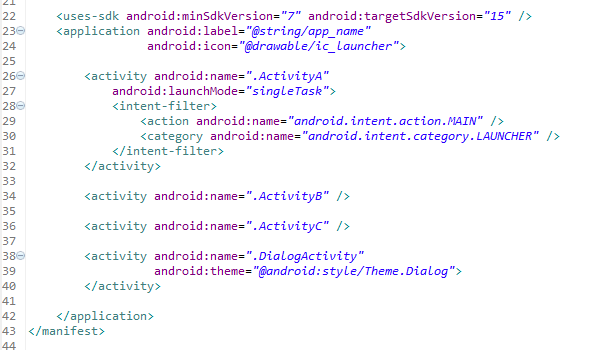
## Android Manifest

The main function of the manifest xml file is to give some basic information for the app store and operating system.

The manifest file starts off with an opening XML tag that contains the version and character encoding for the file. The next tag in the file is the manifest tag. This includes the package name, version, and version name. The uses-sdk tag identifies the minimum SDK version and the target sdk. This will help the “builder” when the code is turned into bytecode. Some older devices won’t run the latest SDK versions. The application tag gives a name and an icon to the OS to display in the list of applications. Within the opening and closing application tags, we have four activities.

The activity tags represent the controllers of the app, or “Activities”. Activity A is the initial activity to be used. It has an intent filter used to control how it is launched. Activity B and C are listed, then finally the dialog activity is listed. The dialog activity has a theme attribute that allows it to act as a dialog pop up instead of a full-window activity like A, B, or C. The next two lines given are the closing application and manifest tags which bring the file to an end.



## 

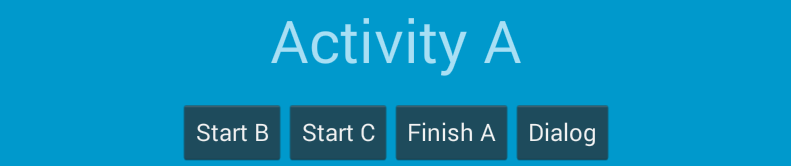
## 

## 

## Layout

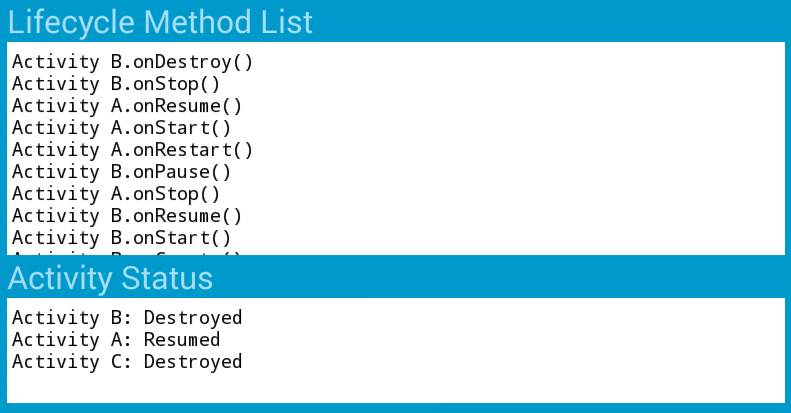
The activity app uses several layouts created with XML. XML is a common markup language used for creating graphical user interfaces. It allows designers to implement their design in a structured way, typically rather platform agnostic so it is a widely accepted standard.. These layouts are in the res>layout folder and are labelled activity\_a, activity\_b, activity\_c, and activity\_dialog.

Activity\_A’s XML layout file starts with a XML tag that let’s the sdk packaging software know what kind of information it’s about to get. The initial tag is a linear layout tag. This will display everything in a single column arranged from top to bottom within the opening and closing tags. Under the opening layout tag is the first view displayed to the screen. The text view is used to display a text header for the activity. Another layout tag, a relative layout, is used to arrange the next group of views below the header. A relative layout allows the views to arrange themselves relative to a parent or sibling. This app in particular uses this layout to arrange the buttons side by side, near the top of the screen. These are the control buttons that will eventually call new activities.

