you should keep all your strings in a key - value manner
2. Click on the warning bulb (or Alt+Enter) and a popup will appear so you can complete the key for your string and it will automatically be added to the strings.xml file



Now your text attribute will hold a reference to the text in the strings file.

```
android:text="@string/hello_world"
```

## Actions

Let's make the app interactive and add an EditText and a Button under the TextView.

```
<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:orientation="vertical"
    tools:context=".MainActivity">

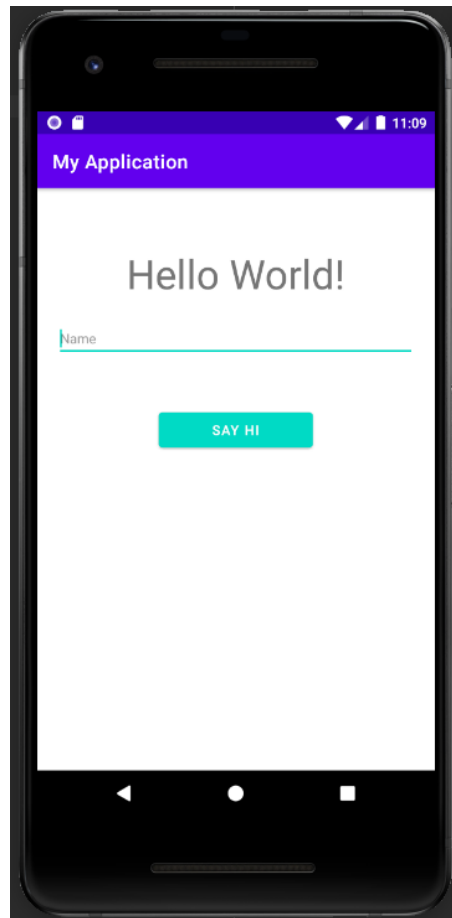
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="42sp"
        android:layout_gravity="center"
        android:layout_marginTop="60dp"
        android:text="@string/hello_world" />

    <androidx.appcompat.widget.AppCompatEditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="40dp"

        android:hint="@string/name"
        android:textColor="@color/black"
        android:inputType="text"
        android:textSize="14sp"
        android:layout_margin="20dp" />

    <Button
        android:id="@+id/button"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="30dp"
        android:layout_gravity="center"
        android:backgroundTint="@color/teal_200"
        android:text="@string/say_hi"
        />

</LinearLayout>
```



This is how your app looks right now. In order to get the text the user writes in the edit text and show it when the button is pressed, we need to bind the Views in the xml with the activity where you can apply logic on them.

First go back to the MainActivity. Then declare your views as fields in your class.

```
private TextView textView;  
private Button button;  
private AppCompatActivity editText;
```

Next you need to bind the fields with the corresponding view or view group in the layout. You do that with the help of the id, which is unique in one layout.

To let the app know that the user has clicked the button you need to attach to the button an OnClickListener. This is an interface with an onClick() function, we override it in order to let us know when the onClick is called.

```

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

    textView = findViewById(R.id.textView);
    button = findViewById(R.id.button);
    editText = findViewById(R.id.editText);

    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            String text = "";
            if(editText.getText() != null)
                text = editText.getText().toString();

            textView.setText(getString(R.string.hello) + text);
        }
    });
}

```

Additionally, after Java 8 you can use lambda expressions to ease your work and make the code more light. The lambda expression in this case will replace the anonymous class declaration of the click listener.

```

button.setOnClickListener( view -> {
    String text = "";
    if(editText.getText() != null)
        text = editText.getText().toString();

    textView.setText(getString(R.string.hello) + text);
});

```

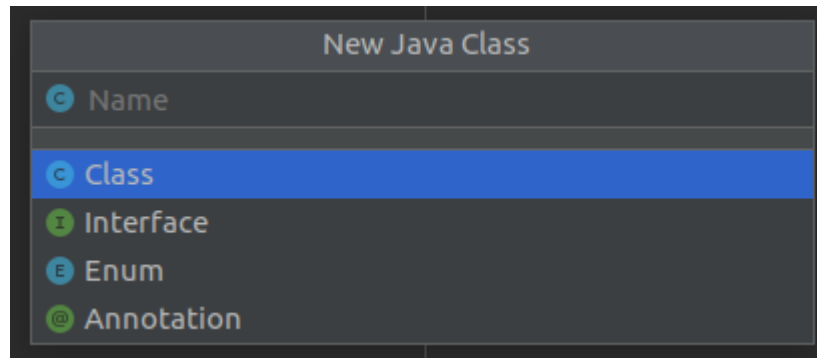
The variable **view** is the same as the parameter in the onClick method. Looks more pretty now and you don't have to worry about knowing to do it from the beginning. If you write the version with the OnClickListener interface, Android Studio will suggest to transform into lambda, Alt + Enter and it will do the magic for you.

