## Objective 1 : Understanding the activities and its lifecycle

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.

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.

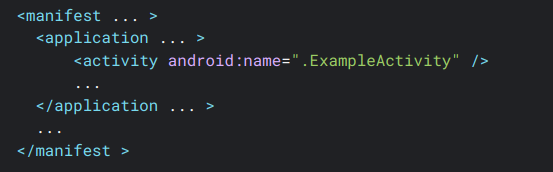
The Activity class is designed to facilitate this paradigm. When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. In this way, the activity serves as the entry point for an app's interaction with the user. You implement an activity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

### Declare activities

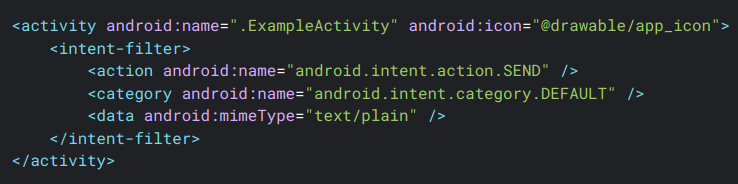
To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element. For example:



### 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:



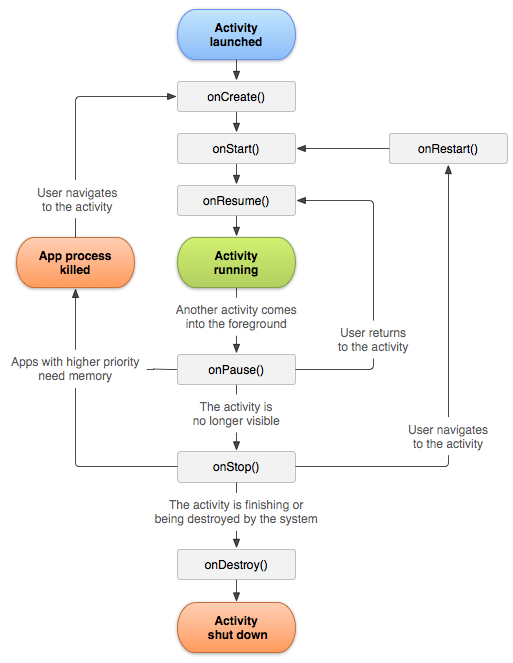
### The activity lifecycle

As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides a number of callbacks that let the activity know when a state changes or that the system is creating, stopping, or resuming an activity or destroying the process the activity resides in.

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and let the user resume the video from the same spot.

Each callback lets you perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant. For example, good implementation of the lifecycle callbacks can help your app avoid the following:

* Crashing if the user receives a phone call or switches to another app while using your app.
* Consuming valuable system resources when the user is not actively using it.
* Losing the user's progress if they leave your app and return to it at a later time.
* Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

Figure 1: Represents a visual representation Activity Lifecycle.

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). The system invokes each of these callbacks as the activity enters a new state.

## Objective 2 : Navigating Between Activities - Intents

An app is likely to enter and exit an activity, perhaps many times, during the app’s lifetime, such as when the user taps the device’s Back button or the activity launches a different activity.

This section covers topics you need to know to implement successful activity transitions. These topics include starting an activity from another activity, saving activity state, and restoring activity state.

An activity often needs to start another activity at some point. This need arises, for instance, when an app needs to move from the current screen to a new one.

Depending on whether or not your activity wants a result back from the new activity it’s about to start, you start the new activity using either the **startActivity**() method or the **startActivityForResult**() method. In either case, you pass in an Intent object.

The Intent object specifies either the exact activity you want to start or describes the type of action you want to perform. The system selects the appropriate activity for you, which can even be from a different application. An Intent object can also carry small amounts of data to be used by the activity that is started.

### Intent

An intent is an abstract description of an operation to be performed. It can be used with startActivity to launch an Activity, broadcastIntent to send it to any interested BroadcastReceiver components, and Context.startService(Intent) or Context.bindService(Intent, BindServiceFlags, Executor, ServiceConnection) to communicate with a background Service.

An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed.

