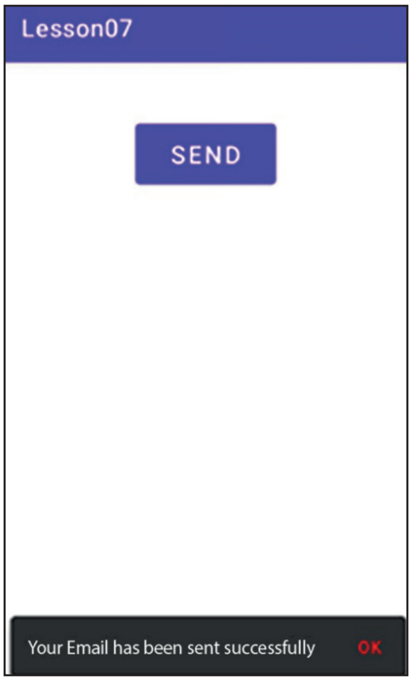
Lesson 7: Snackbar, Activities, Android Intent, Alert Dialog and Android Notifications

Snackbar Class

The Snackbar class provides a lightweight feedback about an operation. Snackbars show a brief message at the bottom of the screen on smart devices and lower left on larger devices. They appear above all other elements on the screen but only one can be displayed at a time.

Snackbars automatically disappear after a timeout or after user interact elsewhere on the screen, particularly after interactions that summon a new surface or activity. Snackbars can be swiped off the screen.

Example:
In this example, you will create an app interface that includes a button, when the app user taps this button, a Snackbar notification message will show at the bottom of the phone emulator screen, as shown in the following screen capture:



Your Email has been sent successfully

To configure a Snackbar class within your app, follow the example below:

1- Open Android Studio, and then click **File** → **New** → **New Project**

2- Select **Empty Activity**, and click **Next**

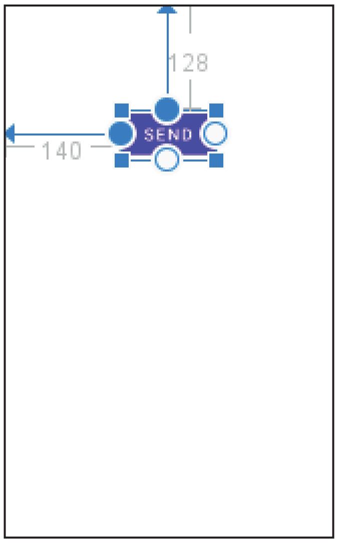
3- Type: **Lesson07** for the application name, then click **Finish**.

4- Open the **activity_main.xml** file in Design mode and then **delete** the “Hello World!” text.

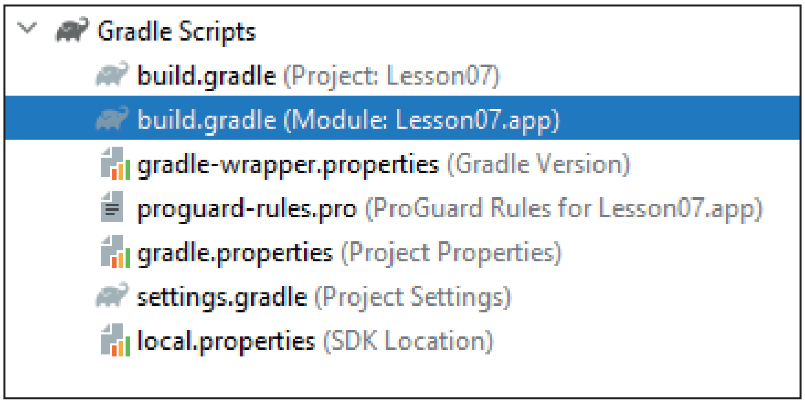
5- Add a **Button** widget to the **activity_main.xml** file using drag and drop technique, set its constraints (margins) and configure its attributes’ values as follows:

ID: sendMessage	**text** : Send	**textSize**: 20sp

The following figure displays your activity in the design mode:



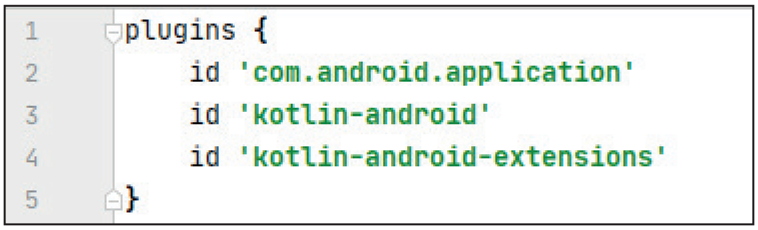
6- Before you start with typing the Kotlin code in your app, check the **build.gardle (Module:Lesson07.app)** file content as illustrated in the figure below:



```
Gradle Scripts
    build.gradle (Project: Lesson07)
    build.gradle (Module: Lesson07.app)
    gradle-wrapper.properties (Gradle Version)
    proguard-rules.pro (ProGuard Rules for Lesson07.app)
    gradle.properties (Project Properties)
    settings.gradle (Project Settings)
    local.properties (SDK Location)
```

Be sure it has the Kotlin plugin. If not, add the following code: **id 'kotlin-android-extensions'**

Click : **Sync Now**. The configuration should be as follows:

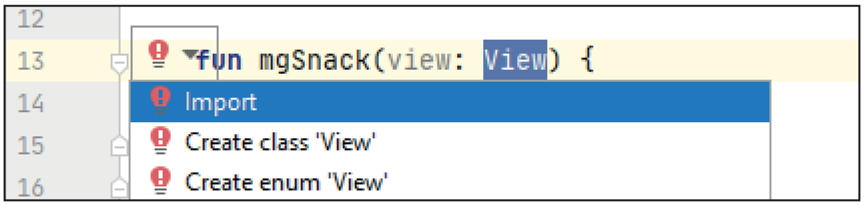


```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-android-extensions'
}
```

7- Open the **MainActivity** file and add : **MgSnack** function as follows:

```
package com.androidatc.lesson07
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity

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

    }

    fun MgSnack(view: View) {

    }
}
```

The **`View`** class in the code above has a red color. You need to import the prerequisite classes for this class. To do this, double click the **`View`** class, click the red pop-up lamp, and click **Import**.



```
fun mgSnack(view: View) {
```

Note: The Android View class is the basic building block of an Android user interface. A View occupies a rectangular area on the screen to draw itself and its children components.

8- Add the **Snackbar** class configuration as follows:

```
fun mgSnack(view: View) {
    Snackbar.make(findViewById(R.id.sendMessage),
     "Your Email has been sent successfully",Snackbar.LENGTH_
LONG).show()
}
```

Double click the **Snackbar** class, click the red pop-up lamp, then click **Import.**

Here, **`make`** is a static method of the class Snackbar. This method has three parameters which are as follows:

a- `findViewById(R.id.sendMessage)`: the button which will generate this Snackbar message.

b- A string: is the message to be displayed in the Snackbar.

c- `LENGTH_LONG`: represents for how long the message will appear. By default, the duration is 3.5 seconds, but if you replace `LENGTH_LONG` with `LENGTH_SHORT` the message will remain for 2 seconds.

9- The full code of the **MainActivity.kt** is as follows:

```
package com.androidatc.lesson07
import android.os.Bundle
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.snackbar.Snackbar

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

    fun mgSnack(view: View) {
        Snackbar.make(
            findViewById(R.id.sendMessage),
            "Your Email has been sent successfully",Snackbar.
LENGTH_LONG).show()
    }
}
```

10- Open **activity_main.xml** file, and add the following attribute to the button view:

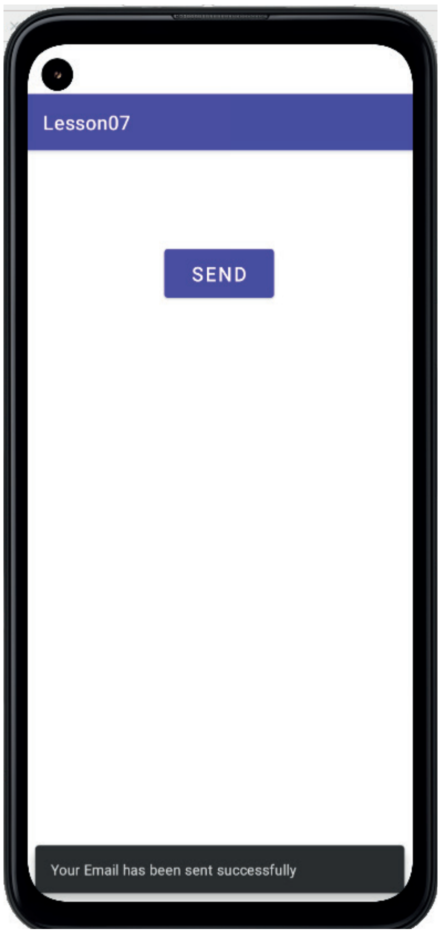
```
android:onClick="mgSnack"
```

This means when your app users tap this button, they will call the **mgSnack** function to run.

The full XML code of this button is as follows:

```
<Button
    android:id="@+id/sendMessage"
    android:layout_width="112dp"
    android:layout_height="60dp"
    android:text="Send"
    android:textSize="20sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.498"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.135"
    android:onClick="mgSnack"
    />
```

11- Run your app. When you tap the **SEND** button, you will get the Snackbar notification message at the bottom of your device screen for a short time (about 3.5 seconds) as illustrated in the figure below:



Your Email has been sent successfully

12- You may add other parameters to your previous **Snackbar** class to add another text to this bar. You may configure this text later to take specific action if your app user taps it. In this example, you will add a text: **OK**. When your app user taps it, the Snackbar notification message will disappear. The code is as follows:

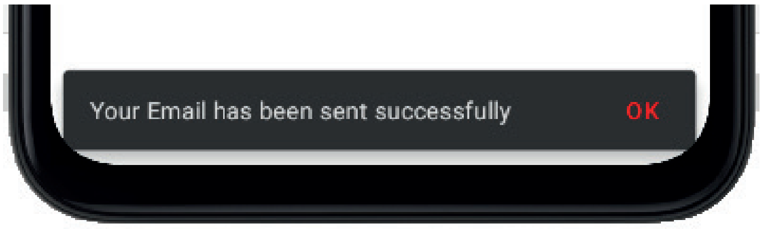
```
fun mgSnack(view: View) {
    Snackbar.make(
        findViewById(R.id.sendMessage),
        "Your Email has been sent successfully",Snackbar.LENGTH_LONG)
        .setAction("OK"){}.setActionTextColor(Color.RED).show()
}
```

Double click the **Color** class, click the red pop-up lamp, and select **Import**.

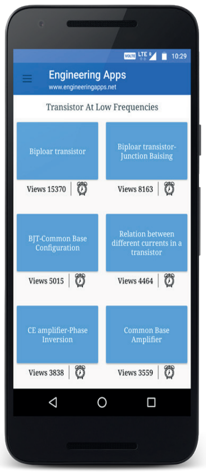
The full code of the **MainActivity** file is as follows:

```
package com.androidatc.lesson07
import android.graphics.Color
import android.os.Bundle
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.snackbar.Snackbar

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

    }

    fun mgSnack(view: View) {
        Snackbar.make(
            findViewById(R.id.sendMessage),
            "Your Email has been sent successfully",Snackbar.
LENGTH_LONG)
            .setAction("OK"){}.setActionTextColor(Color.RED).show()
    }
}
```

13- **Stop**, then **Run** your app. Click the **SEND** button. You should get the following figure:


Your Email has been sent successfully OK

What is an Activity?


Engineering Apps
www.engineeringapps.net
Transistor At Low Frequencies
Biploar transistor
Biploar transistor-Junction Baising
Views 15370
Views 8163
BJT-Common Base Configuration
Relation between different currents in a transistor
Views 5015
Views 4464
CE amplifier-Phase Inversion
Common Base Amplifier
Views 3838
Views 3559

An activity is the entry point for interacting with the user. It represents a single screen with a user interface. The app consists of one or more activities and the user can navigate between them through buttons, menus, images, the back button and other navigation tools.

The Activity class is designed to facilitate this paradigm. This class includes all methods, variables and other classes which operate together to produce a specific function.

An activity provides the window in which the app draws its user interface. This window typically fills the screen, but it may also be smaller than the screen and float on top of other windows. Generally, one activity implements one screen in an app. For instance, one of the activities of an app may implement a *Preferences* screen, while another activity may implement a *Select Photo* screen.

Most apps contain multiple screens since they comprise multiple activities. Typically, one activity in an app is specified as the **main activity** which is the first screen to appear when the user launches the application and it usually includes connections (intent) to other activities. Each activity can then initiate another in order to perform various actions. The main activity is similar to the home page of a web site which opens when you enter the URL in any web browser. However, as web servers can be configured to select which web page is to be the home page by naming it for example index.html, an activity in Android apps~~,~~can be configured to be the main activity by configuring the **AndroidManifest.xml** file.

For example, the following **AndroidManifest.xml** file, includes two activities (MainActivity and Second). Each one is represented by an activity tag and it includes the **name** attribute which declares its name. The two activities' tags have a gray highlighted color in the code below and each tag is closed by `</activity>` tag.

```
<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">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

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

If you move (cut and paste) the code below from the **MainActivity** to the **Second** activity, then run your app, the main activity (launcher activity) will be the **Second** activity.

```
<intent-filter>
      <action android:name="android.intent.action.MAIN" />

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

Now, we will discuss another issue which is related to the app activities too. In this point, if you open the **MainActivity** file, you will see the following code about your activity:

```
class MainActivity : AppCompatActivity() {
    . . . . . .
    . . . . . .
    }
```

This code declares your activity **MainActivity** as a child class of the parent class **AppCompatActivity** which contains methods and part of Android Studio support. Your activity **MainActivity** will inherit all the methods and characteristics (parameters) of the parent class **AppCompatActivity**.

All the older versions of the support libraries have been replaced with **AppCompatActivity**, which will be used to cover everything that is in AppCompat theme.

Activity Lifecycle

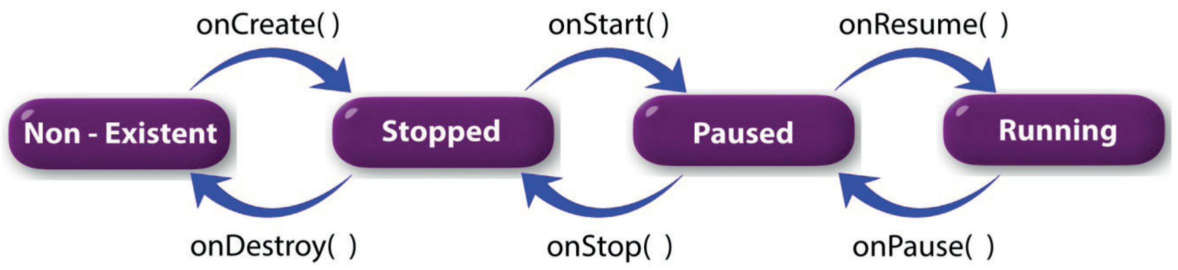
Activities in the system are managed as *activity stack*. When the user touches an icon in the app launcher (or a shortcut on the Home screen), that app's task comes to the foreground. If no task exists for the app (the app has not been used recently), then a new task is created and the "main" activity for that app opens as the root activity in the stack.