All you have to do now is run the app and see how it works.

## 4. Screen navigation

What if you want to say hi to the inserted name in another screen and let the user do other stuff in that screen?

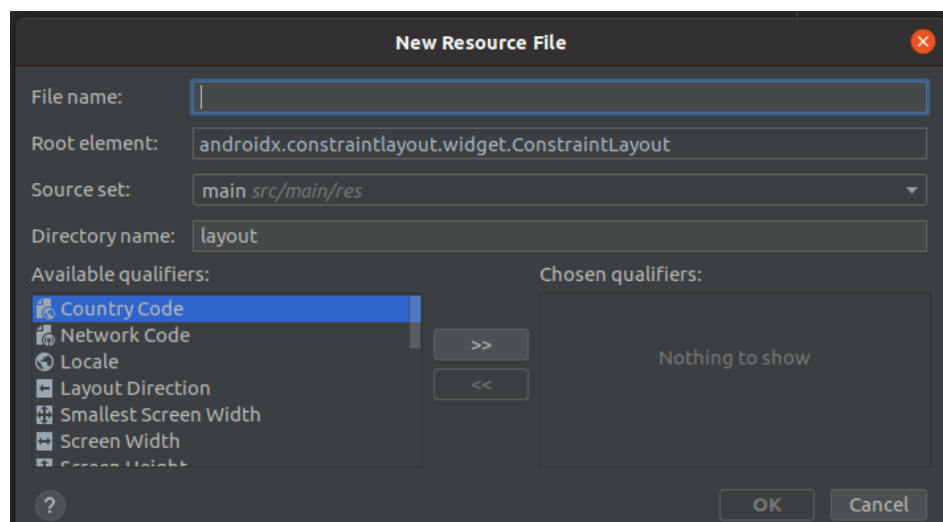
First of all you need to create another Activity. Right click on the package folder (ro.example.myapplication) and click **New > Java class**.



For example, SecondActivity, select Class and press Enter. You can see a new class was created. In order to make the class an Activity, first we need to extend **AppCompatActivity**, the base class for activities and second we have to declare it in the manifest.

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

Next step is to add the second activity layout in the layout folder. Right click on the mouse and choose **New > Layout Resource File**.



Give the file the activity name in the reverse order (activity\_second), so you can keep track of the files in an easier manner.

You can also choose another root element, or simply change it in the xml after the file is created then press ok. In the new layout add a TextView with Hello message.



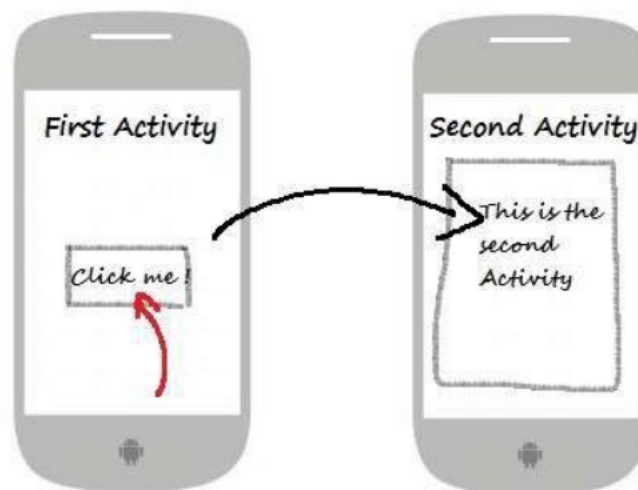
Go back to the SecondActivity and override the onCreate() method in order to bind the layout to the activity with setContentView()

```
setContentView(R.layout.activity_second);
```

You have two activities and in order to start an activity from another you need an Intent.

## Intents

An Intent contains an action carrying some information. Intents allow you to interact with components that belong either to the same application or outside that application.



