

Week 4 - Part1

Building User Screen Flows

This week, we will cover the Android activity lifecycle and explain how the Android system interacts with your app. By the end of this week, you'll have learned how to build user journeys through different screens. You'll also be able to use activity tasks and launch modes, save and restore the state of your activity, use logs to report on your application, and share data between screens.

In the week before we embarked on holiday, you were introduced to the core elements of Android development, from configuring your app using the `AndroidManifest.xml` file, working with simple activities, and the Android resource structure to building an app with `gradle` and running an app on a virtual device. You were then jumped to Building App Navigation, which was supposed to be content of week 5. However, that was deliberately done.

In this week, you'll go further and learn how the Android system interacts with your app through the Android lifecycle, how you are notified of changes to your app's state, and how you can use the Android lifecycle to respond to these changes.

You'll then progress to learning how to create user journeys through your app and how to share data between screens. You'll be introduced to different techniques to achieve these goals so that you'll be able to use them in your own apps and recognize them when you see them used in other apps.

We will cover the following topics during this week's session:

- The Activity lifecycle
- Saving and restoring the Activity state
- Activity interaction with Intents
- Intents, Tasks, and Launch Modes

The Activity lifecycle

In the previous week when we started using android after our introduction to Kotlin, we used the `onCreate(saveInstanceState: Bundle?)` method to display a layout in the UI of our screen. Now, we'll explore in more detail how the Android system interacts with your application to make this happen. As soon as an Activity is launched, it goes through a series of steps to take it through initialization, from preparing to be displayed to being partially displayed and then fully displayed.

There are also steps that correspond with your application being hidden, backgrounded, and then destroyed. This process is called the **Activity lifecycle**. For every one of these steps, there is a **callback** that your Activity can use to perform actions such as creating and changing the display and saving data when your app has been put into the background and then restoring that data after your app comes back into the foreground.

These callbacks are made on your Activity's parent, and it's up to you to decide whether you need to implement them in your own Activity to take any corresponding action. Each of these callback functions has the `override` keyword. The `override` keyword in Kotlin means that either this function is providing an implementation of an interface or an abstract method, or, in the case of your Activity here, which is a subclass, it is providing the implementation that will override its parent.

Now that you know how the Activity lifecycle works in general, let's go into more detail about the principal callbacks you will work with in order, from creating an Activity to the Activity being destroyed:

- `override fun onCreate(savedInstanceState: Bundle?):` This is the callback that you will use the most for activities that draw a full-sized screen. It's here where you prepare your Activity layout to be displayed. At this stage, after the method has completed, it is still not displayed to the user, although it will appear that way if you don't implement any other callbacks. You usually set up the UI of your Activity here by calling the `setContentView(R.layout.activity_main)` method and carrying out any initialization that is required.

This method is only called once in its lifecycle unless the Activity is created again. This happens by default for some actions (such as rotating the phone from portrait to landscape orientation). The `savedInstanceState` parameter of the `Bundle?` type (`?` means the type can be null) in its simplest form is a map of key-value pairs optimized to save and restore data.

It will be null if this is the first time that the Activity has been run after the app has started, if the Activity is being created for the first time, or if the Activity is being recreated without any states being saved.

- `override fun onStart():` When the Activity restarts, this is called immediately before `onStart()`. It is important to be clear about the difference between restarting an Activity and recreating an activity. When the Activity is backgrounded by pressing the home button, when it comes back into the foreground again `onRestart()` will be called. Recreating an Activity is what happens when a configuration change happens, such as the device being rotated. The Activity is finished and then created again, in which case `onRestart()` will not be called.
- `override fun onStart():` This is the first callback made when the Activity is brought from the background to the foreground.
- `override fun onRestoreInstanceState(savedInstanceState: Bundle?):` If the state has been saved using `onSaveInstanceState(outState: Bundle?)`, this is the method that the system calls after `onStart()` where you can retrieve the `Bundle` state instead of restoring the state using `onCreate(savedInstanceState: Bundle?)`.
- `override fun onResume():` This callback is run as the final stage of creating an Activity for the first time, and also when the app has been backgrounded and then is brought into the foreground. Upon the completion of this callback, the screen/activity is ready to be used, receive user events, and be responsive.
- `override fun onSaveInstanceState(outState: Bundle?):` If you want to save the state of the activity, this function can do so. You add key-value pairs using one of the convenience functions depending on the data type. The data will then be available if your Activity is recreated in `onCreate(savedInstanceState: Bundle?)` and `onRestoreInstanceState(savedInstanceState: Bundle?)`.
- `override fun onPause():` This function is called when the Activity starts to be backgrounded or another dialog or Activity comes into the foreground.
- `override fun onStop():` This function is called when the Activity is hidden, either because it is being backgrounded or another Activity is being launched on top of it.
- `override fun onDestroy():` This is called by the system to kill the Activity when system resources are low, when `finish()` is called explicitly on the Activity, or, more commonly, when the Activity is killed by the user closing the app from the recents/overview button.