#### Intent Structure

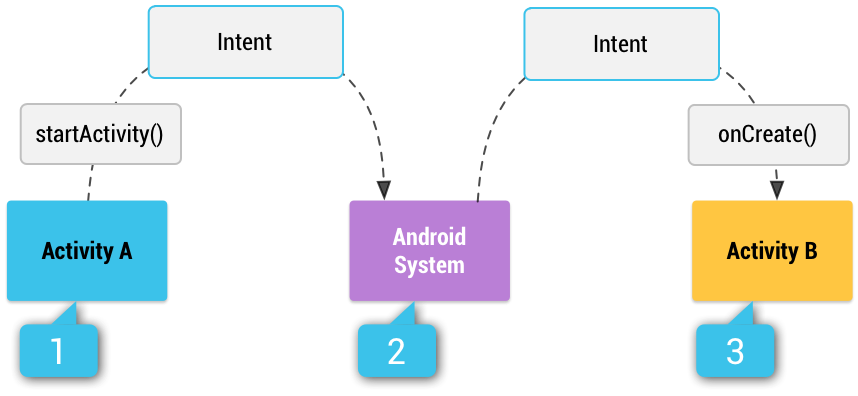
The primary pieces of information in an intent are:

* **action** -- The general action to be performed, such as ACTION\_VIEW, ACTION\_EDIT, ACTION\_MAIN, etc.
* **data** -- The data to operate on, such as a person record in the contacts database, expressed as a Uri.

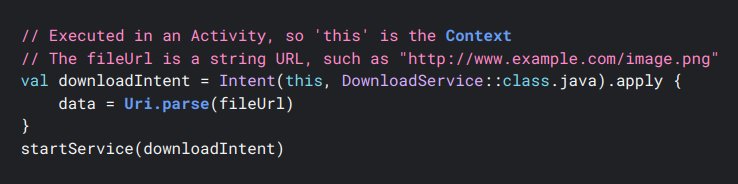
### Intent Resolution

There are two primary forms of intents:

1. **Explicit Intents** have specified a component (via setComponent(ComponentName) or setClass(Context, Class)), which provides the exact class to be run. Often these will not include any other information, simply being a way for an application to launch various internal activities it has as the user interacts with the application.
2. **Implicit Intents** have not specified a component; instead, they must include enough information for the system to determine which of the available components is best to run for that intent.



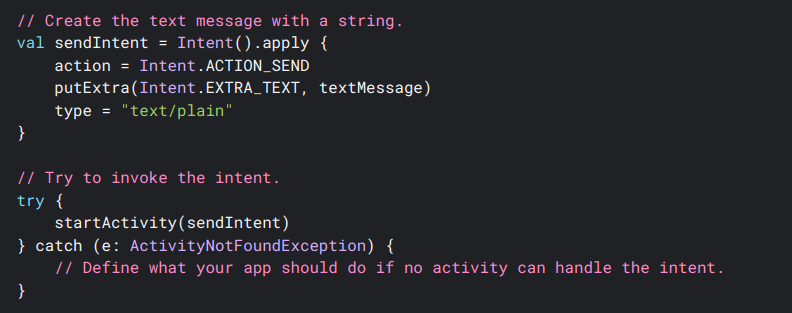
**Example of Explicit Intent**



**Example implicit intent**

An implicit intent specifies an action that can invoke any app on the device able to perform the action. Using an implicit intent is useful when your app cannot perform the action, but other apps probably can and you'd like the user to pick which app to use.

For example, if you have content that you want the user to share with other people, create an intent with the ACTION\_SEND action and add extras that specify the content to share. When you call startActivity() with that intent, the user can pick an app through which to share the content.



## Objective 3 : Navigation in Compose

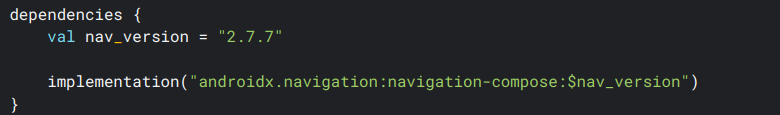
The Navigation component provides support for Jetpack Compose applications. You can navigate between composables while taking advantage of the Navigation component's infrastructure and features.