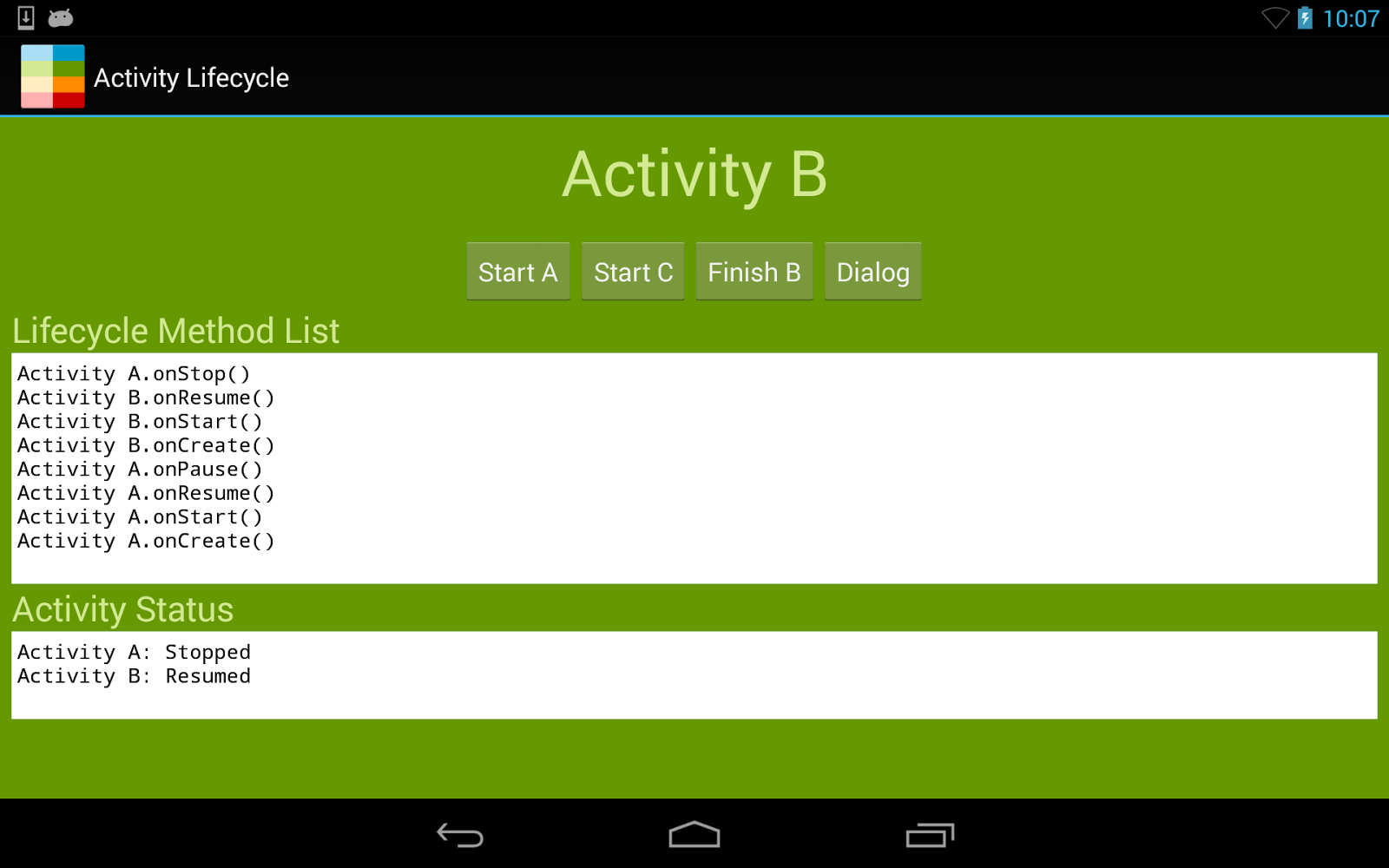


The layout tag is closed and the last four views are written in. These new textviews belong to the LinearLayout element (as that layout tag has yet to be closed).

The first and third textviews are used as headers for textviews 2 and 4. The fourth textview will only ever be 3 lines tall and is fairly static except for the status changes. The second textview will continually fill with data and unless something is done the information will be lost to the user. To prevent this, the view is wrapped with a scrollview to allow the user to scroll through the data within the whitespace.



Activity B and C’s layouts are almost identical to activity A’s. The exceptions are the color, header, and the buttons available for launching other activities, which allows the user to tell which activity is active and have some control over the application.