The flow of callbacks/events is illustrated in the following diagram:

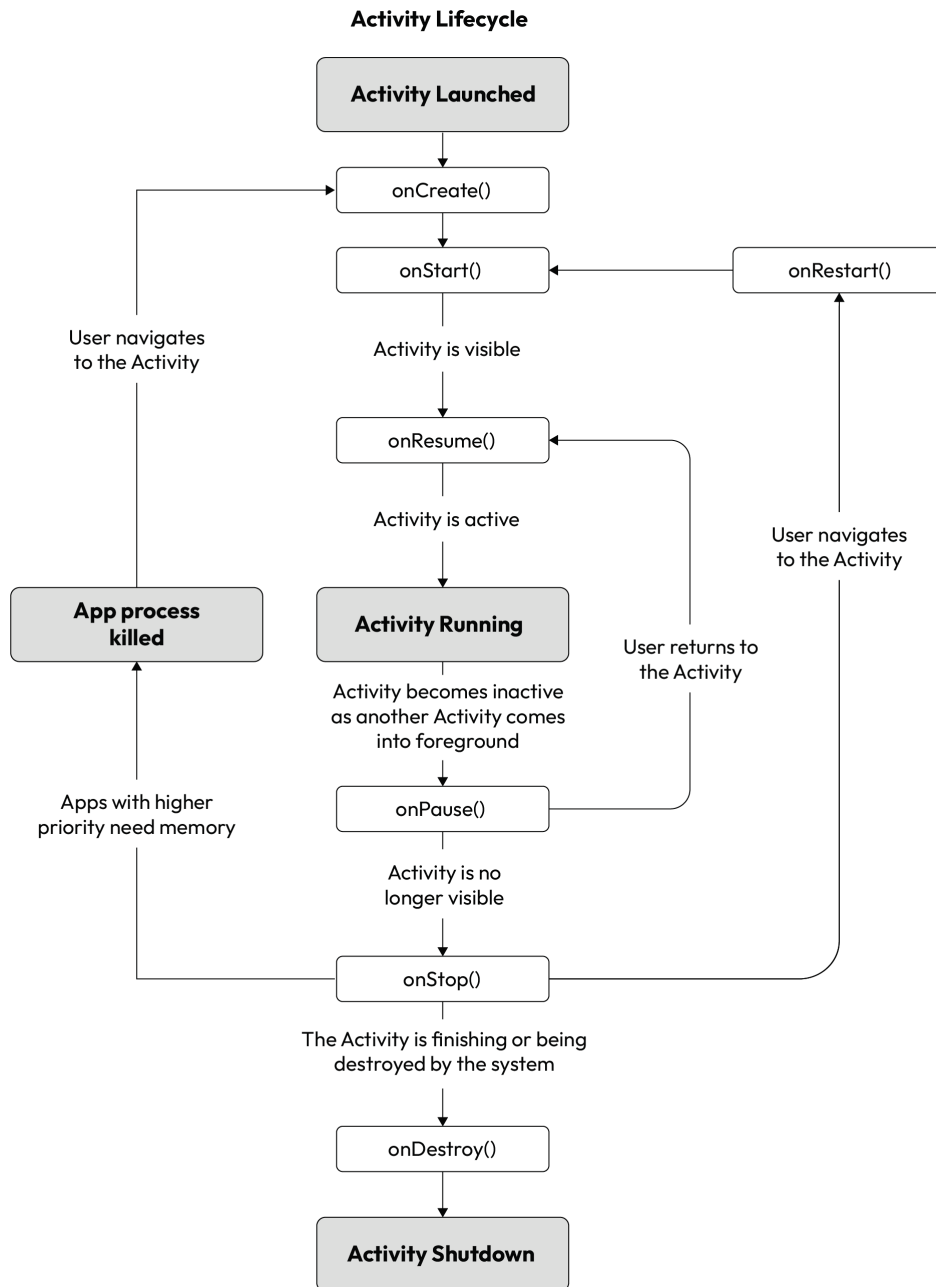


Figure 4.1 - Activity lifecycle

Now that you understand what these common lifecycle callbacks do, let's implement them to see when they are called.