Intent contains the following:

- **Action** (mandatory) to be performed as ACTION\_VIEW, ACTION\_DIAL etc
- **Data** to operate on, such as a person record in the contacts database, expressed as a Uri. (Uniform Resource Identifier)
- **Category** gives additional information about the action to execute. For example CATEGORY\_LAUNCHER means it should appear in the Launcher as a top level application
- **Extras** is the **key value pair** (a Bundle) for additional information that needs delivered to the component handling the intent.
- **Flags** instruct the Android system to launch an activity in a specific manner.

Intent types:

- Explicit
  - It is an intent where you explicitly define the component that needs to be called by the Android System. Normally, Explicit intents are used to start components within your application.

```
private void gotoSecondApp(){  
    Intent intent = new Intent(getApplicationContext(), SecondActivity.class);  
    startActivity(intent);  
}
```

- Implicit
  - It's an intent where instead of defining components, you define the action you want to perform for different activities.

```
Intent i = new Intent();  
i.setAction(Intent.ACTION_SEND);
```

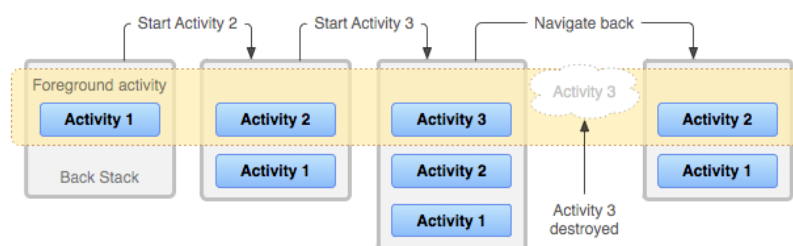
More information about intents:

<https://developer.android.com/reference/android/content/Intent>

## Back Stack

Activities are arranged in a stack (the *back stack*) in the order in which each activity is opened.

When the current activity starts another, the new activity is pushed on the top of the stack and takes focus. The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface. When the user presses the **Back** button, the current activity is popped from the top of the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored). Activities in the stack are never rearranged, only pushed and popped from the stack—pushed onto the stack when started by the current activity and popped off when the user leaves it using the **Back** button. As such, the back stack operates as a "last in, first out" object structure. Figure 1 visualizes this behavior with a timeline showing the progress between activities along with the current back stack at each point in time.



<https://developer.android.com/guide/components/activities/tasks-and-back-stack>

Back to the app, what you need to do now is declare an explicit intent, because you need to open a component from your application and start the activity with that intent.

Insert the `gotoSecondApp()` method in your main activity and call it from the `onClick` method in the click listener.

Run the application, press the Button in the first screen and it will lead you to the next one.

But it would be better if you would say Hello, user's name in the second screen. To do that you need to pass through the intent, the text the user inserted in the edit text field.

## Intent Extras

### 1. Add extra to the intent

```
intent.putExtra(EXTRA_MESSAGE, extra);
```

- EXTRA\_MESSAGE is a public constant key because the next activity uses the key to retrieve the text value.

```
public static final String EXTRA_MESSAGE = "MESSAGE";
```

- extra is the text of the edit text view, send as parameter to the `gotoSecondApp()` method

```
public class MainActivity extends AppCompatActivity {

    public static final String EXTRA_MESSAGE = "MESSAGE";
    private TextView textView;
    private Button button;
    private AppCompatActivity editText;

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

        textView = findViewById(R.id.textview);
        button = findViewById(R.id.button);
        editText = findViewById(R.id.editText);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String text = "";
                if(editText.getText() != null)
                    text = editText.getText().toString();

                gotoSecondApp(text);
            }
        });
    }

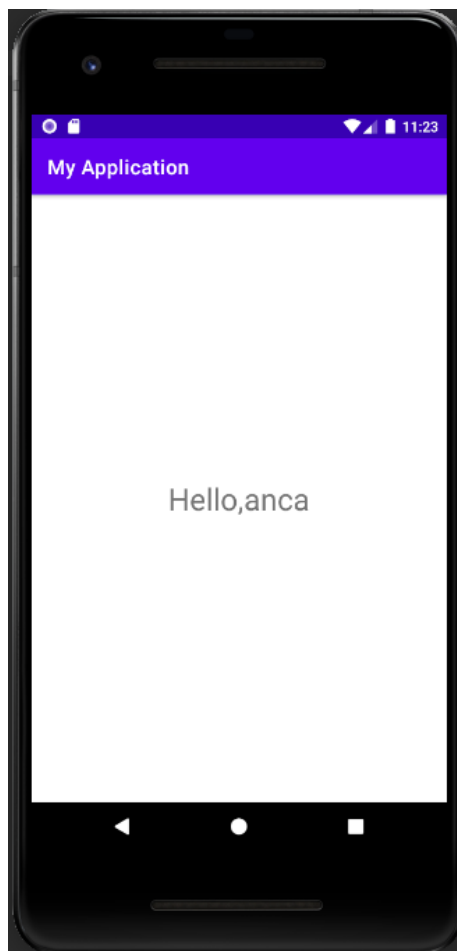
    private void gotoSecondApp(String extra){
        Intent intent = new Intent(getApplicationContext(), SecondActivity.class);
        intent.putExtra(EXTRA_MESSAGE, extra);
        startActivity(intent);
    }
}
```