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.

An activity has essentially four states:

1) If an activity is in the foreground of the screen (at the top of the stack), then it is active or **running**.
2) If an activity has lost focus but is still visible (that is, a new non-full-sized or transparent activity has focus 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 it can be killed by the system in extreme low memory situations.
3) If an activity is completely obscured by another activity, it is **stopped**. It still retains all state and member information; however, it is no longer visible to the user because its window is hidden and it will often be killed by the system when memory is needed elsewhere.
4) 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 figure represents the callback methods that you can implement to perform operations when the Activity moves between states:



Managing the activity lifecycle

Over the course of its lifetime, an activity goes through a number of states. You may use a series of callbacks to handle transitions between states. The following sections introduce these callbacks.

`onCreate()`

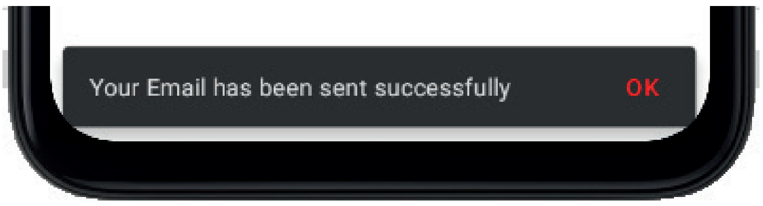
You must implement this callback method, which fires when the system creates your activity. Your implementation should initialize the essential components of your activity.

For example, the following codes of the MainActivity (MainActivity.kt) are declared in the first line. MainActivity is a class that inherits all its characteristics from the parent class AppCompatActivity (super). Your app should create Views and bind data to lists. Most importantly is where the **`onCreate()`** method calls **`setContentView()`** to define the layout for the activity's user interface which is **activity_main.xml**. Remember, **`R.layout`** means : res → layout

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

Let us comeback to the last or the current Android project for this lesson (Lesson07) to test how `onCreate()` and other activity lifecycle methods work. To do that, follow the steps below:

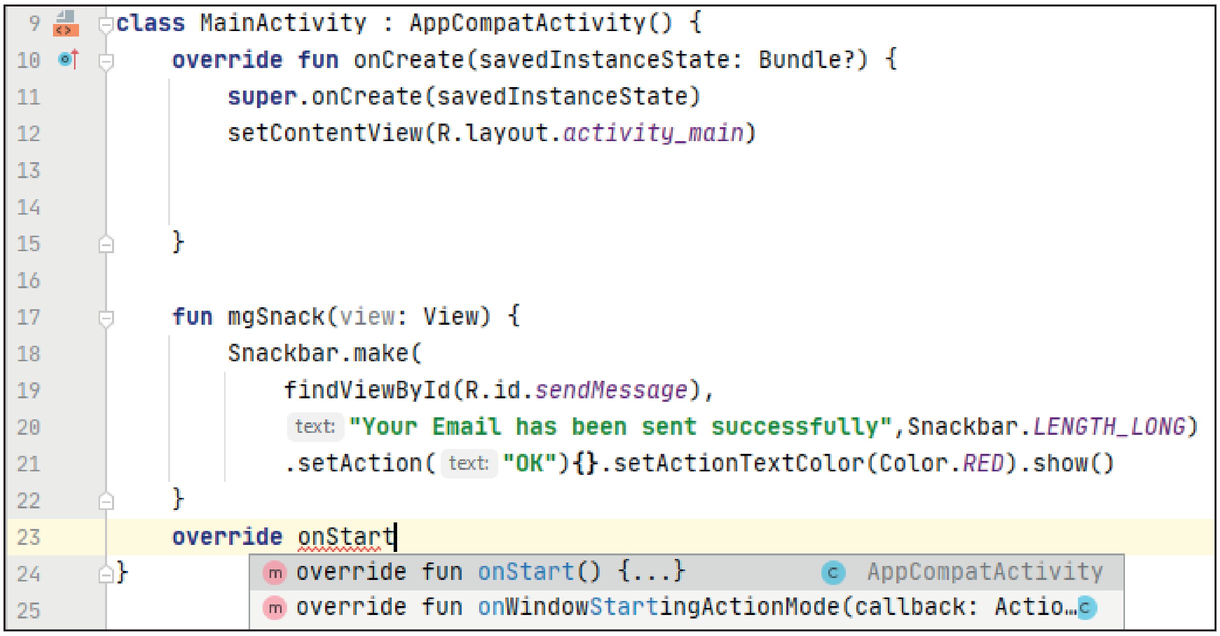
1- Open the **MainActivity** file, **Stop** then **Run** your app, and then tap the **Send** button. Note that when your app starts, **`onCreate()`** method will initialize the essential components of your activity and the Snackbar notification message will appear as illustrated in the following figure:



When `onCreate()` finishes, the next callback is always `onStart()`.

`onStart()` Method:
As `onCreate()` method exits, the activity enters the Started state and becomes visible to the user. This callback method contains the number of times of the activity's final preparations of coming to the foreground and becoming interactive.

2- Continue with the same Android project and follow the steps below:
Add the `onStart()` method the same previous code directly after the ending of the Snackbar class as illustrated in the following figure:



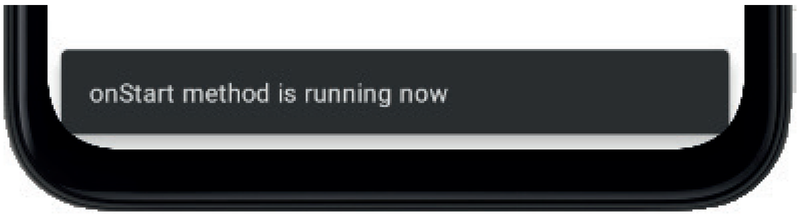
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)


    }

    fun mgSnack(view: View) {
        Snackbar.make(
            findViewById(R.id.sendMessage),
            text: "Your Email has been sent successfully",Snackbar.LENGTH_LONG)
            .setAction( text: "OK"){}.setActionTextColor(Color.RED).show()
    }
    override onStart
}
```

m override fun onStart() {...}
c AppCompatActivity
m override fun onWindowStartingActionMode(callback: Actio…

Add the following Snackbar configuration to your `onStart()` method:

```
override fun onStart() {
    super.onStart()
    Snackbar.make(
        findViewById(R.id.sendMessage),
        "onStart method is running now",Snackbar.LENGTH_LONG).show()

}
```

3- **Stop**, and then **Run** your app. You should get the following run result without tapping or clicking any button:


onStart method is running now

`onResume()` Method

The system invokes this callback method right before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack where it captures all the user's input. Most of an app's core functionality is implemented in the `onResume()` method.
The `onPause()` callback always follows `onResume()`.

`onPause()` Method

The system calls `onPause()` method when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the **Back** or **Recent** buttons. When the system calls `onPause()` for your activity, it technically means that your activity is still partially visible, but most often it is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

An activity in the Paused state may continue to update the UI if the user is expecting the UI to be updated.

Examples of such an activity include one showing a navigation map screen or a media player playing. Even if such activities lose focus, the user expects their UI to continue updating.

Android allows several apps to share the screen at once. For example, a user could split the screen, viewing a web page on the left side while composing an email on the right side. The user experience depends on the version of the Android OS and the kind of device. If the user performs a long press on the Overview button, the device puts the current activity in multi-window mode and opens the Overview screen to let the user choose another activity to share the screen.

Multi-window mode does not change the activity lifecycle.

When the app is in multi-window mode the resumed state will depend on the Android version of the device. For example, in Android 9 (API level 28) and lower, only the activity with focus is in the RESUMED state; all others are PAUSED. When there are multiple activities within a single app process, the activity with the highest Z-order is RESUMED and the others are PAUSED. While in Android 10 (API level 29) and higher, all visible, top-focusable Activities are RESUMED.

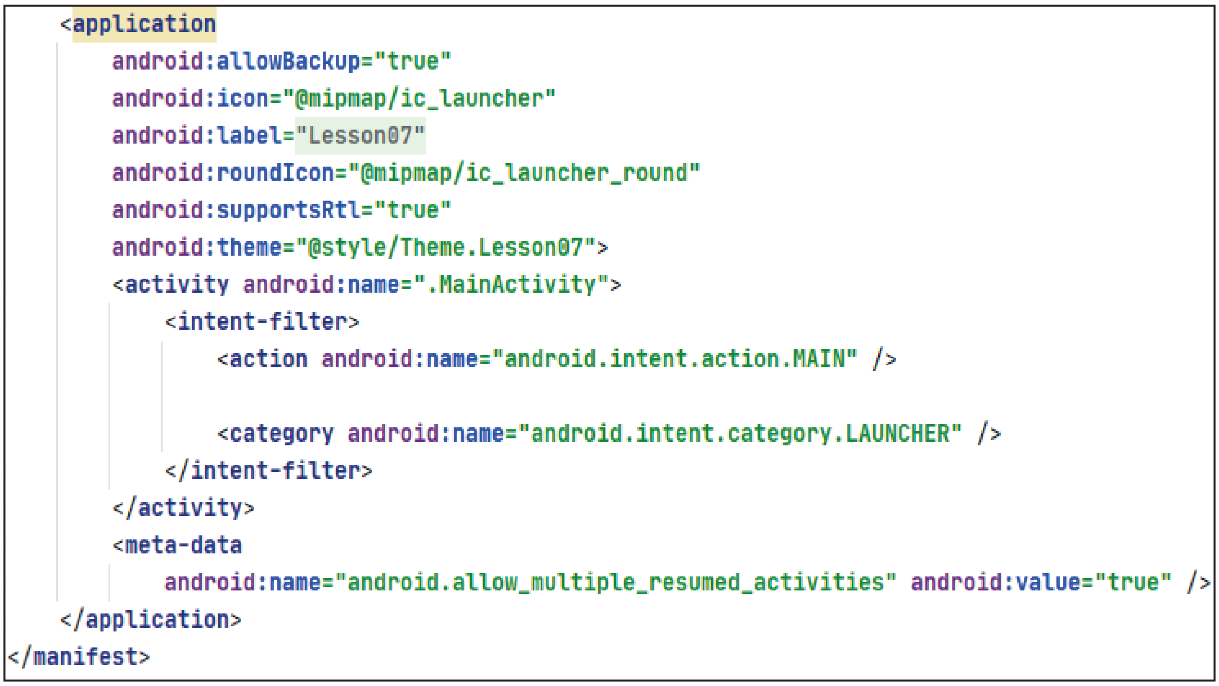
When running on Android 9.0 and earlier, only the app in focus is in the resumed state. All other visible Activities are paused. This can create problems if apps close resources or stop playing content when they pause.

In Android 10, this behavior changes so that all Activities remain in the resumed state when the device is in multi-window mode. This is called *multi-resume*.

Multi-resume is also available on select devices running Android 9.0. To opt-in for multi-resume on those devices, you can add the following manifest meta-data:

```
<meta-data

    android:name="android.allow_multiple_resumed_activities"
android:value="true" />
```

You should add the previous metadata to your app Manifest.xml file under the application tag as illustrated in the following figure:



```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Lesson07"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Lesson07">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <meta-data
            android:name="android.allow_multiple_resumed_activities" android:value="true" />
    </application>
</manifest>
```

These days, you see foldable phones that support more than one screen or display at a time. Handling this configuration is similar to how developers work with projected screens today on Chrome OS.

Android 10 (API level 29) and higher supports activities on secondary displays. If an activity is running on a device with multiple displays, users can move the activity from one display to another. Multi-resume applies to multi-screen scenarios as well. Several activities can receive user input at the same time.

You can specify which display (activity) it should run on when your app launches, or when it creates another activity. This behavior depends on the activity launch mode defined in the manifest file and in the intent flags and options set by the entity launching the activity. You will study in detail with some practice about how you select or configure your app startup (launch) activity in the next topic of this lesson.

You should not use `onPause()` to save application or user data, make network calls, or execute database transactions. Once `onPause()` finishes executing, the next callback is either `onStop()` or `onResume()`, depending on what happens after the activity enters the Paused state.

`onStop()` Method
The system calls `onStop()` method when the activity is no longer visible to the user. This may happen because the activity is being destroyed (Shutdown), a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity. In all of these cases, the stopped activity is no longer visible.

The next callback is either `onRestart()` or `onDestroy()`. If the activity is coming back to interact with the user, the system calls `onRestart()`. `onDestroy()` is called when this activity is completely terminated.

`onRestart()` Method
The system invokes the `onRestart()` callback when an activity in the Stopped state is about to restart. `onRestart()` restores the activity back to the state at the time when it was stopped.
This callback is always followed by `onStart()`.

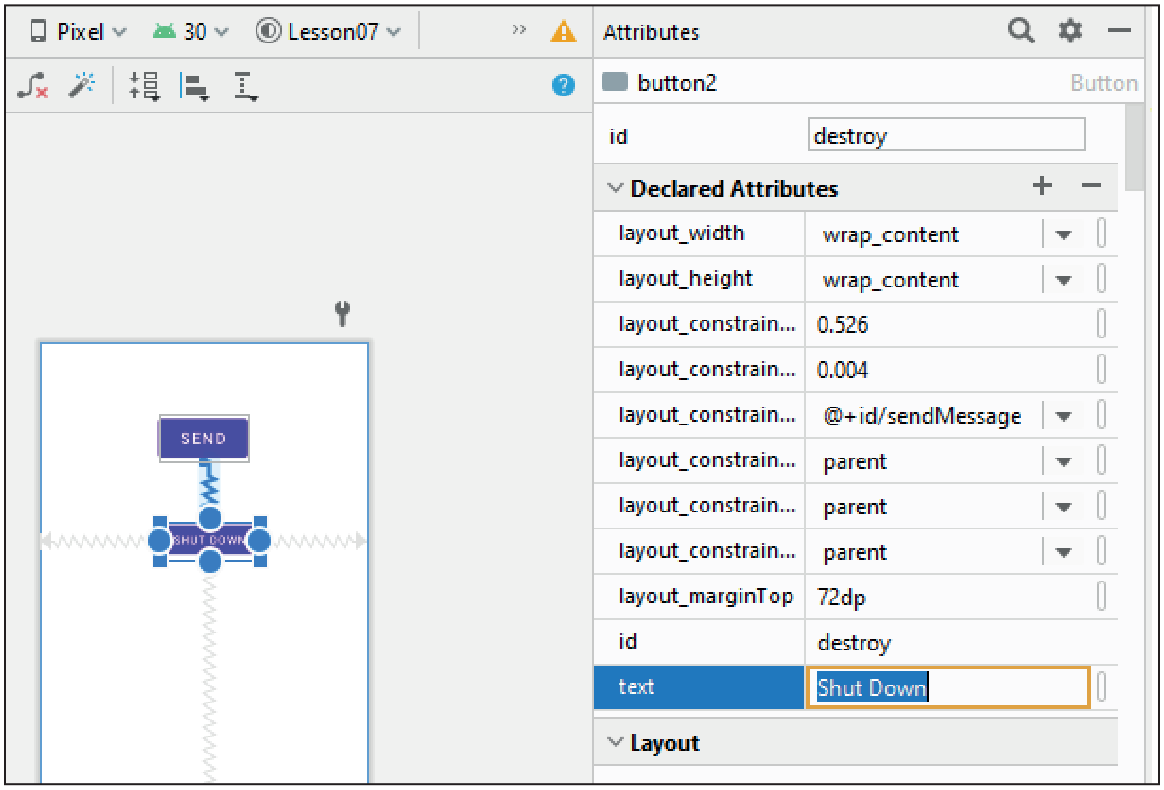
`onDestroy()` Method
The system invokes the **`onDestroy()`** callback before an activity is destroyed (Shutdown). This callback is the last callback that an activity receives. `onDestroy()` is usually implemented to ensure that all of an activity's resources are released when the activity or the process containing it, is destroyed.

Example: Use the same application and continue with the previous example. Add a button to your activity and configure your app to destroy (Shut Down) your app when the user clicks on this button as illustrated in the following steps:

1- Open the **activity_main.xml** in **Design** mode and then add a button to your activity with the following attributes' values:

id:	destroy
text:	Shut Down

2- Set the button constraints (margins) to be as illustrated in the following figure:



Attributes

button2 Button

id	destroy

Declared Attributes

layout_width	wrap_content
layout_height	wrap_content
layout_constrain...	0.526
layout_constrain...	0.004
layout_constrain...	@+id/sendMessage
layout_constrain...	parent
layout_constrain...	parent
layout_constrain...	parent
layout_marginTop	72dp
id	destroy
text	Shut Down

Layout

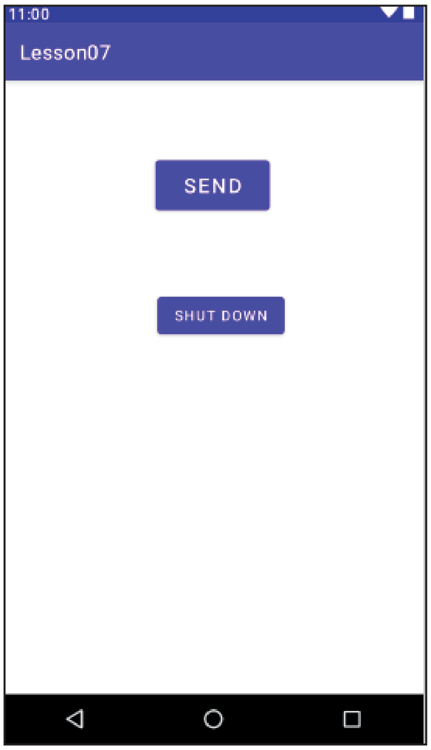
3- Open the **MainActivity** file, add the following **shutdown** function, and then add the `onDestroy()` method for this function as it illustrated in the gray highlighted color in the following code:

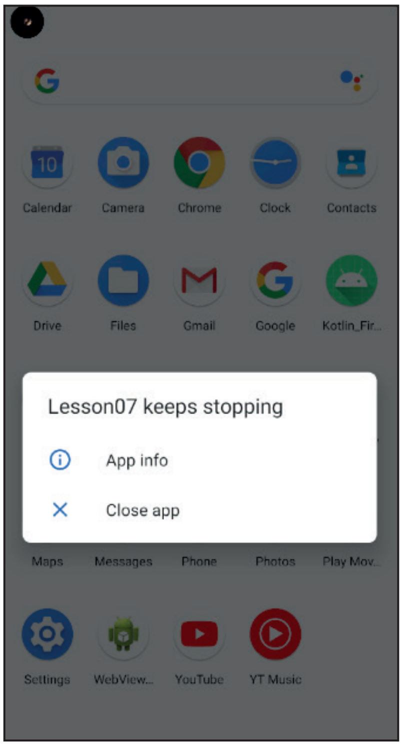
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

    }

    fun mgSnack(view: View) {
        Snackbar.make(
            findViewById(R.id.sendMessage),
            "Your Email has been sent successfully",Snackbar.
LENGTH_LONG)
             .setAction("OK"){}.setActionTextColor(Color.RED).show()
    }
    override fun onStart() {
        super.onStart()
        Snackbar.make(
            findViewById(R.id.sendMessage),
"onStart method is running now",Snackbar.LENGTH_LONG).show()

    }

    fun shutdown(view: View){
        onDestroy()
    }

}
```

4- Open the **activity_main.xml** in the **Code** mode, and configure the **shutdown** Button to run the shutdown function when the app user taps it. To do this, add the `android:onClick="shutdown"` attribute to your shutdown button tag as illustrated in the gray highlighted code in the following XML code of activity_main.xml file :

```
<Button
    android:id="@+id/destroy"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="72dp"
    android:text="Shut Down"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.526"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/sendMessage"
    app:layout_constraintVertical_bias="0.004"
    android:onClick="shutdown"/>
```

5- **Stop**, then **Run** your app. Tap the **Shut Down** button. The app will stop and close or you will get something similar to the following figure:





Lesson07 keeps stopping

App info

Close app

Android Intent

Android application components can connect to other Android applications. This connection is based on a task description represented by an **Intent** object. Android **Intent** is an abstract class description of an operation to be performed. **Intents** are asynchronous messages which allow application components to request functionality from other Android components. **Intents** allow you to interact with components from the same applications (such as other app activity) as well as with components contributed by other applications.

In Android, the reuse of other application components is a concept known as *task*. An app can access other Android components to achieve a *task*. For example, from a component of your app you can trigger another component in the Android system, which manages photos, even if this component is not part of your app. In this component, you select a photo and return to your app to use the photo you have selected.

Example about that is using WhatsApp or Instagram apps, where the app user can share an image from his/her device storage. This is what is called access another Android component and the app developers for these apps configure the **Intent** classes in theirs apps to give the app user the ability to connect and use another system component such as images, files or others. Also, this connection can be between the app activities themselves.

In Android, when you want to move from an activity to another you should use **Intent** class. **Intent** can be used to perform the following tasks:

1) Open another Activity or Service from the current Activity.
2) Pass data between Activities and Services.

3) Delegate responsibility to another application. For example, you can use Intents to open the web browser application to display a specific URL.

The primary pieces of information in intent are:

Action

The action largely determines how the rest of the Intent object is structured. The action in an Intent object can be set by the `setAction()` method and read by `getAction()` method.

The Intent class defines a number of action constants corresponding to different Intents. Here are some of Android Intent Standard Actions:

Activity Action Intent	Description
ACTION_BATTERY_LOW	This broadcast corresponds to the "Low battery warning" system dialog.
ACTION_CALL_BUTTON	The user presses the "call" button to go to the dialer or other appropriate UI for placing a call.
ACTION_CAMERA_BUTTON	The "Camera Button" is pressed.
ACTION_PASTE	Creates a new item in the given container, initializing it from the current contents of the clipboard.
ACTION_VOICE_COMMAND	Starts Voice Command.

Data

The data to operate on, such as a person's record (information/contact information) in the contacts database, expressed as a Uri.

In addition to these primary attributes, there are a number of secondary attributes that you can also include in intent:

Category

The category is an optional part of Intent object. It's a string containing additional information about the kind of component that should handle the intent. The `addCategory()` method places a category in an Intent object; `removeCategory()` deletes a category previously added, and `getCategories()` gets the set of all categories currently in the object.

Here are some of Android Intent Standard Categories:

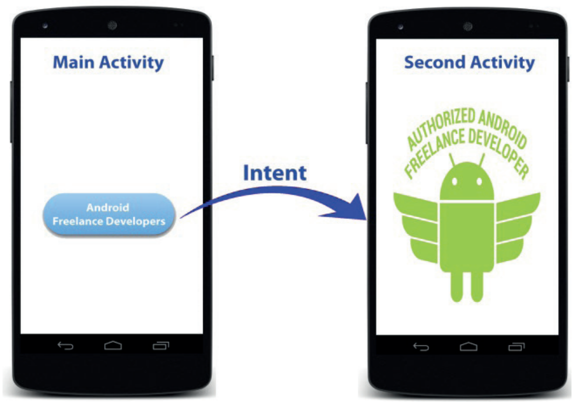
Categories	Description
CATEGORY_APP_CALCULATOR	Used with ACTION_MAIN to launch the calculator application.
CATEGORY_APP_CALENDAR	Used with ACTION_MAIN to launch the calendar application.
CATEGORY_APP_CONTACTS	Used with ACTION_MAIN to launch the contacts application.
CATEGORY_APP_EMAIL	Used with ACTION_MAIN to launch the email application.
CATEGORY_APP_GALLERY	Used with ACTION_MAIN to launch the gallery application.
CATEGORY_APP_MAPS	Used with ACTION_MAIN to launch the maps application.
CATEGORY_APP_MESSAGING	Used with ACTION_MAIN to launch the messaging application.
CATEGORY_HOME	This is the home activity. It is the first activity ~~that is~~ displayed when the device boots.

Extras

These are in key-value pairs for additional information that should be delivered to the component handling the intent. The extras can be set and read using the `putExtras()` and `getExtras()` methods respectively. The key in `putExtra` and `getExtra` should be the same; otherwise, you will not be able to receive the data from the other side (the other activity). In this lesson, you will use the `putExtra` and `getExtra` methods to exchange data between your app activities.

Navigating Between Activities

Normally, when you run your Android app, the main activity will appear as Main Activity. By default, the main activity's name is **MainActivity** and its name is configured within the AndroidManifest.xml file to startup first. Your app users should find in your main activity interface (startup interface) all the navigation tools such as buttons, images and others to give them the ability to move from one activity to another or to use the service which your app has provided. Usually, your app will have more than one activity.


Main Activity
Android Freelance Developers
Intent
Second Activity
AUTHORIZED ANDROID FREELANCE DEVELOPER

In this section of this lesson, you will learn how to create a new activity or more, how connect your app activities together or how the app user can move from one activity to another, and where do you define the starting activity of your app?. To do this, follow the following steps:

1- Open Android Studio, and then click **File** → **New** → **New Project**

2- Select **Empty Activity**, and click **Next**

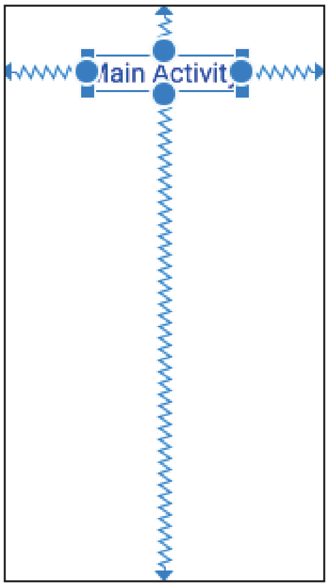
3- Type: **Lesson07_Intent** for the application name, then click **Finish**.

4- Open the **activity_main.xml** file in **Design** mode, select the **TextView** widget: "**Hello World!**", replace its **text** attribute value (Hello World!) with: **Main Activity**.

5- Change the following attributes' values for this TextView widget as follows:

textSize :34sp	**textColor**: 0000FF

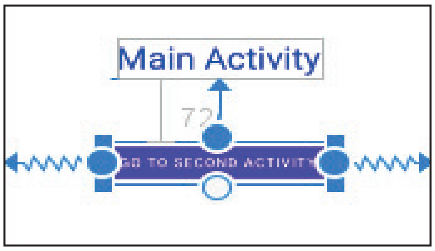
6- Move this **TextView** (Main Activity) to the top of your activity. It should be similar to the following figure:



7- Add a button to your main activity and set its constraints, then configure its attribute values as follows:

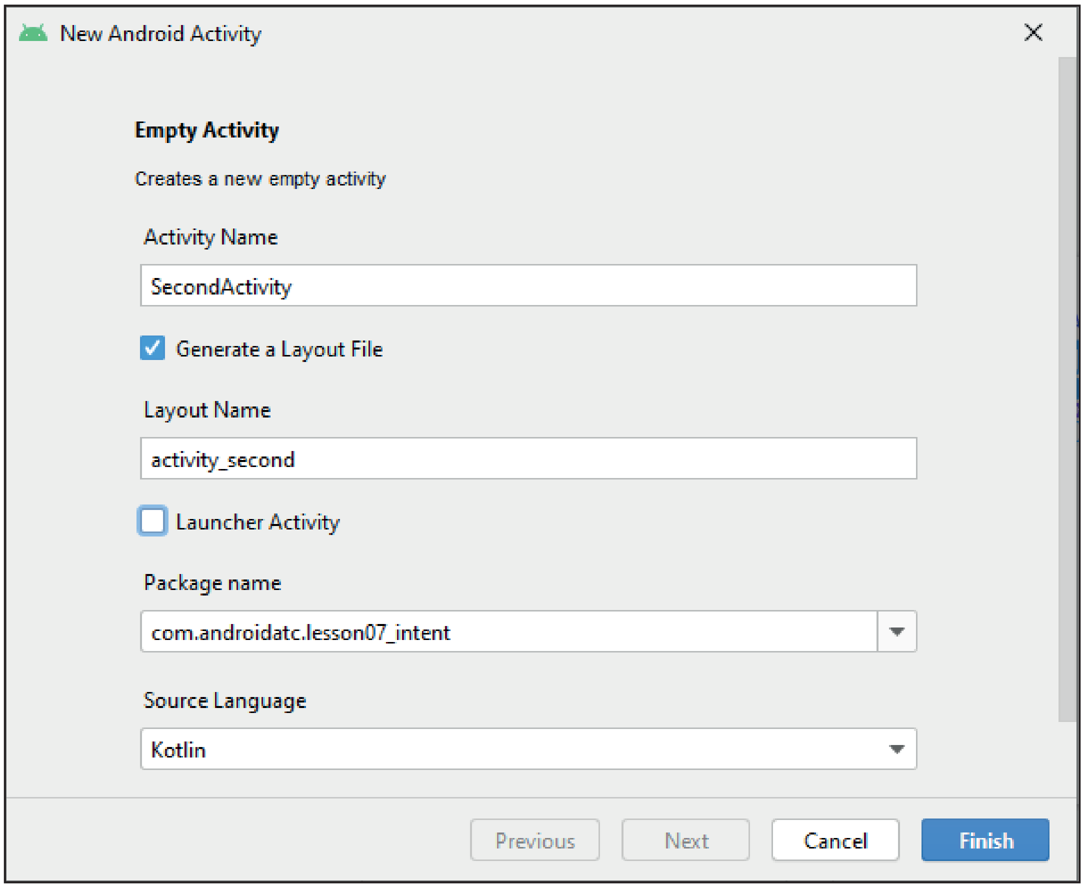
ID: go_to_SecondActivity	**text**: Go To Second Activity

This button will be used later to navigate to the second activity which will be created in the next step. The main activity will look like the following figure:



8- Create a new activity (second activity) by right clicking **com.androidatc.lesson07_intent** → **New** → **Activity** → **Empty Activity**.