The Navigation Component is made up of three major parts:

* **Navigation Graph**: This is a resource that collects all navigation-related data in one place. This includes all of the locations in your app, referred to as destinations, as well as the possible paths a user could take through your app. It’s like a big book that has all the places you can go in an app and how you can move between them. Think of it as a map and a guide combined.
* **NavHost:** This is a unique composable that you can include in your layout. It shows various destinations from your Navigation Graph. The NavHost links the NavController with a navigation graph that specifies the composable destinations that you should be able to navigate between. As you navigate between composables, the content of the NavHost is automatically recomposed. Each composable destination in your navigation graph is associated with a route.
* **NavController:** The NavController is the central API for the Navigation component. It is stateful and keeps track of the back stack of composables that make up the screens in your app and the state of each screen.

#### Setup

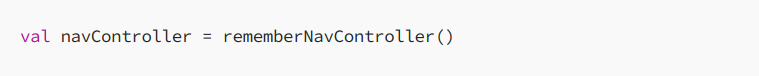
To support Compose, use the following dependency in your app module's **build.gradle** file:



**NavController:**

The NavController is the central API for the Navigation component. It is stateful and keeps track of the back stack of composables that make up the screens in your app and the state of each screen.

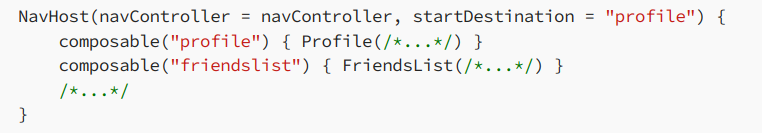
You can create a NavController by using the rememberNavController() method in your composable:



**NavHost:**

Each NavController must be associated with a single NavHost composable. The NavHost links the NavController with a navigation graph that specifies the composable destinations that you should be able to navigate between. As you navigate between composables, the content of the NavHost is automatically recomposed. Each composable destination in your navigation graph is associated with a route.

Creating the NavHost requires the NavController previously created via rememberNavController() and the route of the starting destination of your graph. NavHost creation uses the lambda syntax from the Navigation Kotlin DSL to construct your navigation graph. You can add to your navigation structure by using the composable() method. This method requires that you provide a route and the composable that should be linked to the destination:



**Navigate Without Arguments:**

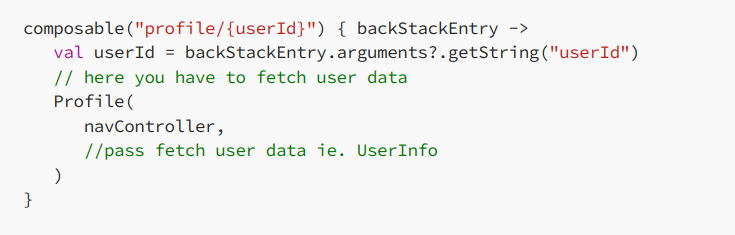


**Navigate With Simple Arguments:**

By default, all arguments are parsed as strings. The arguments parameter of composable() accepts a list of NamedNavArguments. You can quickly create a NamedNavArgument using the navArgument method and then specify its exact type:



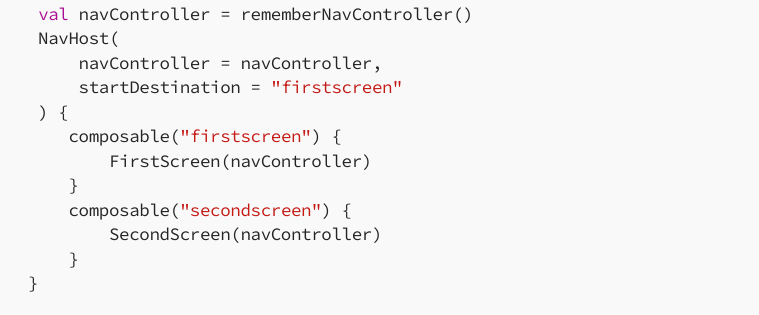
You should extract the arguments from the NavBackStackEntry that is available in the lambda of the composable() function



**Navigating back with Result**

Navigate back with result is most common task. ie. When you open filter dialog and select filter and then navigate back with selected filter to apply those filters in your screen.

There are two screens. 1. FirstScreen and 2. SecondScreen. We need data from SecondScreen to our FirstScreen.