Exercise 4.01 – logging the Activity Callbacks

Create an application called `Activity Callbacks` with an empty Activity. The aim of this exercise is to log the Activity callbacks and the order that they occur for common operations:

1. In order to verify the order of the callbacks, let's add a log statement at the end of each callback. Open up `MainActivity` and prepare the Activity for logging by adding `import android.util.Log` to the import statements. Then, add a constant to the class to identify your Activity. Constants in Kotlin are identified by the `const` keyword and can be declared at the top level (outside the class) or in an object within the class.

Top-level constants are generally used if they are required to be public. For private constants, Kotlin provides a convenient way to add static functionality to classes by declaring a companion object. Add the following at the bottom of the class below `onCreate(savedInstanceState: Bundle?)`:

```
companion object {  
    private const val TAG = "MainActivity"  
}
```

Then, add a log statement at the end of `onCreate(savedInstanceState: Bundle?)`:

```
Log.d(TAG, "onCreate")
```

Our Activity should now have the following code:

```
package com.example.activitycallbacks  
  
import android.os.Bundle  
import android.util.Log  
import androidx.appcompat.app.AppCompatActivity  
  
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        Log.d(TAG, "onCreate")  
    }  
}
```

```
companion object {
    private const val TAG = "MainActivity"
}
}
```

d in the preceding log statement refers to *debug*. There are six different log levels that can be used to output message information from the least to most important – v for *verbose*, d for *debug*, i for *info*, w for *warn*, e for *error*, and wtf for *what a terrible failure* (this last log level highlights an exception that should never occur):

```
Log.v(TAG, "verbose message")
Log.d(TAG, "debug message")
Log.i(TAG, "info message")
Log.w(TAG, "warning message")
Log.e(TAG, "error message")
Log.wtf(TAG, "what a terrible failure message")
```

- Now, let's see how the logs are displayed in Android Studio. Open the **Logcat** window. It can be accessed by clicking on the **Logcat** tab at the bottom of the screen and also from the toolbar by going to **View | Tool Windows | Logcat**.
- Run the app on the virtual device and examine the **Logcat** window output. You should see the log statement you have added formatted like the following line in *Figure 4.2*. If the **Logcat** window looks different, you might have to enable the newest version of **Logcat** by going to **Android Studio | Settings | Experimental** and checking the box that says **Enable New Logcat Tool Window**.

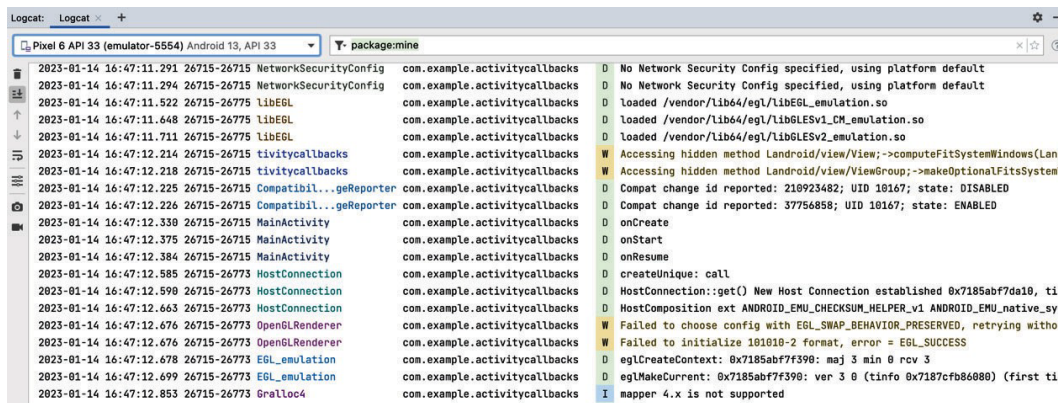


Figure 4.2 - Log output in Logcat

- Log statements can be quite difficult to interpret at first glance, so let's break down the following statement into its separate parts:

```
2025-04-24 16:47:12.330 26715-26715/com.dict311.
activitycallbacks/D/onCreate
```

Let's examine the elements of the log statement in detail:

Fields	Values
Date	2025-04-24
Time	16:47:12.330
Process identifier and thread identifier (your app process ID and current thread ID)	26715-26715
Class name	MainActivity
Package name	com.dict311.activitycallbacks
Log level	D (for Debug)
Log message	onCreate

Table explaining a log statement

By default, in the log filter (the text box above the log window), it says `package:mine`, which is your app logs. You can examine the output of the different log levels of all the processes on the device by changing the log filter from `level:debug` to other options in the drop-down menu. If you select `level:verbose`, as the name implies, you will see a lot of output.