The final layout is the dialog layout. It is a very simple linear layout with a button to allow it to close. This layout will eventually be displayed on top of other activities with a special method within the dialog activity java file.

**Java Activity Class Code**

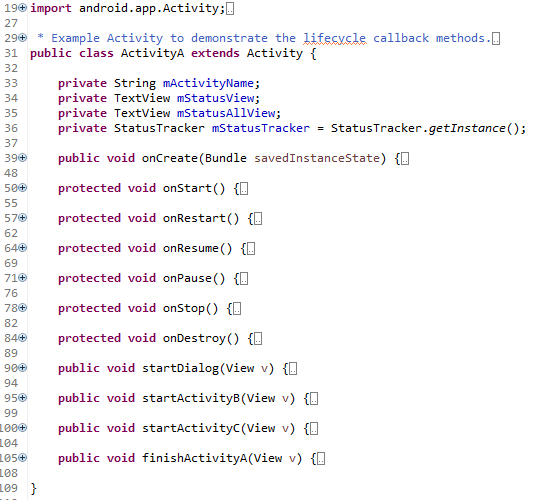
The source code responsible for driving the app is in the java folder. All of the user created packages will be stored in this folder. The androidmanifest.xml file indicates activity A as the first activity to launch.

Like most graphical or multitasking operating systems, the OS operates on a system of messages, events, and callbacks. This means the program doesn’t necessarily have a main function that is performed until the program finishes. Instead a class is created that inherits from an android.app.Activity class. This allows the Activity to fire off different methods at different events and stages of its life.

One of the most important methods is the onCreate method. This bit of code is the first function that will execute as the Activity is instantiated, followed by the onStart method, and finally the onResume method. These methods are indicated as Overrides by the @Override attribute (an example can be seen in ActivityA.java, line 38), and as such indicate that they come from a parent class. These methods in fact originate in the Activity class, provided by the android.app package.

After these three functions have fired, the app is running and has focus. If the app loses focus, say to the dialog box activity, it will use the onPause method. If the app is stopped and is no longer displayed to the screen it will call the onStop method. If the Operating system is in need of more resources and the app is stopped it will use the onDestroy method thus completing the cycle of life. It is also worth mentioning the restart method in case the app would rather use a specific method that is not the start method.

The remaining methods are event handlers for the buttons at the top of the screen. They will create and finish activities for the user by sending Intents to the OS. The Intent class is provided by the android.content package, and is essentially a description of some activity for the operation system to perform. These intents will then activate the other activities and start the whole process again.