Retrieve data after navigating back from SecondScreen with the help of savedStateHandle of current back-stack entry of NavController



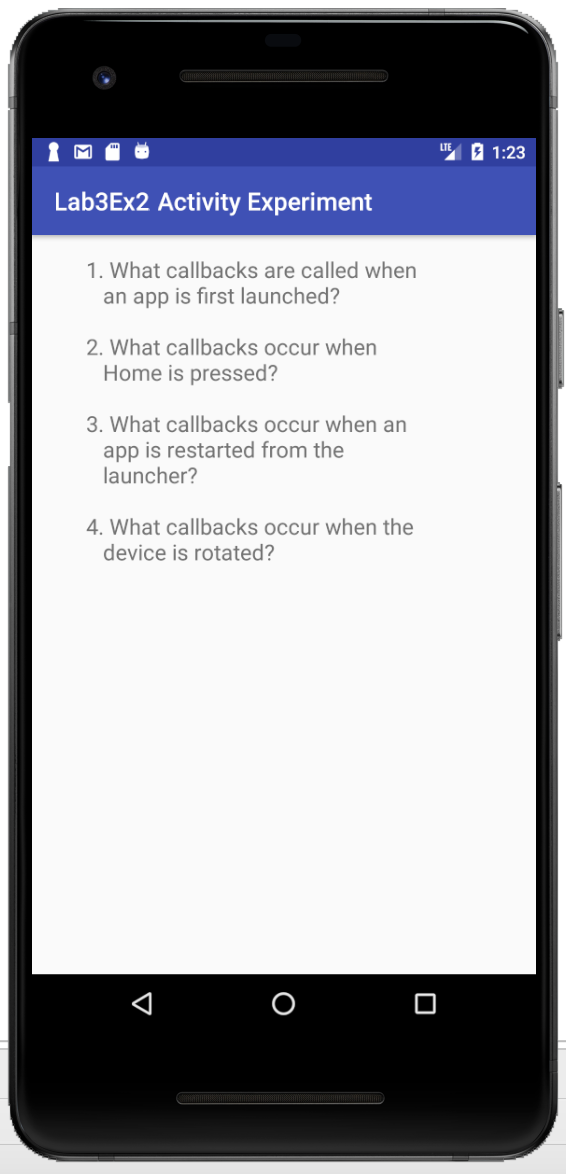
Sending data from second to first screen



## Objective 4 – Practice Tasks

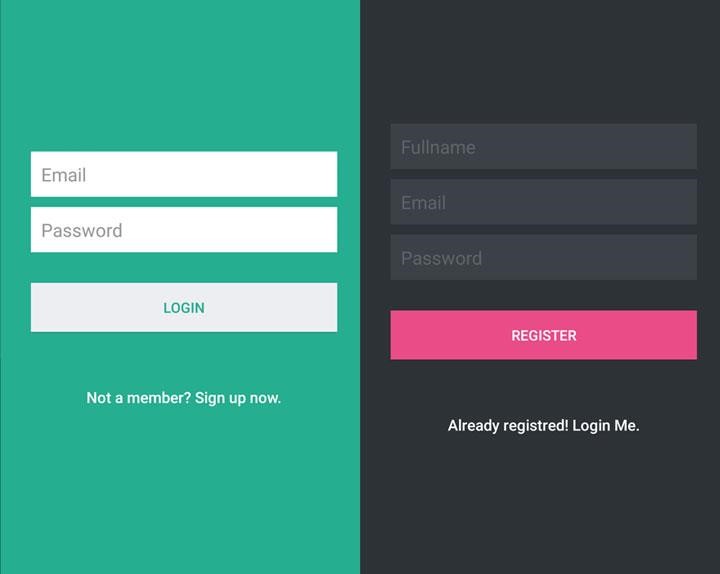
### Activity 1 :

Test all the activity methods and check which calls are called on the following actions:



### Activity 2:

Create two screen, login and sign up. Traverse user between the two screens on clicking the appropriate buttons.



### Activity 3 : Create a Freindsr App

* Create a new project
* Add 6 image boxes and assign images accordingly
* Add Onclick event on all images.
* When clicked any image, use proper action to move to second screen with more detail of the clicked person (Use iamges, RatingsBar and a labels to show all data)
* Data is provided, use it appropriately.

Hint: Use following method to get data from array created in strings.xml file