9- Enter the new activity name as **SecondActivity**, and click **Finish** as illustrated in the figure below:



New Android Activity

Empty Activity

Creates a new empty activity

Activity Name: SecondActivity

☑ Generate a Layout File

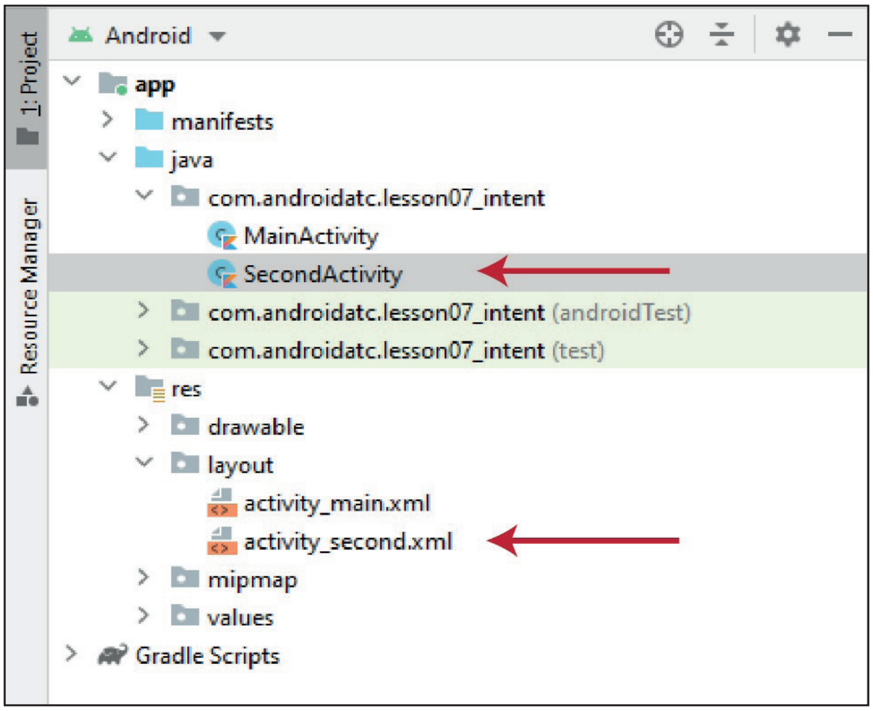
Layout Name: activity_second

☐ Launcher Activity

Package name: com.androidatc.lesson07_intent

Source Language: Kotlin

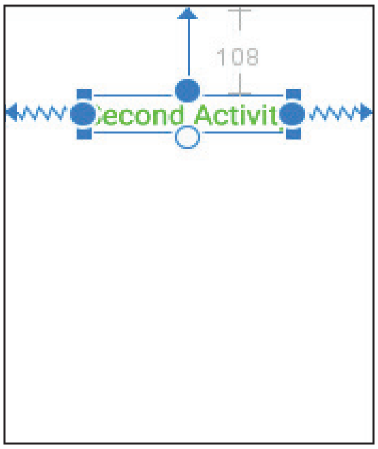
Two files will be added to your app, **SecondActivity.kt** and **activity_second.xml**, as illustrated in the following figure:



10- Open **activity_second.xml** in the **Design** mode, then add the **TextView** widget with the following attribute values:

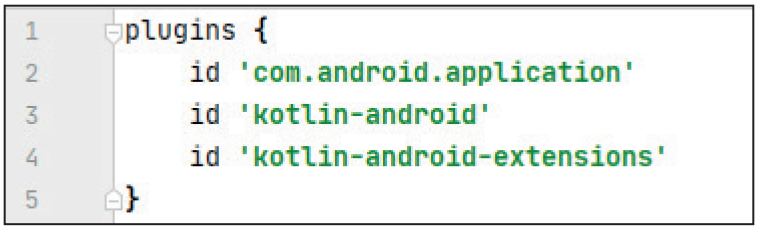
ID : second_activity_ID	**text**: Second Activity
textSize :34sp	**textColor**: 00FF00

Set its constraints, as well. The second activity should be like the following figure:



11- Before you start with typing the Kotlin code in your app, check the **build.gardle (Module:Lesson07_Intent.app)** file content and be sure it has the Kotlin plugin. If not, add the following code: **id 'kotlin-android-extensions'**

And click the **Sync Now**. The configuration should be as follows:



```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-android-extensions'
}
```

12- Open the **MainActivity** and write the following code which configures the "**Go To Second Activity**" button.

Create a "**go2secondActivity**" function which includes the Intent (link) to the second activity.

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

    fun go2secondActivity(view: View){}
}
```

13- Double click the **View** class which has a red color, click the pop-up red lamp, and then select **Import**.

14- Now, you can configure the **go2secondActivity** function to perform navigation from **MainActivity** to **SecondActivity**. Add the following code to this function:

```
var intent  =Intent(this,SecondActivity::class.java)
```

You should configure a variable that declares the link from the existing activity (MainActivity) to the destination activity (SecondActivity). Here in this example its name is ***intent***, you can use any name.

15- Double click the **Intent** class which has a red color, click the pop-up red lamp, and then select **Import**.

The following is an explanation for the last code:

`this`: means the existing activity (MainActivity)
`SecondActivity::class.java` : Means the destination activity is "**SecondActivity**" and any activity is a class (Java class).

The code is as follows:

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

    fun go2secondActivity(view: View){
        var intent  = Intent(this,SecondActivity::class.java)
    }
}
```

16- Now, add the **`startActivity()`** method which is responsible to launch or move to the **SecondActivity** activity using the intent information. The full code will be as follows:

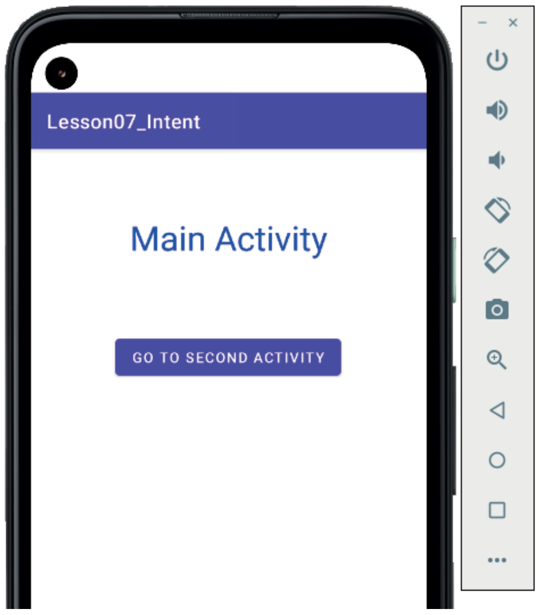
```
package com.androidatc.lesson07_intent

import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View

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

    fun go2secondActivity(view: View){
        var intent  = Intent(this,SecondActivity::class.java)
        startActivity(intent)
    }
}
```

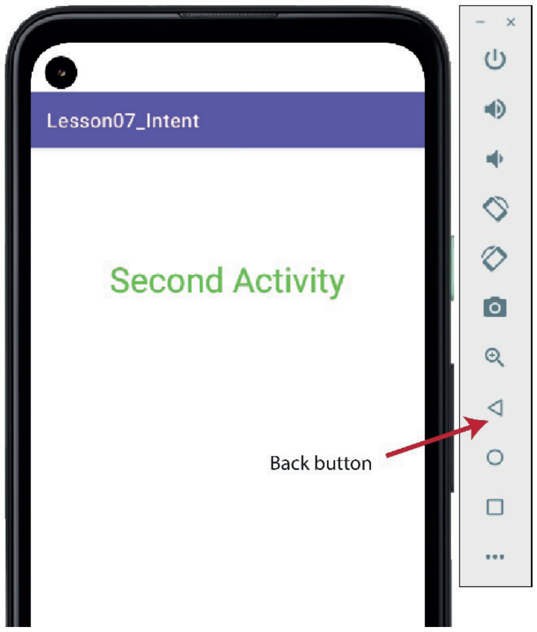
17- To link this **`go2secondActivity`** function operations with the: **Go To Second Activity** button, open the **activity_main.xml** file in the Code mode, and then add the **onClick** attribute to the Button tag as illustrated in the gray highlighted color in the following XML code:

```
<Button
    android:id="@+id/go_to_SecondActivity"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="72dp"
    android:text="Go To Second Activity"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.498"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    android:onClick="go2secondActivity"/>
```

18- Click the **Run** button to test the navigation button.

The following figures display the test result after clicking the "**GO TO SECOND ACTIVITY**" button. You can also click the back button to return to the main activity:





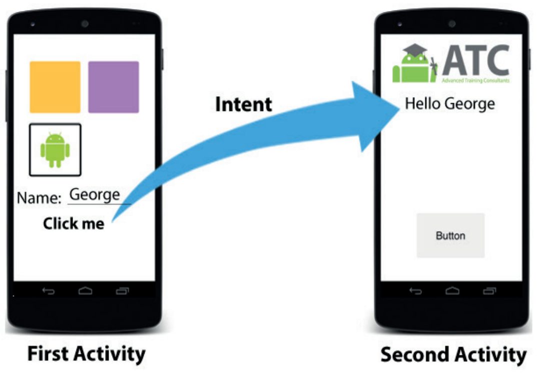
You can write the same **Intent** code in a different way and get the same operations result as illustrated in the code below in the **MainActivity** file:

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

    fun go2secondActivity(view: View){

        startActivity(Intent(this,SecondActivity::class.java))
    }

}
```

The second method is the shortest way to navigate from one activity to another, but the first way is also used to passing data from one activity to another as you will see in the following topic.

Passing Data between Activities

It is very likely that you may need to transfer some data to the activity you want to start through Intent. Android SDK provides this option using extra methods. You can attach data to your Intent using the Intent's method: **`putExtra()`** to include extra data in calling activity or the **`getExtra()`** method to retrieve data from the called activity; for example, sending one piece of data from the **First Activity** to the **Second Activity** when it starts.



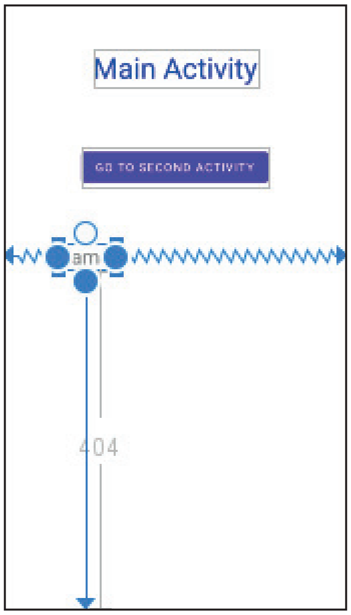
An Intent object is a bundle of information used by the component that receives the Intent as well as information used by the Android system.

Example:
Use the same Android project as used before which has two activities (MainActivity and Second Activity) to test passing data from the **MainActivity** to the **Second Activity** in your app. To do this, follow the steps below:

1- Open the **activity_main.xml** in the **Design** mode, add a **TextView** widget, set its constraints, and modify its attributes as follows:

text : **Name:**	textSize: 24sp

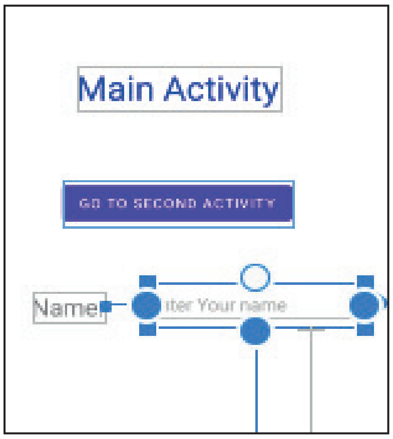
The interface should have the following figure:

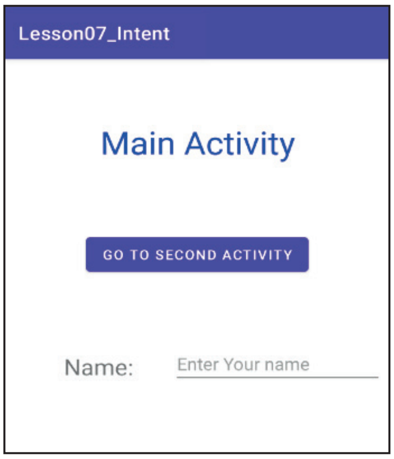


2- Add the **Plain Text** widget to the **activity_main.xml** file and set its constraints. Then configure its attributes values as follows:

id : name_ID	**textSize**: 24sp
text : delete its attribute value.	**hint:** Enter your name

Your activity should be like the following figure:



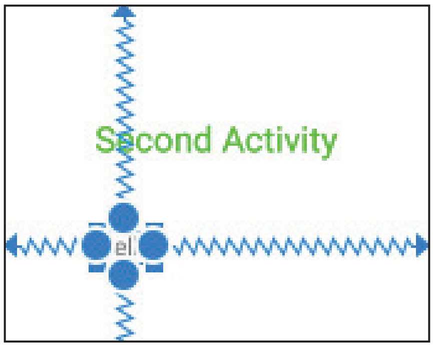


In this example, you will test using the Intent class to pass data which will be typed in the **Plain Text** widget and which has ID : **name_ID** to the **SecondActivity** in a **TextView** widget which will be added in the next step.

3- Open the **activity_second.xml** file in the **Design** mode, and add a **TextView** widget, set its constraints, and set its attributes values as follows:

text: Hello	**textSize**: 24sp

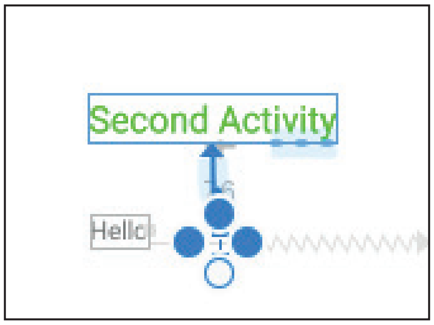
Until this step, the second activity should have the following figure:



4- Add a **TextView** beside the previous Hello text. This **TextView** message will display the app user name which will be typed in the main activity. Set its constrains, and change its attributes values as follows:

id: hello_message	**text**: Delete the value (keep it blank)	**textSize**: 24sp

The second activity should have the following figure:



5- Open the **MainActivity** file, and then add the: **`intent.putExtra`** method to the `go2secondActivity` function as illustrated in the following code:
`putExtra` method which is responsible for passing the data from app activity (interface) to another. This data includes the **Plain Text** widget whose **id** is: **name_id**

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

    fun go2secondActivity(view: View){
        var intent  = Intent(this,SecondActivity::class.java)
        intent.putExtra("name", name_id.text.toString())

        startActivity(intent)
    }
}
```

In the previous code, double click the: **name_id** which has a red color, click the red pop-up lamp, and click **Import**.

Note that "**name**" string in the first parameter is the name of the extra data (`putExtra` Key). This key in `putExtra` and `getExtra` methods must be the same; otherwise, you will not be able to receive the data from the other side (the other activity).

`name_id.`*`text`*`:` is the text which will be written in the field "putExtra" value.
`startActivity(intent)`: Launch a new activity which is configured before with intent.