**Launching the App**

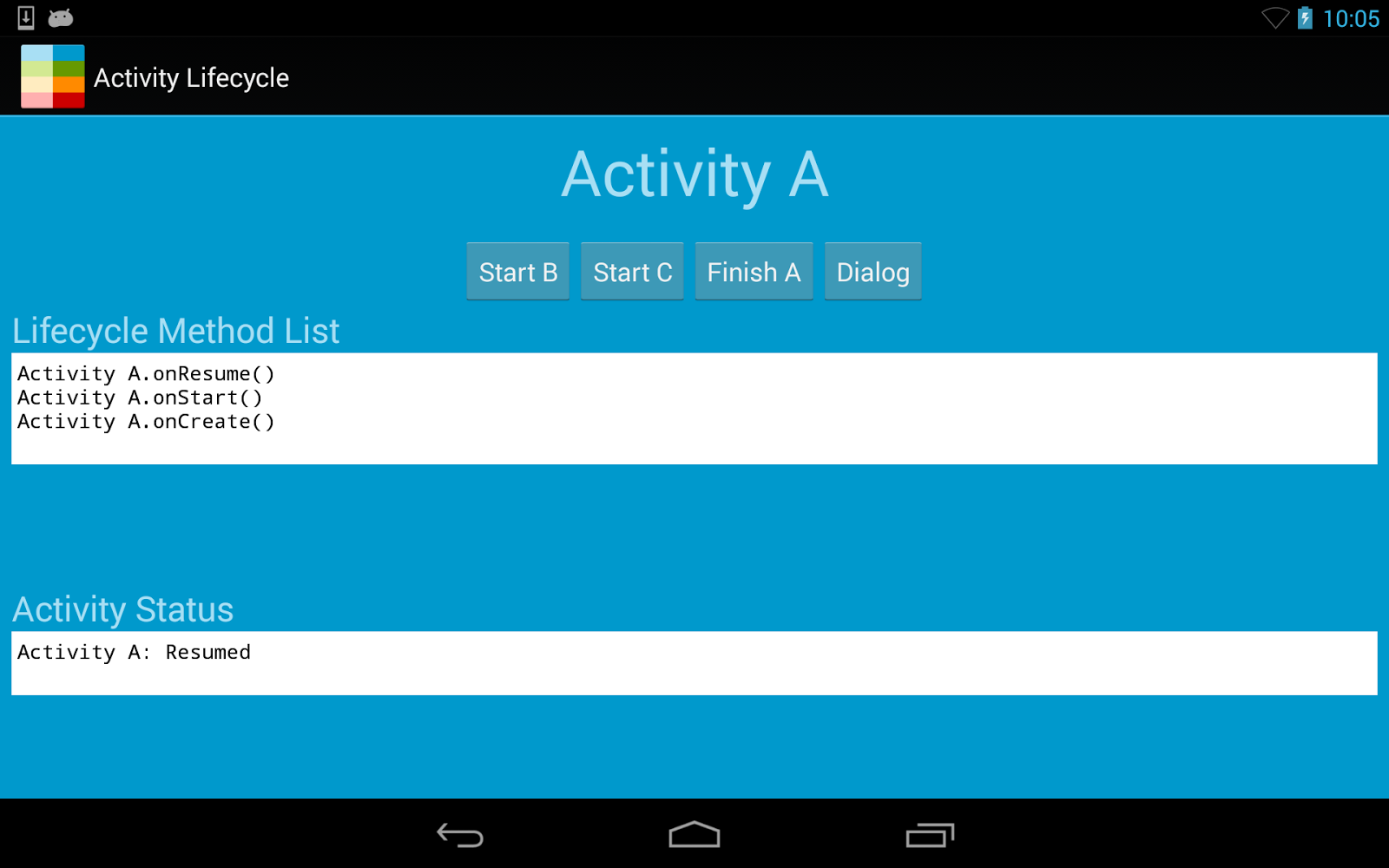
Launching the app, we’re greeted with a blue screen labelled Activity A. Within the layout for this activity, there are 4 buttons (from android.widget package) and two TextViews (from android.widget as well), one of which displaying the overall lifecycle method list, the second of which shows the individual activity’s current status. Upon launching, we wind up with the following:

Activity A.onResume()

Activity A.onStart()

Activity A.onCreate()

This shows the call stack at this point for the lifecycle methods. The main activity launches (Activity A), runs through onCreate, onStart and onResume and then holds, waiting for input.