- What's great about the `tag` option of the log statement is that it enables you to filter the log statements that are reported in the **Logcat** window of Android Studio by typing in `tag` followed by the text of the tag, `tag:MainActivity`, as shown in *Figure 4.4*:

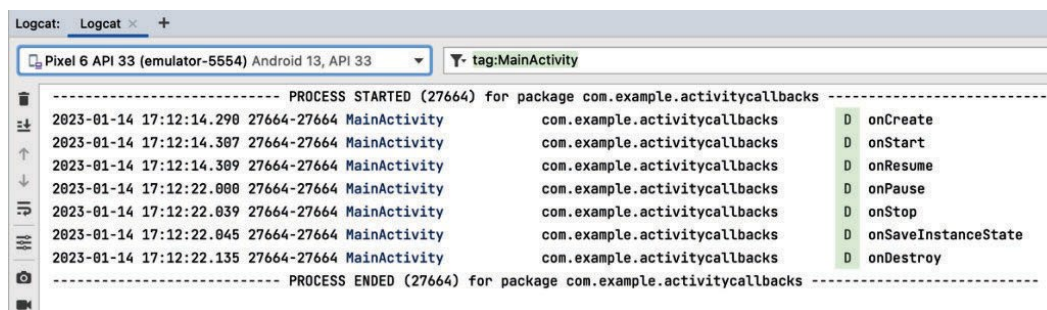


Figure 4.4 - Filtering log statements by the tag name

So, if you are debugging an issue in your Activity, you can type in the tag name and add logs to your Activity to see the sequence of log statements. This is what you are going to do next by implementing the principal Activity callbacks and adding a log statement to each one to see when they are run.

6. Place your cursor on a new line after the closing brace of the `onCreate(savedInstanceState: Bundle?)` function and then add the `onRestart()` callback with a log statement. Make sure you call through to `super.onRestart()` so that the existing functionality of the Activity callback works as expected:

```
override fun onRestart() {  
    super.onRestart()  
    Log.d(TAG, "onRestart")  
}
```

Note

In Android Studio, you can start typing the name of a function and autocomplete options will pop up with suggestions for functions to override. Alternatively, if you go to the top menu and then **Code** | **Generate** | **Override methods**, you can select the methods to override.

Do this for all of the following callback functions:

```
onCreate(savedInstanceState: Bundle?)  
onRestart()  
onStart()  
onRestoreInstanceState(savedInstanceState: Bundle)  
onResume()  
onPause()  
onStop()  
onSaveInstanceState(outState: Bundle)  
onDestroy()
```

7. The completed activity will now override the callbacks with your implementation, which adds a log message. The following truncated code snippet shows a log statement in `onCreate(savedInstanceState: Bundle?)`.

```
package com.example.activitycallbacks  
import android.os.Bundle  
import android.util.Log  
import androidx.appcompat.app.AppCompatActivity
```



```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        Log.d(TAG, "onCreate")  
    }  
    companion object {  
        private const val TAG = "MainActivity"  
    }  
}
```

8. Run the app, and then once it has loaded, as in *Figure 4.5*, look at the **Logcat** output; you should see the following log statements (this is a shortened version):

```
D/MainActivity: onCreate  
D/MainActivity: onStart  
D/MainActivity: onResume
```

The Activity has been created, started, and then prepared for the user to interact with:

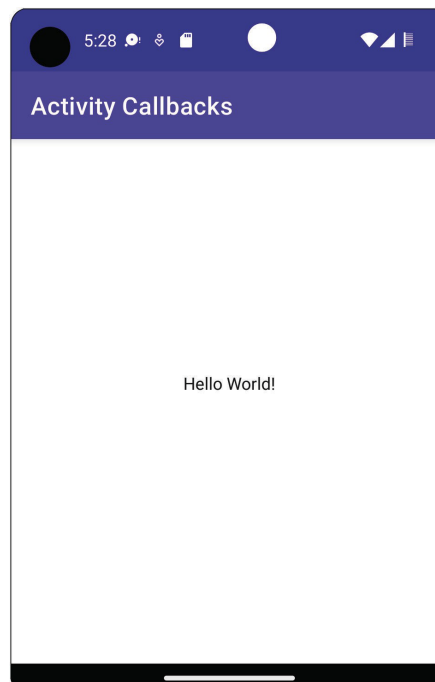


Figure 4.5 - The app loaded and displaying MainActivity

9. Press the round home button in the center of the navigation controls in the emulator window above the virtual device and background the app. Not all devices use the same three-button navigation of *back* (triangle icon), *home* (circle icon), and *recents/overview* (square icon). Gesture navigation can also be enabled so all these actions can be achieved by swiping and optionally holding. You should now see the following **Logcat** output:

```
D/MainActivity: onPause
D/MainActivity: onStop
D/MainActivity: onSaveInstanceState
```

For apps that target versions below Android Pie (API 28), `onSaveInstanceState (outState: Bundle)` may also be called before `onPause ()` or `onStop ()`.

10. Now, bring the app back into the foreground by pressing the recents/overview square button in the emulator controls and selecting the app. You should now see the following:

```
D/MainActivity: onRestart
D/MainActivity: onStart
D/MainActivity: onResume
```

The Activity has been restarted. You might have noticed that the `onRestoreInstanceState (savedInstanceState: Bundle)` function was not called. This is because the Activity was not destroyed and recreated.

11. Press the recents/overview square button again and then swipe the app image upward to kill the activity. This is the output:

```
D/MainActivity: onPause
D/MainActivity: onStop
D/MainActivity: onDestroy
```

12. Launch your app again and then rotate the phone. You might find that the phone does not rotate, and the display is sideways. If this happens, drag down the status bar at the very top of the virtual device, look for a button with a rectangular icon with arrows called **Auto-rotate**, and select it.

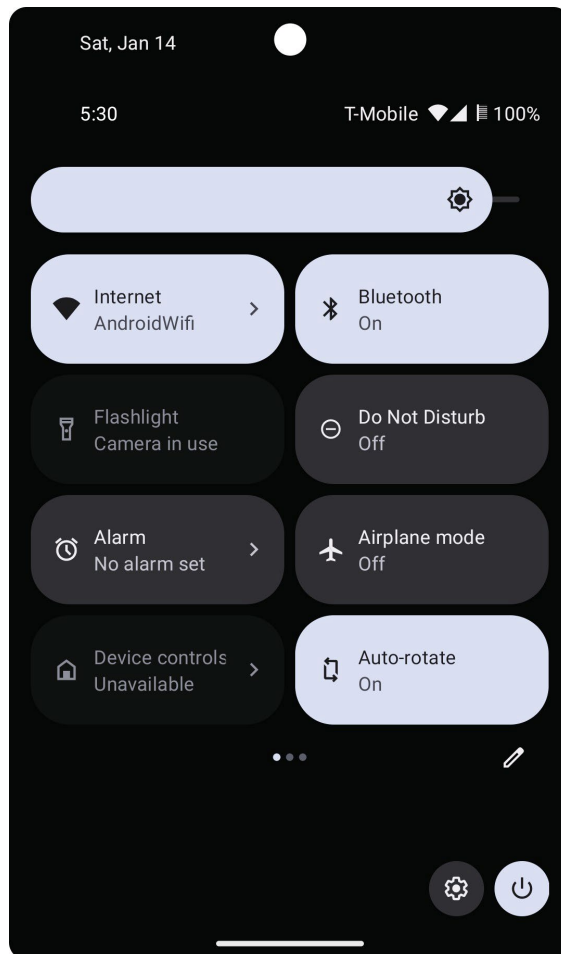


Figure 4.6 - Quick settings bar with Wi-Fi and Auto-rotate button selected

You should see the following callbacks:

```
D/MainActivity: onCreate
D/MainActivity: onStart
D/MainActivity: onResume
D/MainActivity: onPause
D/MainActivity: onStop
D/MainActivity: onSaveInstanceState
D/MainActivity: onDestroy
D/MainActivity: onCreate
D/MainActivity: onStart
```

```
D/MainActivity: onRestoreInstanceState
D/MainActivity: onResume
```

Please note that as stated in step 9, the order of the `onSaveInstanceState (outState: Bundle)` callback may vary.

Configuration changes, such as rotating the phone, by default recreate the activity. You can choose not to handle certain configuration changes in the app, which will then not recreate the activity.