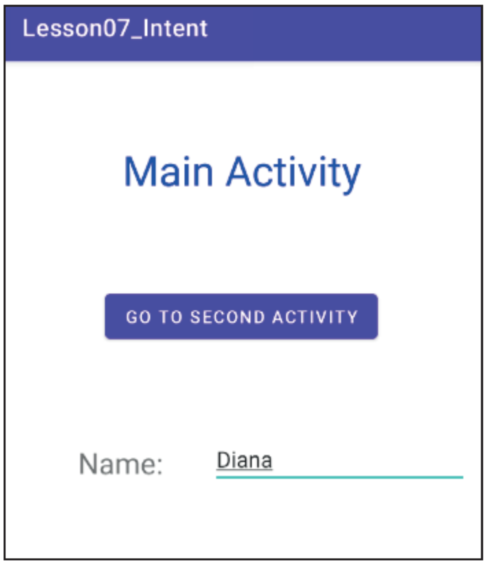
6- Now, you should configure the `getExtra` method in the second activity to retrieve the data which has been entered in the `putExtra` method. You want to display the data in the second activity in the **TextView** widget which has the id: hello_message

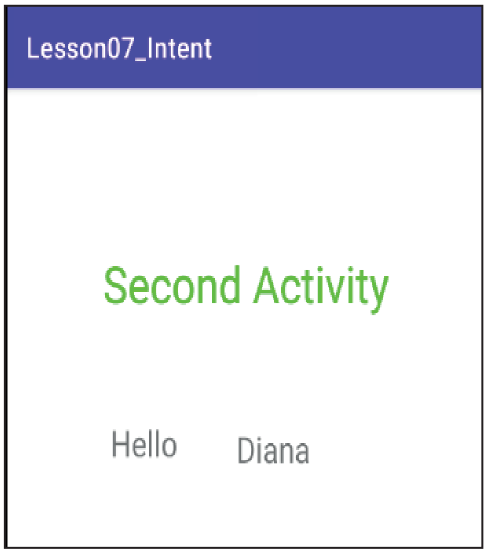
Open the **SecondActivity** file, then write the following code:

```
class SecondActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_second)
        var message = intent.getStringExtra("name")
        hello_message.text=message
    }
}
```

Double click the id: `hello_message` which has a red color, click the red pop-up lamp, and then select **Import**.

7- Run your app, type your name in the Main Activity, tape the : **GO TO SECOND ACTIVITY** button. The run result will be as follows:





Now, your app consists of two activities. If you want to configure the second activity to be the startup activity when you run your app, you should make some configurations in your app AndroidManifest.xml file. To do this, follow the following steps:

1- Open **AndroidManifest.xml** file, and note that you have two activity tags as illustrated in the gray highlighted part of the code below:

```
<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/Theme.Lesson07_Intent">
    <activity android:name=".SecondActivity"></activity>
    <activity android:name=".MainActivity">
        <intent-filter>
```

```
            <action android:name="android.intent.action.MAIN" />

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

2- Cut the **Intent-filter** tags content from the **MainActivity** activity tag and paste it inside the SecondActivity tag as illustrated in the following code:

```
<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/Theme.Lesson07_Intent">
    <activity android:name=".SecondActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

    </activity>
</application>
```

3- Stop and Run your app again. The startup activity will be the: **SecondActivity**

Android Alert Dialog

Dialogs constitute an important part of Android applications. A dialog is a small window that prompts the user to make a decision, enter additional information, or give a feedback. It does not take the full screen but part of it and asks users to take specific action before it can proceed.

Dialog Title
Alert Dialog Message
NO
YES

This dialog consists of a title, can display a number of buttons (one, two or three buttons), and/or a list of selectable items that may include checkboxes or radio buttons. The Alert Dialog is capable of constructing most dialog user interfaces; therefore, the dialog type is generally suggested. If you only want to display a String in this dialog box, use the `setMessage()` method.

To create an alert dialog box, you should create a new Android project and follow these steps:

1- Open Android Studio, and then click **File → New → New Project**

2- Select **Empty Activity**, and click **Next**

3- Type: **Lesson07_Dialog** for the application name, then click **Finish**.

4- Open the **activity_main.xml** file in Design mode, and then **delete** the "Hello World!" text.

5- Before starting with typing the Kotlin code in your app, check the **build.gardle (Module:Lesson07_Dialog.app)** file content and be sure it has the Kotlin plugin. If not, add the following code: **id 'kotlin-android-extensions'**

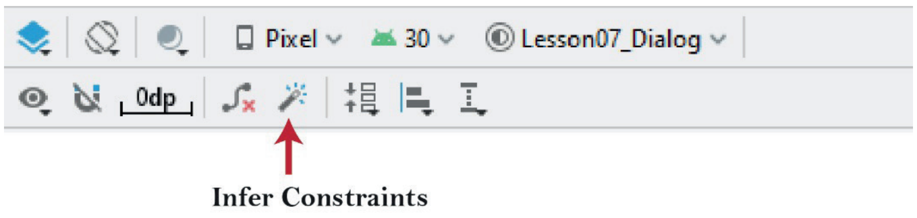
And click the **Sync Now**. The configuration should be as follows:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-android-extensions'
}
```

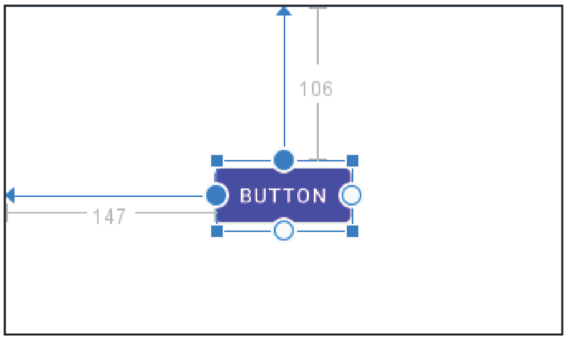
Your Scenario: Now, you will start creating an alert dialog box that asks your app users a question with two buttons (Yes and No). This alert dialog box will appear when the user performs a specific action. In this example, you will create a button (Save button) that when your app user taps, will make an an alert dialog box appear as illustrated in the next steps.

6- To create a "**Save**" button which will run that alert dialog box, open the **activity_main.xml** file in **Design** mode, then add a button widget using the drag and drop technique.

7- This time, you may use a different technique to set the button constraints (margins) or any other widgets. Select your button, click the "**Infer Constraints**" button as illustrated in the following figure.



You will find your button constraints have been set automatically, and you will get the following result or similar to it.



8- Change your button text attribute value to: **Save**

9- Open the **MainActivity** file to create a “**save**” function which you are going to link later with the **Save** button that you created previously. The code is as follows:

```
package com.androidatc.lesson07_dialog

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

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

    fun save(view: View){

    }

}
```

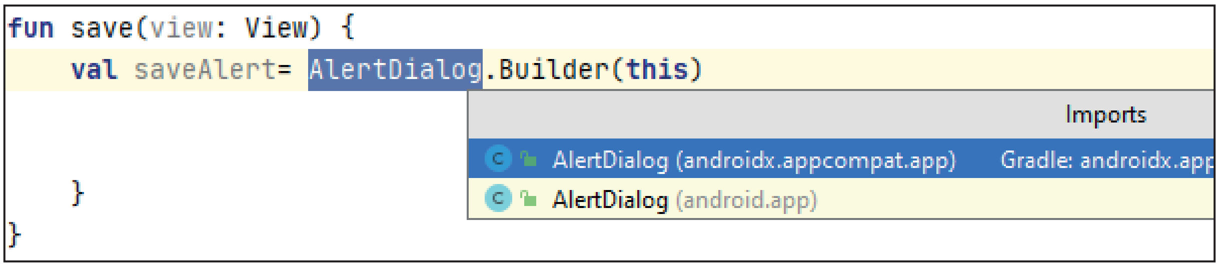
10- Double click the **View** class, click the red pop-up lamp, and select **Import**.

11- Now, you are going to use the **AlertDialog.Builder** class which creates a builder for an alert dialog and uses the default alert dialog theme. This class includes all the properties related to the alert dialog box content such as alert title, alert message and buttons.

Add a value represents this Alert dialog builder as illustrated in the code below.

```
fun save(view: View){
    val saveAlert=AlertDialog.Builder(this)

}
```

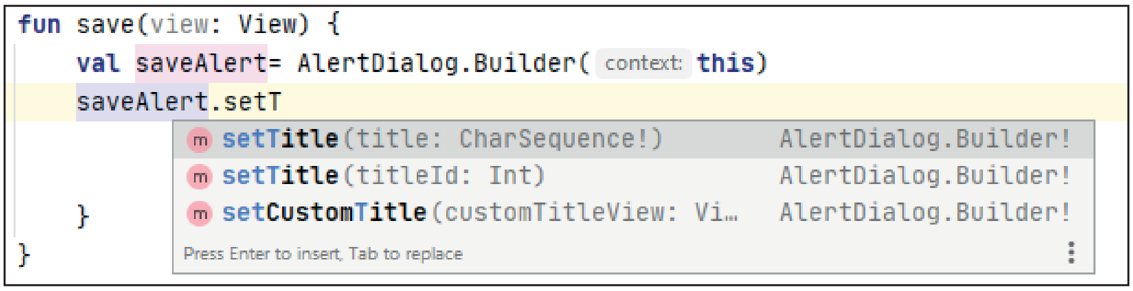
12- Double click the **`AlertDialog`**, click the red pop-up lamp, and select **Import** as illustrated in the figure below:



```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder(this)

    }
}
```

Imports
AlertDialog (androidx.appcompat.app) Gradle: androidx.app
AlertDialog (android.app)

The following figure displays how you can configure the **`AlertDialog.Builder`** class:



```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder( context: this)
    saveAlert.setT
    }
}
```

```
setTitle(title: CharSequence!)          AlertDialog.Builder!
setTitle(titleId: Int)                  AlertDialog.Builder!
setCustomTitle(customTitleView: Vi…     AlertDialog.Builder!
Press Enter to insert, Tab to replace
```

To configure the alert dialog box title, select: **`setTitle(title:CharSequence!)`**.

The code is as follows:

```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder(this)
    saveAlert.setTitle("Save")

    }
```

13- To set the message which will appear on the alert dialog box, add the **`saveAlert.setMessage`** to your save function as shown below:

```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder(this)
    saveAlert.setTitle("Save")
    saveAlert.setMessage("Are you sure you want to save your
changes?")
    }
```

14- The last step in configuring your alert dialog box is adding the buttons such as **Yes**, **No** or **Cancel**. Each one of these buttons can run a specific block of code. For example, if your app user selects the **Cancel** button, the alert dialog box will close without any extra action.

There are three different action buttons you can add by doing the following:

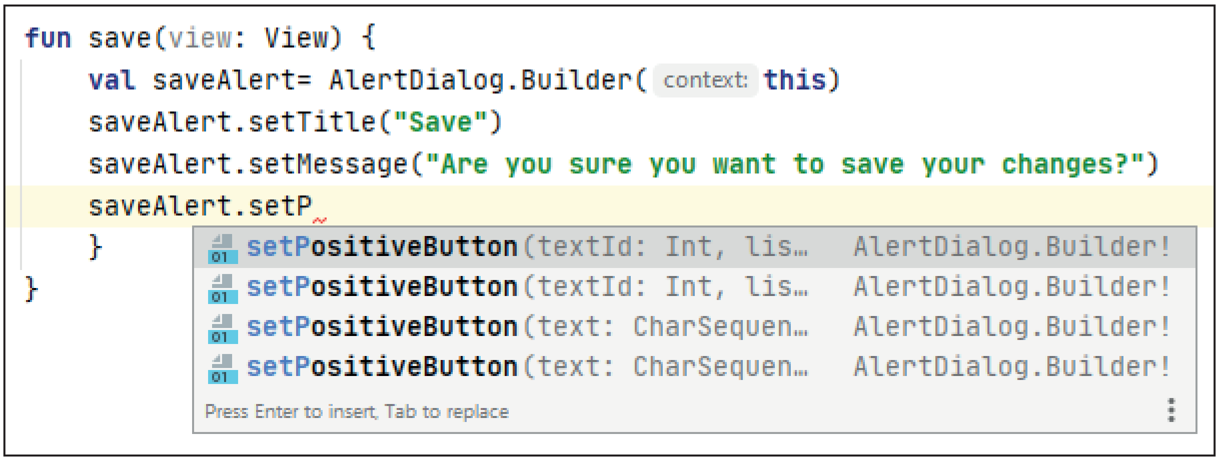
i -Positive: You should use this to accept and continue with the action (the "**OK**" action).

ii -Negative: You should use this to cancel the action.

iii -Neutral: You should use this when the user does 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."

Note: Only one button from each type can be added to an **`AlertDialog`** class. That is, you cannot have more than one "**positive**" button.

The following figure displays how you configure the positive button options which you select for your alert dialog box:



```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder( context: this)
    saveAlert.setTitle("Save")
    saveAlert.setMessage("Are you sure you want to save your changes?")
    saveAlert.setP
    }        setPositiveButton(textId: Int, lis…    AlertDialog.Builder!
}            setPositiveButton(textId: Int, lis…    AlertDialog.Builder!
             setPositiveButton(text: CharSequen…    AlertDialog.Builder!
             setPositiveButton(text: CharSequen…    AlertDialog.Builder!
             Press Enter to insert, Tab to replace
```

Add the following code:

```
saveAlert.setPositiveButton("Yes") { }
```

15- Between the previous two braces { }, you may add the block of code which will run if the app user clicks the "**Yes**" button. In this example, you are going to configure a **Snackbar** message, where a "Saved" message will appear when the user clicks the **Yes** button.

This code will be as follows:

```
saveAlert.setPositiveButton("Yes")
{ dialogInterface: DialogInterface, _: Int -> Snackbar.make(
findViewById(R.id.button),
    "Saved",Snackbar.LENGTH_LONG).show()}
```

16- Double click **`DialogInterface`** class, click the red pop-up lamp, and select **Import**. Repeat the same thing to import the Snackbar class.

17- Now, you should add the **`saveAlert.show()`** to your code to display the alert dialog box.

The full code of the **save** function is as follows:

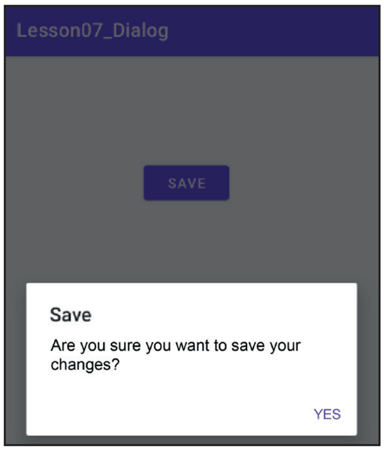
```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder(this)
    saveAlert.setTitle("Save")
    saveAlert.setMessage("Are you sure you want to save your
changes?")
    saveAlert.setPositiveButton("Yes")
    { dialogInterface: DialogInterface, _: Int -> Snackbar.make(
findViewById(R.id.button),
        "Saved",Snackbar.LENGTH_LONG).show()}
    saveAlert.show()

}
```

18- Now, let us test this alert dialog code until this step, and to do that assign this save function to operate when the app user taps the **SAVE** button.