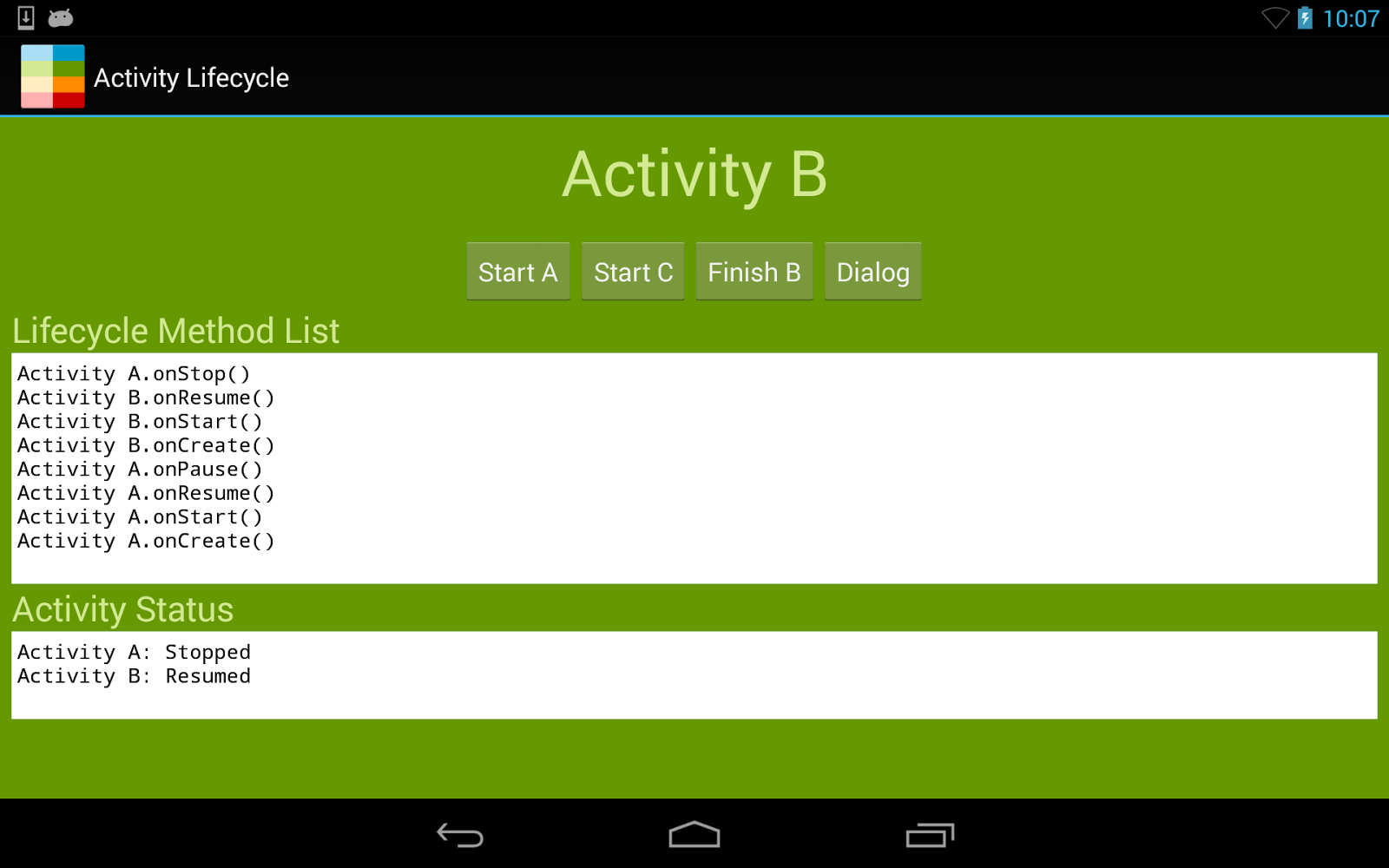


In this scenario, the operating system is calling those methods in order from onCreate, onStart, and onResume within Activity A’s class, located at lines 39, 50, and 64 respectively. Effectively, Android is telling our ActivityA class to execute those methods in that order to get our activity running. These methods are brought in from the Activity class and overridden to add additional functionality.

