ACTIVITIES IN ANDROID

Introduction to activities

bookmark_border

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. 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.

This document introduces the concept of activities, and then provides some lightweight guidance about how to work with them. For additional information about best practices in architecting your app, see Guide to App Architecture.

The concept of activities

The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place. Instead, the user journey often begins non-deterministically. For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

The Activity class is designed to facilitate this paradigm. When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. In this way, the activity serves as the entry point for an app's interaction with the user. You implement an activity as a subclass of the Activity class.

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

Most apps contain multiple screens, which means 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 app. Each activity can then start another activity in order to perform different actions. For example, the main activity in a simple e-mail app may provide the screen that shows an e-mail inbox. From there, the main activity might launch other activities that provide screens for tasks like writing e-mails and opening individual e-mails.

Although activities work together to form a cohesive user experience in an app, each activity is only loosely bound to the other activities; there are usually minimal dependencies among the activities in an app. In fact, activities often start up activities belonging to other apps. For example, a browser app might launch the Share activity of a social-media app.

To use activities in your app, you must register information about them in the app's manifest, and you must manage activity lifecycles appropriately. The rest of this document introduces these subjects.

Configuring the manifest

For your app to be able to use activities, you must declare the activities, and certain of their attributes, in the manifest.

Declare activities

To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element. For example:

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

The only required attribute for this element is android:name, which specifies the class name of the activity. You can also add attributes that define activity characteristics such as label, icon, or UI theme. For more information about these and other attributes, see the <activity> element reference documentation.

Note: After you publish your app, you should not change activity names. If you do, you might break some functionality, such as app shortcuts. For more information on changes to avoid after publishing, see Things That Cannot Change.

Declare intent filters

Intent filters are a very powerful feature of the Android platform. They provide the ability to launch an activity based not only on an *explicit* request, but also an *implicit* one. For example, an explicit request might tell the system to "Start the Send Email activity in the Gmail app". By contrast, an implicit request tells the system to "Start a Send Email screen in any activity that can do the job." When the system UI asks a user which app to use in performing a task, that's an intent filter at work.

You can take advantage of this feature by declaring an <intent-filter> attribute in the <activity> element. The definition of this element includes an <action> element and, optionally, a <category> element and/or a <data> element. These elements combine to specify the type of intent to which your activity can respond. For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
                                                    <intent-filter>
                            <action android:name="android.intent.action.SEND" />
                        <category android:name="android.intent.category.DEFAULT" />
                                    <data android:mimeType="text/plain" />
                                                    </intent-filter>
</activity>
```

In this example, the <action> element specifies that this activity sends data. Declaring the <category> element as DEFAULT enables the activity to receive launch requests. The <data> element specifies the type of data that this activity can send. The following code snippet shows how to call the activity described above:

KotlinJava

```
val sendIntent = Intent().apply {
    action = Intent.ACTION_SEND
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, textMessage)
}
startActivity(sendIntent)
```

If you intend for your app to be self-contained and not allow other apps to activate its activities, you don't need any other intent filters. Activities that you don't want to make available to other applications should have no intent filters, and you can start them yourself using explicit intents. For more information about how your activities can respond to intents, see Intents and Intent Filters.

Declare permissions

You can use the manifest's <activity> tag to control which apps can start a particular activity. A parent activity cannot launch a child activity unless both activities have the same permissions in their manifest. If you declare a <uses-permission> element for a parent activity, each child activity must have a matching <uses-permission> element.

For example, if your app wants to use a hypothetical app named SocialApp to share a post on social media, SocialApp itself must define the permission that an app calling it must have:

```
<manifest>
<activity android:name="...."
    android:permission=”com.google.socialapp.permission.SHARE_POST”
```

/>

Then, to be allowed to call SocialApp, your app must match the permission set in SocialApp's manifest:

```
<manifest>
  <uses-permission android:name="com.google.socialapp.permission.SHARE_POST" />
</manifest>
```

For more information on permissions and security in general, see Security and Permissions.

Managing the 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. The following sections introduce these callbacks.

onCreate()

You must implement this callback, which fires when the system creates your activity. Your implementation should initialize the essential components of your activity: For example, your app should create views and bind data to lists here. Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.

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

onStart()

As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user. This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.

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. Most of an app's core functionality is implemented in the onResume() method.

The onPause() callback always follows onResume().

onPause()

The system calls onPause() when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or Recents button. When the system calls onPause() for your activity, it technically 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.

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

You should **not** use onPause() to save application or user data, make network calls, or execute database transactions. For information about saving data, see Saving and restoring activity state.

Once onPause() finishes executing, the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused state.

onStop()

The system calls onStop() when the activity is no longer visible to the user. This may happen because the activity is being destroyed, 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 at all.

The next callback that the system calls is either onRestart(), if the activity is coming back to interact with the user, or by onDestroy() if this activity is completely terminating.

onRestart()

The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.

This callback is always followed by onStart().

onDestroy()

The system invokes this callback before an activity is destroyed.

This callback is the final one that the 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.

This section provides only an introduction to this topic. For a more detailed treatment of the activity lifecycle and its callbacks, see The Activity Lifecycle.

Android Layout and Views – Types and Examples

Welcome back to **DataFlair Android Tutorial series**. In this article, we'll learn about **Android Layout and Views**. Let us begin with what is a View and then move to Layout.

What is Android View?

A View is a simple building block of a user interface. It is a small rectangular box that can be TextView, EditText, or even a button. It occupies the area on the screen in a rectangular area and is responsible for drawing and event handling. View is a superclass of all the graphical user interface components.

Why and How to use the View in Android?

Now you might be thinking what is the use of a View. So, the use of a view is to draw content on the screen of the user's Android device. A view can be easily implemented in

an Application using the java code. Its creation is more easy in the XML layout file of the project. Like, the project for hello world that we had made initially.

If you have not tried it, refer DataFlair hello world app in Android.

Types of Android Views

Another thing that might now come to your mind must be, "what are the available types of view in Android that we can use?"
For that, we'll see all these types one by one as follows:

- TextView
- EditText
- Button
- Image Button
- Date Picker
- RadioButton
- CheckBox buttons
- Image View

And there are some more components. Learn more about Android UI Controls.
Another important feature in Android is **ViewGroup** which is as follows.

What is Android View Group?

A View Group is a subclass of the ViewClass and can be considered as a superclass of Layouts. It provides an invisible container to hold the views or layouts. ViewGroup instances and views work together as a container for Layouts. To understand in simpler words it can be understood as a special view that can hold other views that are often known as a child view.

Following are certain commonly used subclasses for ViewGroup:

- LinearLayout
- RelativeLayout
- FrameLayout
- GridView
- ListView

Here is how Views and ViewGroups are linked:

Now we'll move towards the Android layouts:

What is Android Layout?

Layout basically refers to the arrangement of elements on a page these elements are likely to be images, texts or styles. These are a part of **Android Jetpack**. They define the structure of android user interface in the app, like in an activity. All elements in the layout are built with the help of Views and ViewGroups. These layouts can have various widgets like buttons, labels, textboxes, and many others.
We can define a Layout as follows:

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



 <LinearLayout

    android:id="@+id/layout2"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:layout_weight="1"

    android:background="#8ED3EB"

    android:gravity="center"

    android:orientation="vertical" >
```

```
    <TextView

        android:id="@+id/textView4"

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:layout_marginLeft="10dp"

        android:layout_marginTop="-40dp"

        android:fontFamily="@font/almendra_bold"

        android:text="This is a TextView" />



 </LinearLayout>
```

Attributes of Layout in Android

The following are the attributes for customizing a Layout while defining it:

- **android:id:** It uniquely identifies the Android Layout.
- **android:hint:** It shows the hint of what to fill inside the EditText.
- **android:layout_height:** It sets the height of the layout.
- **android:layout_width:** It sets the width of the layout.
- **android:layout_gravity:** It sets the position of the child view.

- **android:layout_marginTop:** It sets the margin of the from the top of the layout.
- **android:layout_marginBottom:** It sets the margin of the from the bottom of the layout.
- **android:layout_marginLeft:** It sets the margin of the from the left of the layout.
- **android:layout_marginRight:** It sets the margin of the from the right of the layout.
- **android:layout_x:** It specifies the x coordinates of the layout.
- **android:layout_y:** It specifies the y coordinates of the layout.

Types of Layouts in Android

Now that we've learned about the view and view groups and also somewhat about the layouts. Subsequently let us see the types of Layouts in Android, that are as follows:

- Linear Layout
- Relative Layout
- Constraint Layout
- Table Layout
- Frame Layout
- List View
- Grid View
- Absolute Layout
- WebView
- ScrollView

These are the types of layouts and out of them we'll learn about the two very important layouts:

1. Linear Layout

We use this layout to place the elements in a linear manner. A Linear manner means one element per line. This layout creates various kinds of forms on Android. In this, arrangement of the elements is in a top to bottom manner.

This can have two orientations:

a. Vertical Orientation – It is shown above where the widgets such as TextViews, and all in a vertical manner.

b. Horizontal Orientation – It is shown above where the widgets such as TextViews, and all in a horizontal manner.

2. Relative Layout

This layout is for specifying the position of the elements in relation to the other elements that are present there.

In the relative layout, alignment of the position of the elements to the parent container is possible. To define it in such a way, we write the following:

- android:layout_alignParentTop= “true”
- android:layout_alignParentLeft= “true”

If we write the above code, the element will get aligned on the top left of the parent container.

If we want to align it with some other element in the same container, it can be defined is as follows:

- android:layout_alignLeft= “@+id/element_name”
- android:layout_below= “@+id/element_name”

This will align the element below the other element to its left.

Here are the pictorial representations of different layouts-