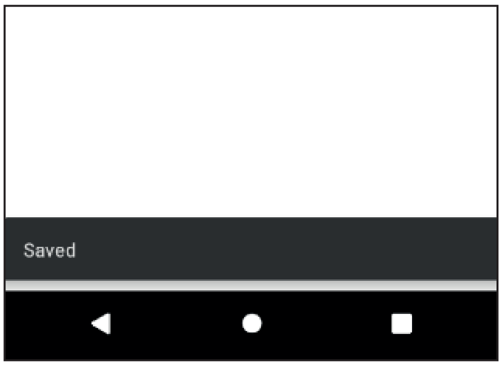
Open the **activity_main.xml** file in the **Code** mode, and add the **onClick** attribute to the **Button** tag as illustrated in the gray highlighted code of the XML code below:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="147dp"
    android:layout_marginTop="106dp"
    android:text="Save"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:onClick="save"/>
```

19- Run your app, and tap the **SAVE** button to test your alert dialog code. You should get the following screenshots when you tape the **SAVE** button and when you select **YES**.



Are you sure you want to save your changes?


Saved

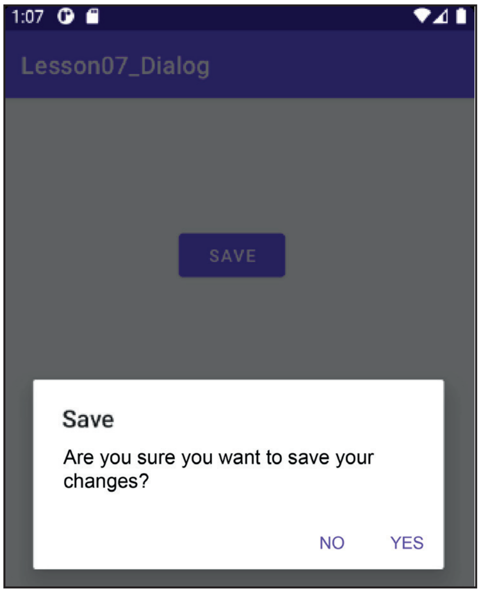
20- Now, you should add the **Negative** action to your alert dialog by adding the following **`setNegativeButton`** method:

```
saveAlert.setNegativeButton("No"){}
```

Here, between these two braces { } you can add the code which will run when your app user clicks the: **No** button. We will just add a Snackbar message in this example to test the operation of the negative action as illustrated in the code below:

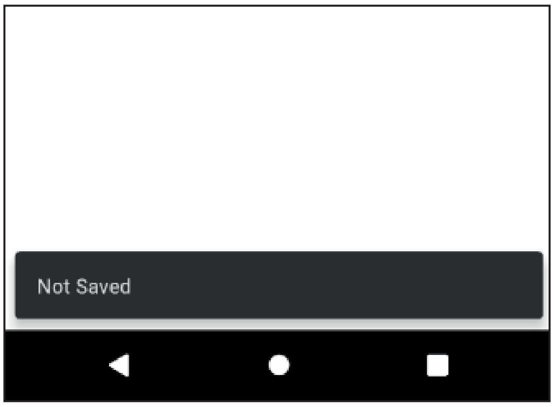
```
fun save(view: View) {
    val saveAlert= AlertDialog.Builder(this)
    saveAlert.setTitle("Save")
    saveAlert.setMessage("Are you sure you want to save your 
changes?")
    saveAlert.setPositiveButton("Yes")
    { dialogInterface: DialogInterface, _: Int -> Snackbar.make(
findViewById(R.id.button),
        "Saved",Snackbar.LENGTH_LONG).show()}

    saveAlert.setNegativeButton("No")
    { dialogInterface: DialogInterface, _: Int -> Snackbar.make(
findViewById(R.id.button),
        "Not Saved",Snackbar.LENGTH_LONG).show()}

    saveAlert.show()

}
```

21- **Stop** and then **Run** your app. Test your negative alert dialog operations. The following figures display the alert dialog and the Snackbar message which you will get when you tap **NO** choice:



Save

Are you sure you want to save your changes?



Not Saved

22- You can add a third button of a **Neutral** action to your alert dialog such as one with a **Remind me later** message if you add the **`saveAlert.setNeutralButton`** method as illustrated in the following code:

```
saveAlert.setNeutralButton("Remind me later"){ }
```

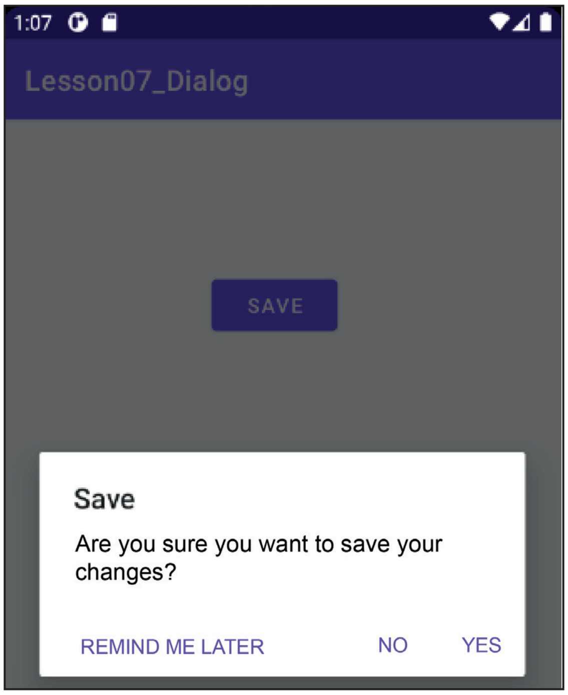
Because no action is expected by selecting this choice, you may use this method as the code below.Your app user will just close the alert dialog box if he/she taps this: **Remind me later** choice:

```
saveAlert.setNeutralButton("Remind me later", null)
```

The full code is as follows:

```
package com.androidatc.lesson07_dialog

import android.app.AlertDialog
import android.content.DialogInterface
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import com.google.android.material.snackbar.Snackbar

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

    fun save(view: View) {
        val saveAlert= AlertDialog.Builder(this)
        saveAlert.setTitle("Save")
        saveAlert.setMessage("Are you sure you want to save your 
changes?")
        saveAlert.setPositiveButton("Yes")
        { dialogInterface: DialogInterface, _: Int -> Snackbar.
make( findViewById(R.id.button),
            "Saved",Snackbar.LENGTH_LONG).show()}

        saveAlert.setNegativeButton("No")
        { dialogInterface: DialogInterface, _: Int -> Snackbar.
make( findViewById(R.id.button),
            "Not Saved",Snackbar.LENGTH_LONG).show()}

        saveAlert.setNeutralButton("Remind me later", null)

        saveAlert.show()