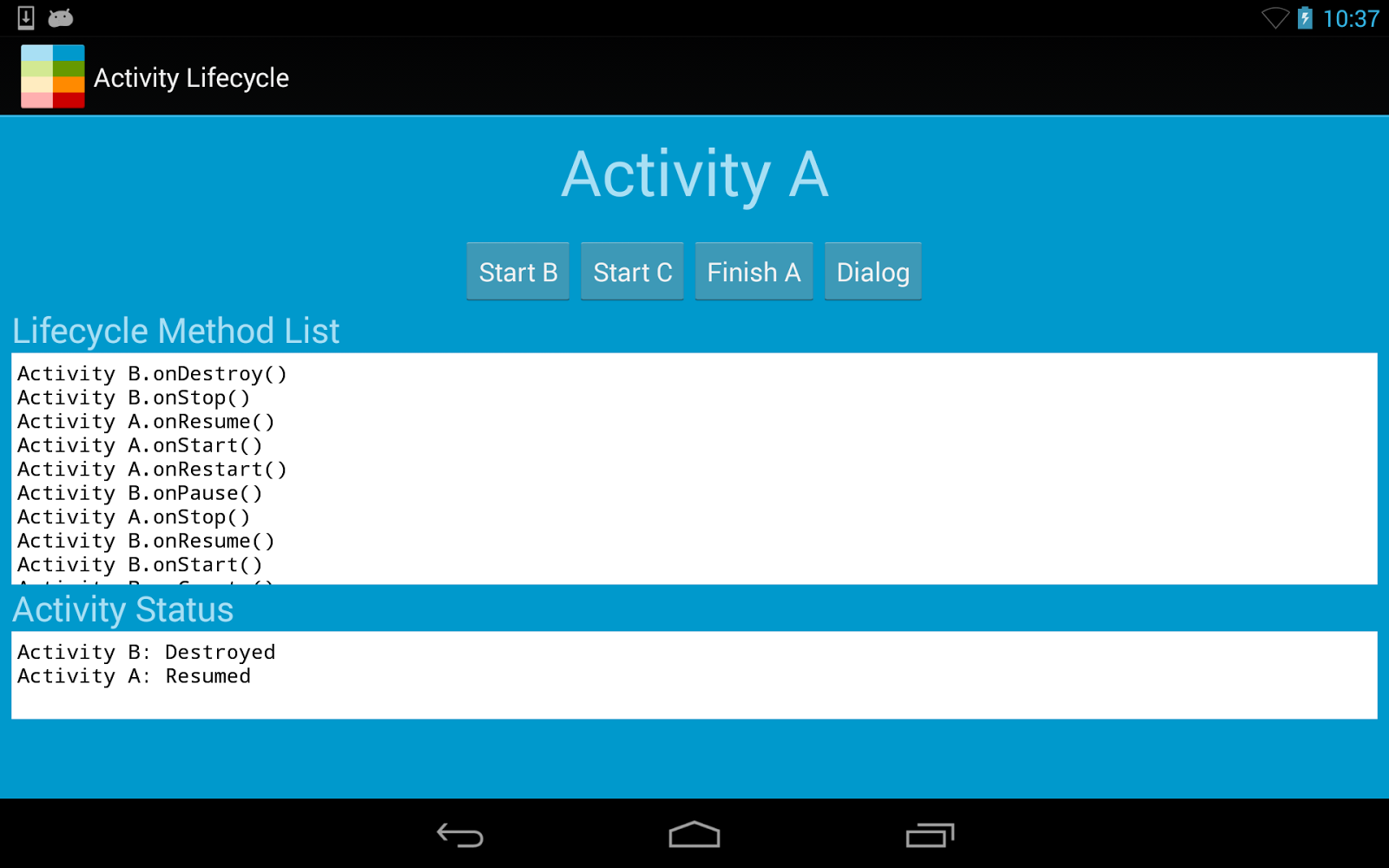
Within onCreate, ActivityA calls Activity’s onCreate method from the super call, then proceeds to set some variables within the class. It then indicates to its reference to the StatusTracker class (com.example.android.lifecycle.util.StatusTracker) that ActivityA’s onCreate method override has been called, and retrieves the current state of the lifecycle application method calls from the singleton and prints it out to the textview embedded within a scrollview located within activity\_a.xml on line 88.

This is achieved by using the Utils (com.example.android.lifecycle.util.Utils) static method printStatus, taking in a target TextView to populate with methods called, and another target TextView to populate with current Status.

Clicking the button “Start B” will bring up Activity B. First, Activity A has to be paused, followed by Activity B being created, started, and resumed. Once Activity B is running and in focus, Activity A will be stopped because it is no longer being displayed on screen. This can be seen in Figure 2 below. Again, this is all given to us by android’s standard lifecycle - in order for B to be started, we first must pause ActivityA.