To retrieve and show the data in the second activity:

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

    // Get the Intent that started this activity and extract the string
    Intent intent = getIntent();
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);

    // Capture the layout's TextView and set the string as its text
    TextView textView = findViewById(R.id.textView);
    textView.setText(getString(R.string.hello) + message);
}
```



## 2. Add a Bundle

The difference from the previous method is that this time you put the text in the bundle as a key-value element and the bundle will be the extra component of the intent.

This method is more recommended because Bundle class is very optimised.

```
private void gotoSecondApp2(String extra){
    Intent intent = new Intent(this, SecondActivity.class);

    Bundle mBundle = new Bundle();
    mBundle.putString(EXTRA_MESSAGE, extra);
    intent.putExtras(mBundle);
}
```

## Context

If you take a closer look at the Intent constructor's parameters you will see that the last one is the destination class, but what is the first one?

Context is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.

Here are most used function for retrieving the Context:

## The "this" keyword

The "**this**" keyword in general sense refers to the current class instance. So, when used inside an Activity it refers to that Activity instance. And as Activity is a subclass of "Context", you will get context of that activity.

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

Some functionalities the Activity's context provides are

- Loading resource values
- Layout inflation
- Starting an activity
- Showing a dialog
- Etc

## View.getContext()

This method can be called on a View like textView.getContext(). This will give the context of activity in which the view is currently hosted in.

```
editText.getContext();
```

## **getApplicationContext()**

This method returns the Context which is linked to Application.

Activity's context will be destroyed when the activity is destroyed and the application's context will be destroyed only after the app is completely closed. This means that you shouldn't use it to Inflate a Layout, Start an Activity nor Show a Dialog, otherwise they can cause memory leaks.

## **getBaseContext()**

It is related to ContextWrapper, which is created around existing Context and lets you change its behavior. With *getBaseContext()* we can fetch the existing Context inside ContextWrapper class.

## Simple Dialog

Let's go back to the app now. You made your app to go to another screen where a Hello text is displayed followed by the user name.

But If the user does not complete any text in the EditText in the first screen, the second screen will say Hello, blank.

In order to prevent that from happening you can put a constraint on the click listener, if only the edit text is not empty then go to the next screen, otherwise let the user know he should complete the edit text.

You can do that in many ways.

## Hidden TextView

You can make a **TextView** under the EditText with the id `errorText`. When needed you can make the TextView visible and set with an error message.

## Toast

A toast is a view containing a quick little message for the user. When the view is shown to the user, it appears as a floating view over the application.

```
Toast.makeText(this, getString(R.string.error_msg), Toast.LENGTH_LONG).show();
```

# Hello World!

SAY HI

Name field must be completed

## Alert Dialog

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 events that require users to take an action before they can proceed.

An Alert Dialog is a subclass of Dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout. It is created with the Builder pattern

```
new AlertDialog.Builder(this)
    .setTitle("Error")
    .setMessage(R.string.error_msg)

// Specifying a listener allows you to take an action before dismissing the dialog.
// The dialog is automatically dismissed when a dialog button is clicked.
.setPositiveButton(android.R.string.yes, new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        dialog.dismiss();
    }
})

// A null listener allows the button to dismiss the dialog and take no further action.
// .setNegativeButton(android.R.string.no, null)
.setIcon(android.R.drawable.ic_dialog_alert)
.show();
```