    }
}
```

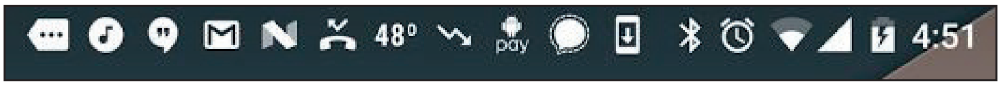
23- The following figure displays the alert dialog which you should get when you tap the **Save** button. When you tap the: **REMIND ME LATER** choice, the alert dialog will be closed without any extra action.



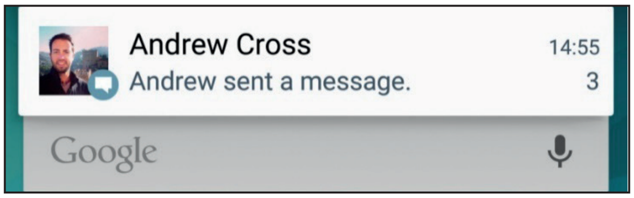
Android Notifications

A notification is a message you display to the user outside of your app's user interface to provide the user with reminders, communication from other people or other timely information from your app. Users can tap the notification to open your app or take action directly from the notification. When your app sends a notification to the Android operation system, the notification manager service (Android service) receives your app notification and issues it. The notification appears first as an icon in the **notification area**. To see the details of the notification, the user should open the **notification drawer**. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.

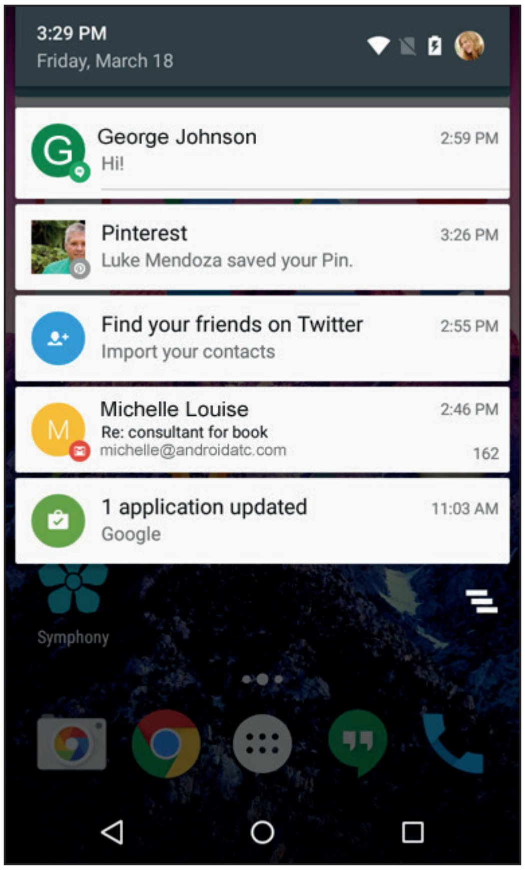
The following figures display the notifications in the ***notification area***:



The following is also a ***notification area***:



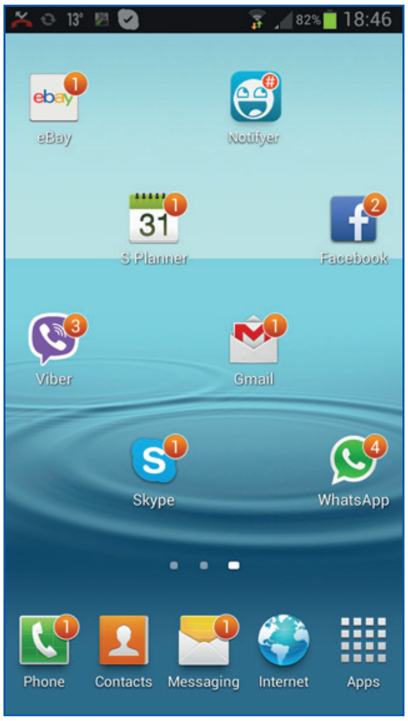
The figure below displays the ***notification drawer***. Users can swipe down on the status bar to open the notification drawer, where they can view more details and take actions with the notification.



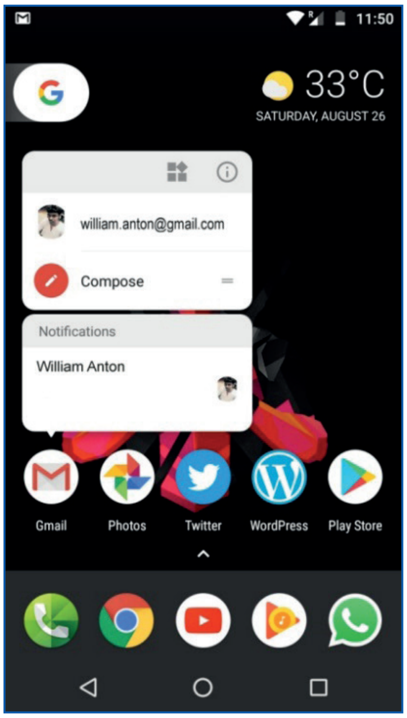
A notification remains visible in the notification drawer until dismissed by the app or the user. Beginning with Android 5.0, notifications can briefly appear in a floating window called a *heads-up notification*. This behavior is normally for important notifications that the user should know about immediately.

In supported launchers on devices running Android 8.0 (API level 26) and higher, app icons indicate new notifications with a colored “badge” on the corresponding app launcher icon.

Users can long-press on an app icon to see the notifications for that app. Users can then dismiss or act on notifications from that menu, similar to the notification drawer.



Notification badges on app icons.



Notifications associated with a notification badge.

Creating an Android Notification

Android notification contains the following objects:

1- A small icon, set with `setSmallIcon()`.

2- A Title: This is optional and set with `setContentTitle()`.

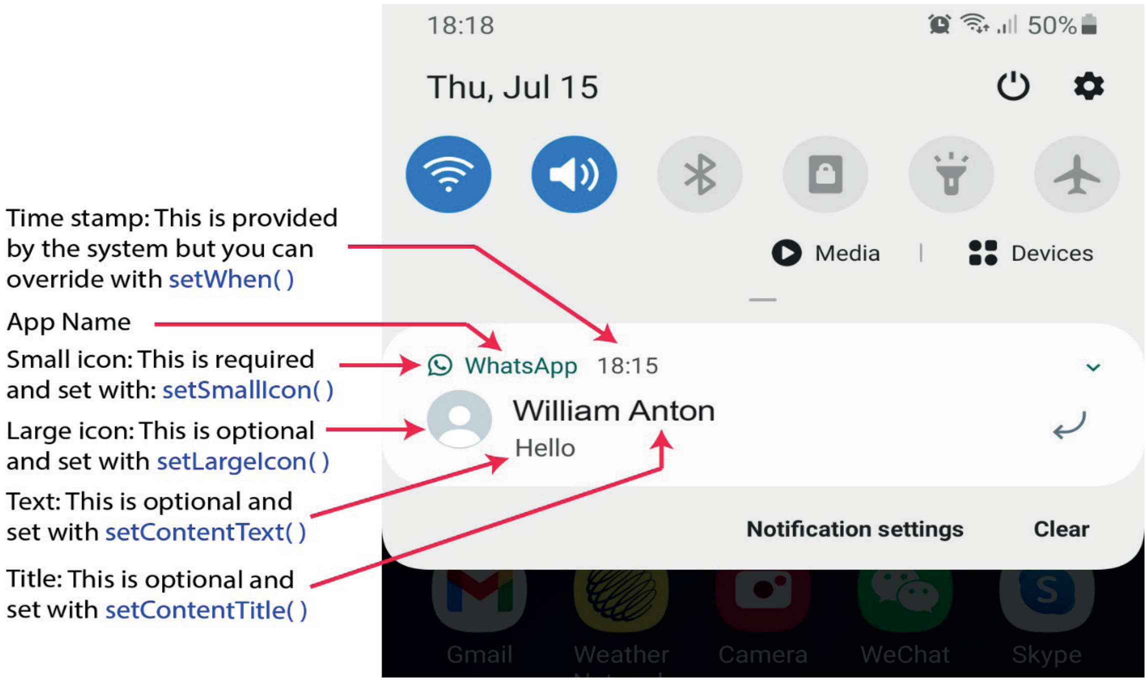
3- Text (text details), set with `setContentText()`.

4- Large icon: This is option (usually used only for contact photos; do not use it for your app icon) and set with `setLargeIcon()`.

5- Time stamp: This is provided by the system but you can override with `setWhen()` or hide it with `setShowWhen(false)`.

6- App name: This is provided by the system.

The following figure displays the notification objects:


Time stamp: This is provided by the system but you can override with setWhen( )
App Name
Small icon: This is required and set with: setSmallIcon( )
Large icon: This is optional and set with setLargeIcon( )
Text: This is optional and set with setContentText( )
Title: This is optional and set with setContentTitle( )
18:18
50%
Thu, Jul 15
Media
Devices
WhatsApp 18:15
William Anton
Hello
Notification settings
Clear
Gmail
Weather
Camera
WeChat
Skype

Starting in Android 7.0 (API level 24), you can also add an action to reply to messages or enter other text directly from the notification.

Starting in Android 10 (API level 29), the platform can automatically generate action buttons with suggested intent-based actions.

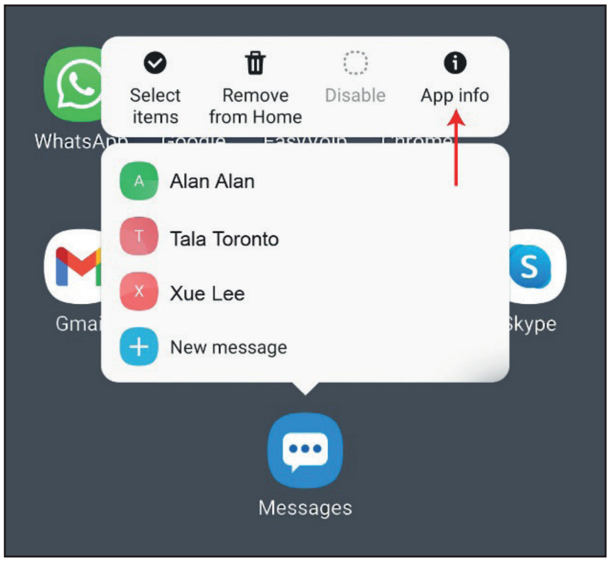
You will know more about adding action buttons to your app notification in the creating notification topic in this lesson.

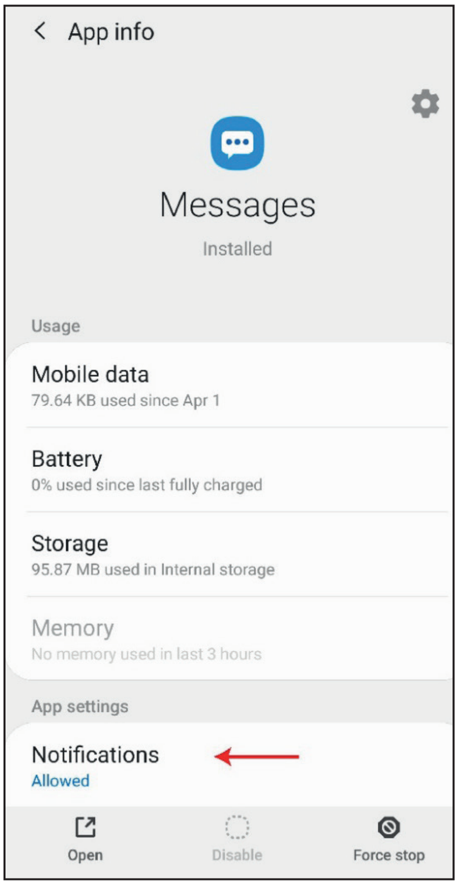
Notification Channel

Why is a notification channel important for Android apps? What are the control features to be applied with the notification channel?

Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel or it will not appear. Therefore, you must implement one or more notification channels to display your app notifications to users.

By categorizing notifications into channels, users can disable specific notification channels for your app (instead of disabling *all* your notifications), and users can control the visual and auditory options for each channel—all from the Android system settings as illustrated in the following figure for the Messages app which is installed by default on each Android phone. If you long-press the app icon, then press the **App info**, you will get the app settings as illustrated in the following figure:




App info

Messages
Installed

Usage

Mobile data
79.64 KB used since Apr 1

Battery
0% used since last fully charged

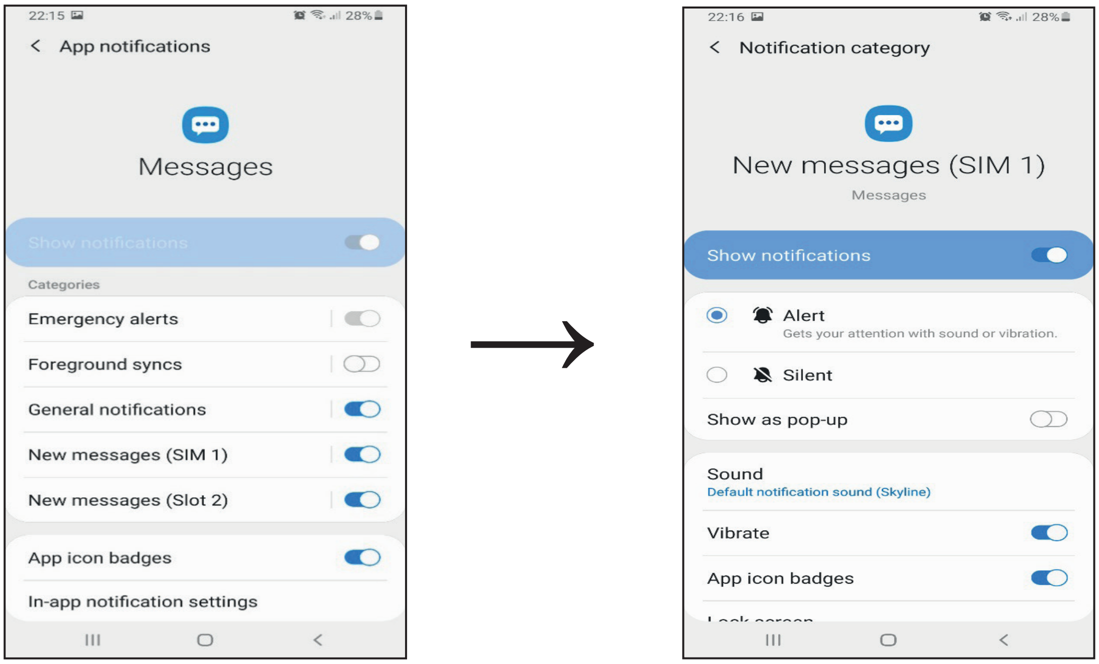
Storage
95.87 MB used in Internal storage

Memory
No memory used in last 3 hours

App settings

Notifications
Allowed

If you tap the **Notifications**, you may change behaviors for the associated channels as illustrated in the following figures:


22:15 28%

App notifications

Messages

Show notifications

Categories

Emergency alerts

Foreground syncs

General notifications

New messages (SIM 1)

New messages (Slot 2)

App icon badges

In-app notification settings

22:16 28%

Notification category

New messages (SIM 1)
Messages

Show notifications

Alert
Gets your attention with sound or vibration.

Silent

Show as pop-up

Sound
Default notification sound (Skyline)

Vibrate

App icon badges

You can create an instance of *Notification Channel* for each distinct type of notification you need to send. You can also create notification channels to reflect choices made by users of your app. For example, you may set up separate notification channels for each conversation group created by a user in a messaging app.
Users can manage most of the settings associated with notifications using a consistent system UI.

All notifications posted to the same notification channel have the same behavior. When a user modifies the behavior of any of the characteristics blow, the changes will apply to the notification channel as well:

- Importance
- Sound
- Lights
- Vibration
- Show on lock screen
- Override do not disturb

Each notification channel must have a valid notification ***channel ID*** which is set with **`setChannelId()`** or is provided in the **`NotificationCompat.Builder`** constructor when creating a channel. You should configure a *notification channel* code before you configure a notification message, because the configuration of the notification message needs to use the notification *channel id* which you have configured first.

Example:
The following example explains how you can configure Android notification in your app step by step:

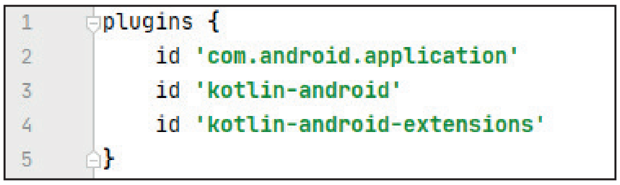
1- Open Android Studio, and then click **File** → **New** → **New Project**

2- Select **Empty Activity**, and click **Next**

3- Type: **Lesson07_Notification** for the application name, then click **Finish**.

4- Before you start with typing the Kotlin code in your app, check the **build.gardle (Module:Lesson07_Notification.app)** file content and be sure it has the Kotlin plugin. If not, add the following code: **id 'kotlin-android-extensions'**

And click the **Sync Now**. The configuration should be as follows:



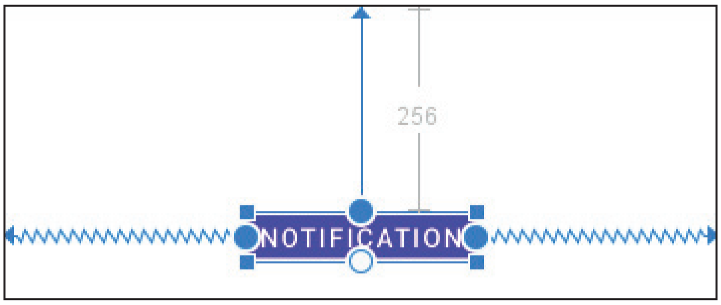
```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-android-extensions'
}
```

5- Open the **activity_main.xml** in Design mode and **Delete** the "Hello World!" **TextView**.

6- Now, you are going to create a button called **notification** button. When the user clicks the button, the notification generation process will start. To do so, open the **activity_main.xml** file in "**Design**" mode, then add a button to your activity using the drag and drop technique. Set the button widget's constraints, and then configure its attribute values as follows:

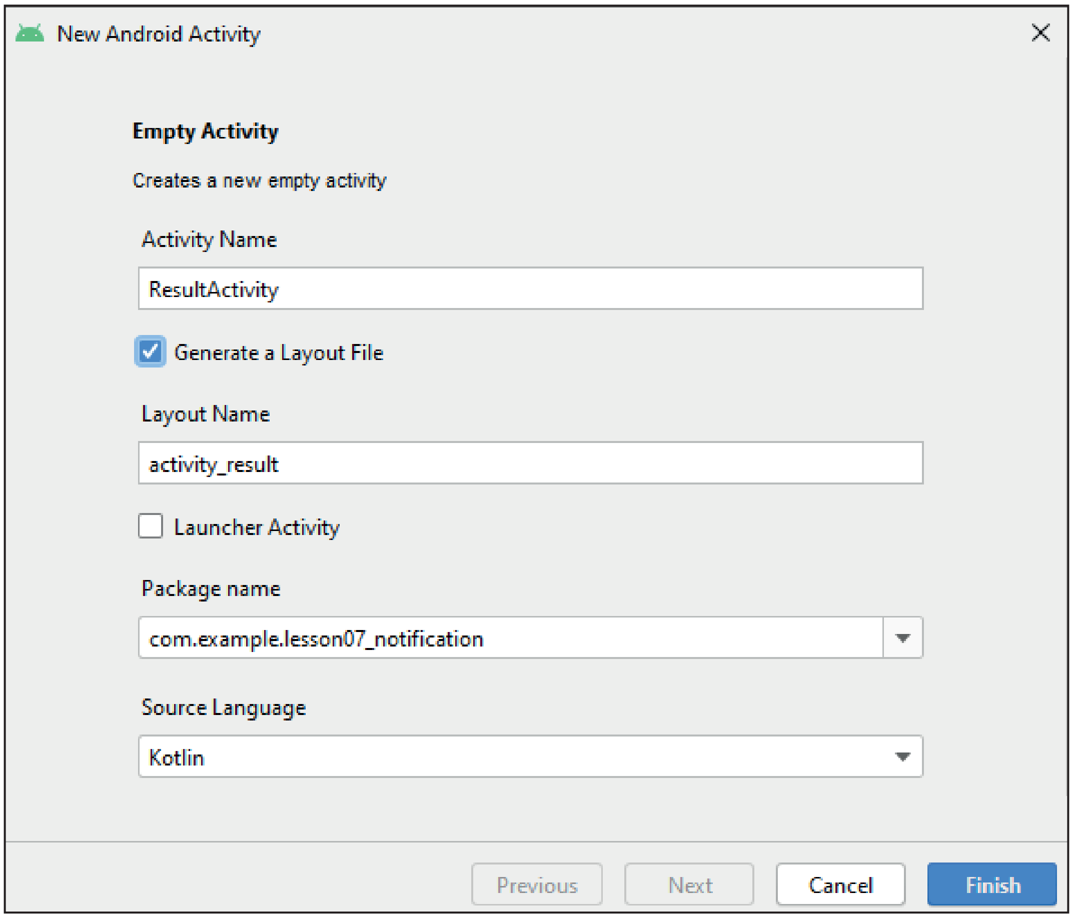
ID : notificationBtn	text : Notification	textSize: 34sp

The following figure displays the main activity after the button is added:



7- Our scenario for this example is, when the app user taps the notification message which appears on the notification area or on the status bar, another activity will be opened.

Now, you are going to create this other activity to be ready for use when you configure your notification message generation. To do this, right click "**com.androidatc.lesson_07_notification**" → **New** → **Activity** → **Empty Activity**. Configure the **Activity Name: ResultActivity** as illustrated in the following figure, then click: **Finish**


New Android Activity

Empty Activity

Creates a new empty activity

Activity Name: ResultActivity

[✓] Generate a Layout File

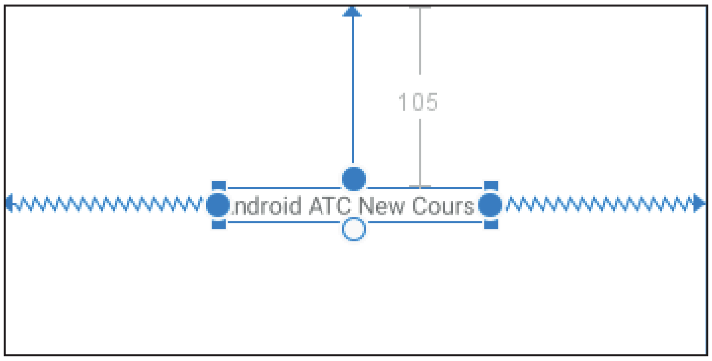
Layout Name: activity_result

[ ] Launcher Activity

Package name: com.example.lesson07_notification

Source Language: Kotlin

8- Open the **activity_result.xml** file in the **Design** mode, drag and drop a **TextView** widget, set its constraints, set its **text** attribute value to: "**Android ATC New Course**" as illustrated in the following figure:



9- Open the **MainActivity** file, Create a function called `courseupdate( )`. This function is responsible to generate and display the notification message. You will link the operation of this function later with the "**NOTIFICATION**" button in your first activity. The code is as follows:

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

    fun courseupdate(view: View){ }
}
```

10- Double Click the **View** class, click the red pop-up lamp, and select **Import**.