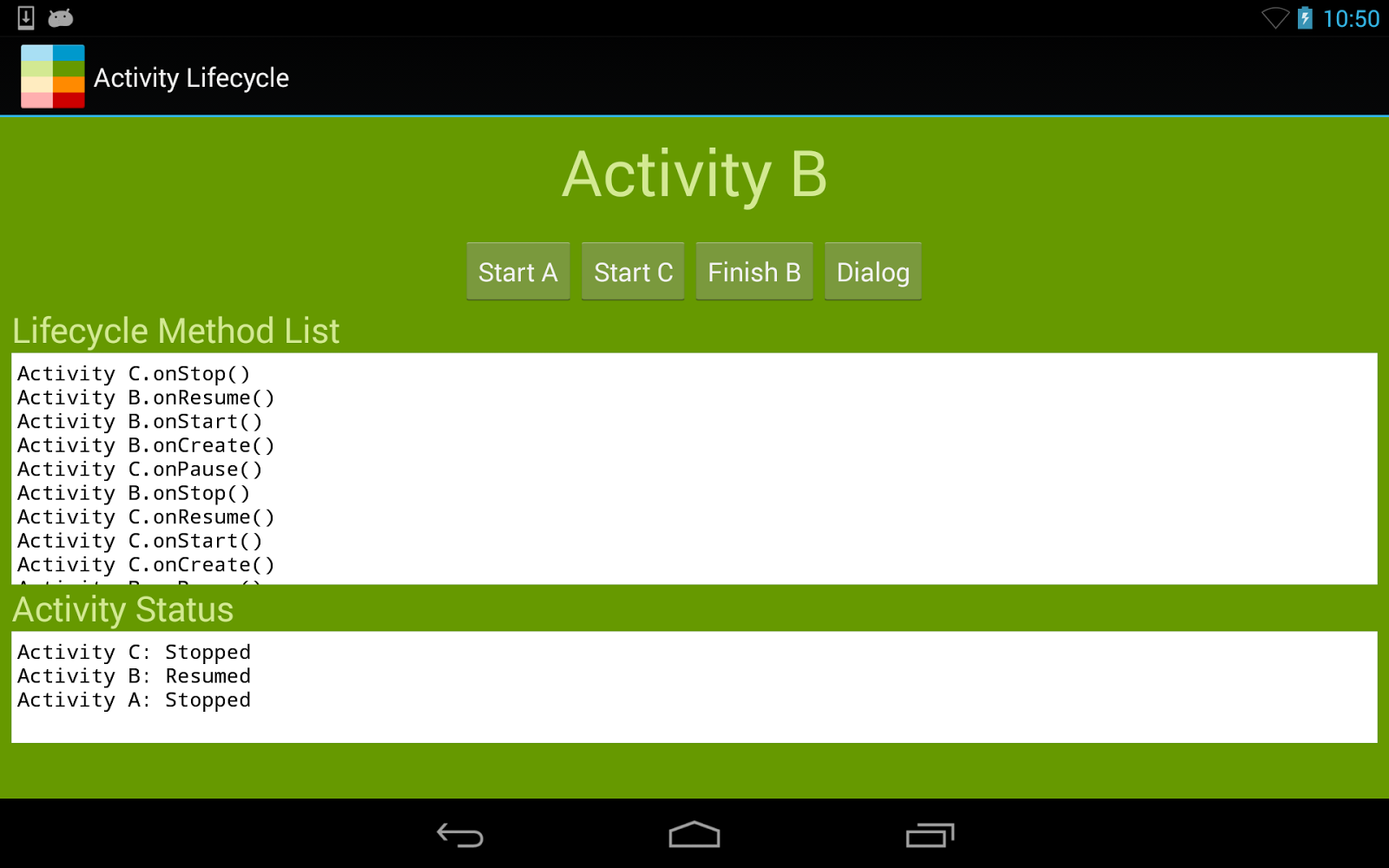


At this point, we can click Start A, C, Finish B, or go into Dialog. We went ahead and clicked “Finish B”. Viewing the layout file for Activity B shows the button as having an onClick attribute of finishActivityB, thereby firing that method in ActivityB when pressed, resulting in Activity B being paused and Activity A having onRestart, onStart, and onResume fired. Finally, onStop and onDestroy fire on Activity B after Activity A is fully in focus and Activity B is popped from the stack. This can be viewed in Figure 3 below.



Activities B and C are identical aside from labeling and color. Each time a button is hit to start either B or C, a new activity is created regardless if there is already a B or C in existence. This is visible within the Activity’s methods startActivityA, startActivityB or startActivityC. Each one spins up a new intent to start a given Activity for each of the given Activity types. However, Activity A is your Main entry point, and this is the entry point to all future activities pushed onto the stack after it.

This allows you to back through multiple B and C activities until you get to an A activity. Starting a new A activity while the program is running will cause all B & C activities to be destroyed, as they are necessarily above Activity A on the Activity stack managed by android. On an example run, we start with Activity A, click Start B -> Start C -> Start B, resulting in a Lifecycle Method List as seen below.

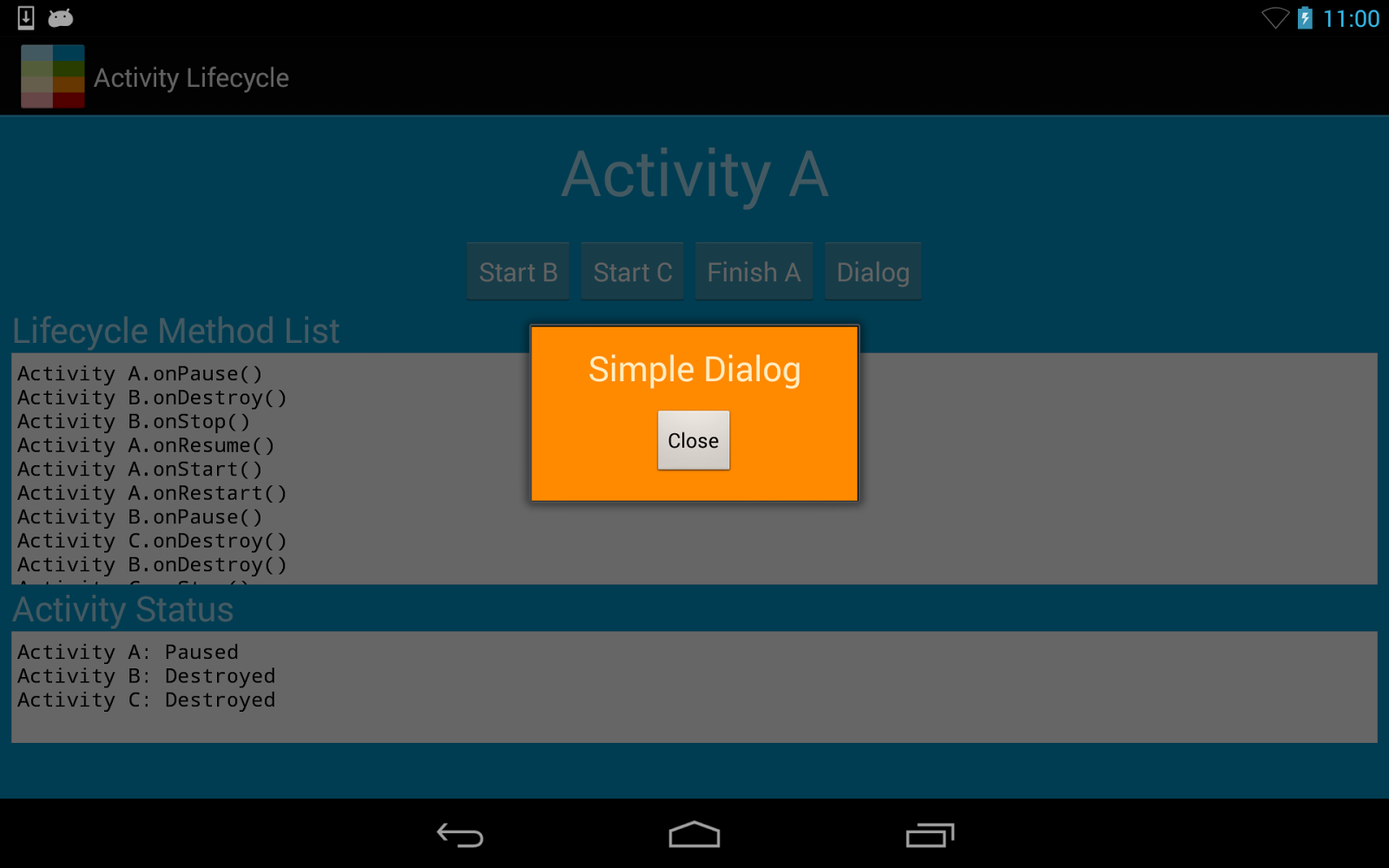


At this point, if we tap Start A, we wind up with an Activity Status of Activity B and C destroyed, while A is Resumed, as seen in Figure 5 Below. This demonstrates the stack-based nature of the Activity Management by Android, and how children activities spawned from a parent will be destroyed upon a new instance of that activity reinstantiated.



Figure 5 - Child Activities of previous Activity A destroyed, new Activity A started.

In addition, each of the activities has the Dialog button to demonstrate the usage of the requestWindowFeature method. In the onCreate method of the DialogActivity, the Activity declares itself as a modal through the usage of the constant Window.FEATURE\_NO\_TITLE. Upon tapping the button, a modal dialog appears with a close button, with the previous Activity (Activity A) visible in the background. From that you can see the Lifecycle Method List, indicating that Activity A has gone to a paused state while the modal dialog is open.



Closing the dialog resumes Activity A and you can start the cycle over again. The main gist of the app is that there is only ever one Activity A as that spawns all other Activities, and any child activities of Activity A will be destroyed upon a relaunch from intent for Activity A.