11- Creating and managing notification channels in Android 8 (Android Oreo) or Android API level 26 and higher is different. So, we will add an IF condition to our code to create a notification if the Android SDK API level 26 (Android O or Oreo) or higher as illustrated in the code below:

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

    }

    fun courseupdate(view: View) {

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

        }
    }}
```

12- Double Click **`Build`**, click the red pop-up lamp, and select **Import**.

Note: The O letter in this code: **`Build.VERSION_CODES.O`** represents Android Oreo or Android SDK version 26.

13- As we mentioned in the notification channel topic before, you should configure a notification channel configuration before you configure a notification message, because the configuration of the notification message needs to use the notification ***channel id*** which you have created first.

Add the following code to create your notification channel:

```
fun courseupdate(view: View) {

 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

   // Create the Notification Channel
   val channel_id = "channel_01"
   val channel_Name="notification"
   val importance = NotificationManager.IMPORTANCE_DEFAULT
   val mChannel=NotificationChannel(channel_id, channel_Name,
importance)
    mChannel.description = "test description"
    mChannel.enableLights(true)
    mChannel.lightColor = Color.RED
    mChannel.enableVibration(true)

    }
}
```

14- Double Click the **NotificationManager** class, click the red pop-up lamp, and select **Import**.

15- Double Click the **NotificationChannel** class, click the red pop-up lamp, and select **Import**.

16- Double Click the **Color** class, click the red pop-up lamp, and select **Import**.

17- Now, to create your notification message use the **Notification.Builder** class as illustrated in the following code:

```
fun courseupdate(view: View) {

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

// Create the Notification Channel

 val channel_id = "channel_01"
 val channel_Name="notification"
 val importance = NotificationManager.IMPORTANCE_DEFAULT
 val mChannel = NotificationChannel(channel_id, channel_Name,
importance)
   mChannel.description = "test description"
   mChannel.enableLights(true)
   mChannel.lightColor = Color.RED
   mChannel.enableVibration(true)
// Use Notification.Builder to generate your notification message

 val notification: Notification = Notification.Builder(this, channel_id)
       .setSmallIcon(R.drawable.ic_launcher_background)
       .setContentTitle("Android ATC Notification")
       .setContentText("Check Android ATC New Course !!")
       .build()

     }
}
```

18- Double Click the **Notification** class, click the red pop-up lamp, and select **Import**.

19- Register the channel with the system using **getSystemService** method which is used to access your Android system-level services as illustrated in the following figure:

```
fun courseupdate(view: View) {

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

// Create the Notification Channel
 val channel_id = "channel_01"
 val channel_Name="notification"
 val importance = NotificationManager.IMPORTANCE_DEFAULT
 val mChannel = NotificationChannel(channel_id, channel_Name,
importance)
   mChannel.description = "test description"
   mChannel.enableLights(true)
   mChannel.lightColor = Color.RED
   mChannel.enableVibration(true)
// Use Notification.Builder to generate your notification message
 val notification: Notification = Notification.Builder(this, channel_id)
      .setSmallIcon(R.drawable.ic_launcher_background)
      .setContentTitle("Android ATC Notification")
      .setContentText("Check Android ATC New Course !!")
      .build()
// Register the channel with your Android system
val mNotificationManager = getSystemService(NOTIFICATION_SERVICE) as
NotificationManager

     }
}
```

20- Now, add the last code to display the notification message in the notification channel as illustrated in the full code:

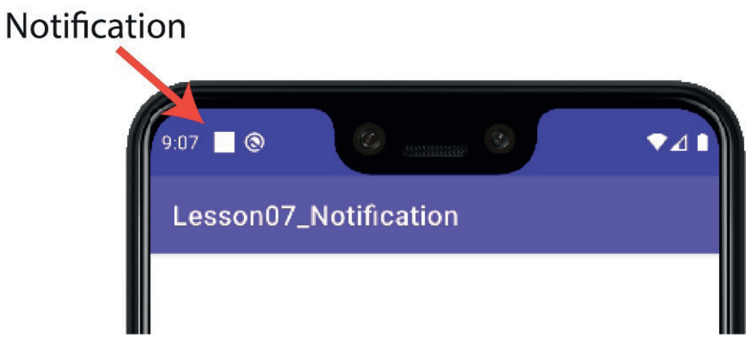
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

    }

fun courseupdate(view: View) {

 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

// Create the Notification Channel
 val channel_id = "channel_01"
 val channel_Name="notification"
 val importance = NotificationManager.IMPORTANCE_DEFAULT
 val mChannel = NotificationChannel(channel_id, channel_Name,
importance)
     mChannel.description = "test description"
     mChannel.enableLights(true)
     mChannel.lightColor = Color.RED
      mChannel.enableVibration(true)

// Use Notification.Builder to add the notification objects
  val notification: Notification = Notification.Builder(this,
channel_id)
      .setSmallIcon(R.drawable.ic_launcher_background)
      .setContentTitle("Android ATC Notification")
      .setContentText("Check Android ATC New Course !!")
      .build()

// Register or add the channel with your Android system
  val mNotificationManager = getSystemService(NOTIFICATION_SERVICE)
as NotificationManager

  if (mNotificationManager != null) {
      mNotificationManager.createNotificationChannel(mChannel)
//Show the notification
      mNotificationManager.notify(1, notification)
            }


        }
    }
}
```

Note: The number 1 in the last line of the previous code represents the notification ID. This id must be an integer number.

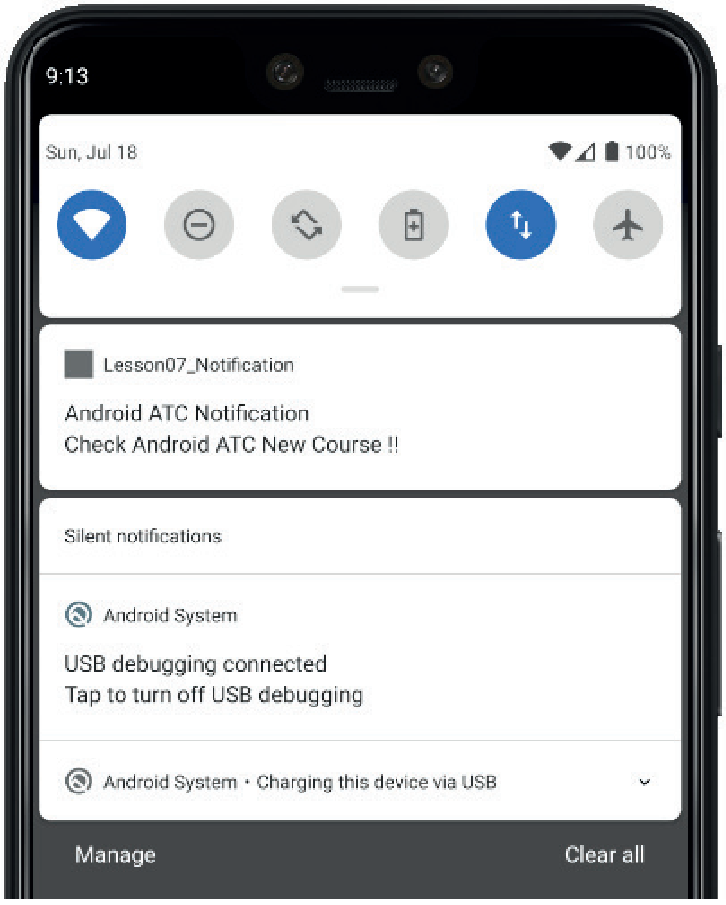
21- Open the **activity_main.xml** file in **Code** mode, then add the **android:onClick="courseupdate"** attribute to the **Button** tag as illustrated in the gray highlighted part of the code below. This button is responsible for starting the **courseupdate** function work (generation the notification message) when the user taps the button.

```
<Button
    android:id="@+id/notificationBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="328dp"
    android:layout_marginTop="256dp"
    android:text="Notification"
    android:textSize="34sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:onClick="courseupdate"/>
```

22- **Run** your app and click the "**NOTIFICATION**" button. The run result should be as follows:



23- Swap down the notification status bar to get the notification message details. You should get the following figure:


9:13

Sun, Jul 18 100%

Lesson07_Notification

Android ATC Notification
Check Android ATC New Course !!

Silent notifications

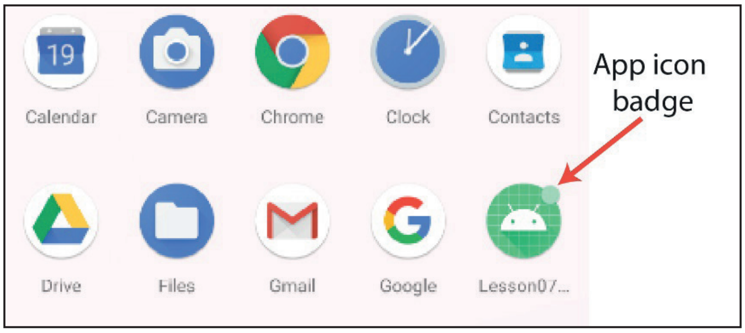
Android System

USB debugging connected
Tap to turn off USB debugging

Android System • Charging this device via USB

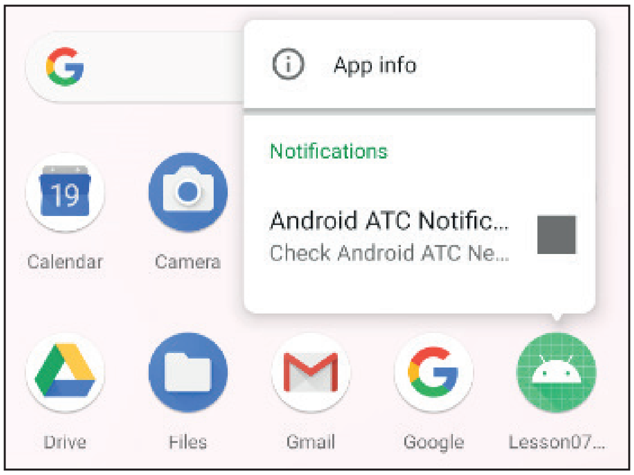
Manage Clear all

24- Click the **Home** button of your phone emulator and check your apps list. You will find your app icon Lesson07_Notification has an icon badge as illustrated in the following figure:



As you see here, this app icon indicates a new notification with a colored “badge” (also known as a “notification dot”) on the corresponding app launcher icon.

25- Long-press on your app icon to see the notifications for your app. You should get a notification similar to what you get in your phone emulator notification drawer as illustrated in the following figure:



Android ATC Notific...
Check Android ATC Ne...

26- In case you have a plan to make your app available too for the devices which have SDK version less than 26, you should add the following ***else*** statement code to your app as illustrated in the following code:
Note here you will use **`NotificationCompat.Builder`** class instead of **`Notification.Builder`** class.

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

    }

fun courseupdate(view: View) {

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

 // Create the Notification Channel
 val channel_id = "channel_01"
 val channel_Name="notification"
 val importance = NotificationManager.IMPORTANCE_DEFAULT
 val mChannel = NotificationChannel(channel_id, channel_Name,
importance)
     mChannel.description = "test description"
     mChannel.enableLights(true)
     mChannel.lightColor = Color.RED
     mChannel.enableVibration(true)

 // Use Notification.Builder to add the notification objects
val notification: Notification = Notification.Builder(this, channel_id)
     .setSmallIcon(R.drawable.ic_launcher_background)
     .setContentTitle("Android ATC Notification")
     .setContentText("Check Android ATC New Course !!")
     .build()
 // Register the channel with your Android system
 val mNotificationManager = getSystemService(NOTIFICATION_SERVICE)
as NotificationManager

  if (mNotificationManager != null) {
      mNotificationManager.createNotificationChannel(mChannel)
      mNotificationManager.notify(1, notification)
            }

    else
     {
   // When SDK version is less than 26
     val notification:Notification  =  NotificationCompat.
Builder(this, channel_id)
         .setSmallIcon(R.drawable.ic_launcher_background)
         .setContentTitle("Android ATC Notification")
         .setContentText("Check Android ATC New Course !!")
         .build()
      mNotificationManager.notify(1,notification)
            }

        }
    }
}
```

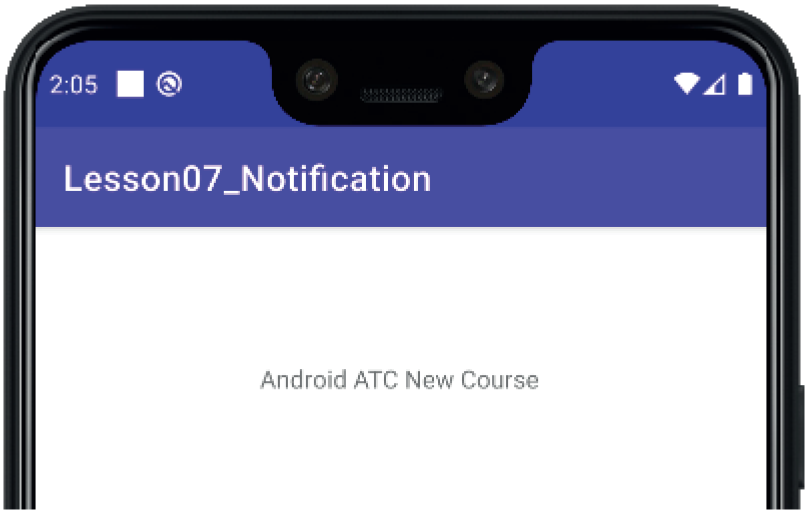
27- Now, you will add an Intent to your notification, where when the app user taps this notification, this app user will move to the other app activity (**ResultActivity**).
To do this, add only the gray highlighted part of the following code to your app code:

```
fun courseupdate(view: View) {

 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

// Create the Notification Channel
 val channel_id = "channel_01"
 val channel_Name="notification"
 val importance = NotificationManager.IMPORTANCE_DEFAULT
 val mChannel = NotificationChannel(channel_id, channel_Name,
importance)
     mChannel.description = "test description"
     mChannel.enableLights(true)
     mChannel.lightColor = Color.RED
     mChannel.enableVibration(true)
```

```
// Create an explicit intent for an Activity in your app
 val intent = Intent(this, ResultActivity::class.java).apply {
 flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_
CLEAR_TASK
  }
val pendingIntent: PendingIntent = PendingIntent.getActivity(this,
0, intent, 0)

 // Use Notification.Builder to add the notification objects
val notification: Notification = Notification.Builder(this, channel_
id)
    .setSmallIcon(R.drawable.ic_launcher_background)
    .setContentTitle("Android ATC Notification")
    .setContentText("Check Android ATC New Course !!")
// Set the intent that will fire when the user taps the notification
    .setContentIntent(pendingIntent)
    .build()
```

28- Double Click the **`Intent`** class, click the red pop-up lamp, and select **Import**.

29- **Stop**, then **Run** your app again. Tap the **Notification** button, and swap down your emulator status bar or notification area. If you tap your notification drawer, the intent configuration will open the **ResultActivity** interface as illustrated in the figure below:



Also, the same thing will happen if you tap your app badge, then tap your app notification content.