13. To not recreate the activity for rotation, add `android:configChanges="orientation|screenSize|screenLayout"` to `MainActivity` in the `AndroidManifest.xml` file. Launch the app and then rotate the phone, and these are the only callbacks that you have added to `MainActivity` that you will see:

```
D/MainActivity: onCreate
D/MainActivity: onStart
D/MainActivity: onResume
```

The `orientation` and `screenSize` values have the same function for different Android API levels for detecting screen orientation changes. The `screenLayout` value detects other layout changes that might occur on foldable phones.

These are some of the config changes you can choose to handle yourself (another common one is `keyboardHidden` to react to changes in accessing the keyboard). The app will still be notified by the system of these changes through the following callback:

```
override fun onConfigurationChanged(newConfig:
Configuration) {
    super.onConfigurationChanged(newConfig)
    Log.d(TAG, "onConfigurationChanged")
}
```

If you add this callback function to `MainActivity`, and you have added `android:configChanges="orientation|screenSize|screenLayout"` to `MainActivity` in the manifest, you will see it called on rotation.

This approach of not restarting the activity is not recommended as the system will not apply alternative resources automatically. So, rotating a device from portrait to landscape won't apply a suitable landscape layout.

In this exercise, you have learned about the principal Activity callbacks and how they run when a user carries out common operations with your app through the system's interaction with `MainActivity`. In the next section, we will cover saving the state and restoring it, as well as see more examples of how the Activity lifecycle works.

Saving and restoring the Activity state

In this section, you'll explore how your Activity saves and restores the state. As you've learned in the previous section, configuration changes, such as rotating the phone, cause the Activity to be recreated. This can also happen if the system has to kill your app in order to free up memory.

In these scenarios, it is important to preserve the state of the Activity and then restore it. In the next two exercises, you'll work through an example ensuring that the user's data is restored when `TextView` is created and populated from a user's data after filling in a form.

Exercise 4.02 – saving and restoring the state in layouts

In this exercise, firstly create an application called `Save and Restore` with an empty activity. The app you are going to create will have a simple form that offers a discount code for a user's favorite restaurant if they enter some personal details (no actual information will be sent anywhere, so your data is safe):

1. Open up the `strings.xml` file (located in `app | src | main | res | values | strings.xml`) and add the following strings that you'll need for your app:

```
<string name="header_text">Enter your name and email for  
a discount code at Your Favorite Restaurant!</string>  
    <string name="first_name_label">First Name:</string>  
    <string name="email_label">Email:</string>  
    <string name="last_name_label">Last Name:</string>  
    <string name="discount_code_button">GET DISCOUNT  
    </string>  
    <string name="discount_code_confirmation">Hey %s!  
    Here is your discount code</string>  
    <string name="add_text_validation">Please fill in all  
    form fields</string>
```

2. In `R.layout.activity_main`, replace the contents with the following XML that creates a containing layout file and adds a `TextView` header with the text `Enter your name and email for a discount code at Your Favorite Restaurant!` This is done by adding the `android:text` attribute with the `@string/header_text` value:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.ConstraintLayout  
    xmlns:android=  
        "http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

```

xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:padding="4dp"
android:layout_marginTop="4dp"
tools:context=".MainActivity">
<TextView
    android:id="@+id/header_text"
    android:gravity="center"
    android:textSize="20sp"
    android:paddingStart="8dp"
    android:paddingEnd="8dp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/header_text"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"    />
</androidx.constraintlayout.widget.ConstraintLayout>

```

You will see here that `android:textSize` is specified in `sp` which stands for Scale-independent pixels. This unit type represents the same values as density-independent pixels, which define the size measurement according to the density of the device that your app is being run on, and also change the text size according to the user's preference, defined in **Settings | Display | Font style** (this might be **Font size and style** or something similar, depending on the exact device you are using).

Other attributes in the layout affect positioning. The most common ones are padding and margin. Padding is applied on the inside of Views and is the space between the text and the border. Margins are specified on the outside of Views and are the space from the outer edges of Views. For example, `android:padding` sets the padding for the View with the specified value on all sides. Alternatively, you can specify the padding for one of the four sides of a View with `android:paddingTop`, `android:paddingBottom`, `android:paddingStart`, and `android:paddingEnd`. This pattern also exists to specify margins, so `android:layout_margin` specifies the margin value for all four sides of a View, and `android:layout_marginTop`, `android:layout_marginBottom`, `android:layout_marginStart`, and `android:layout_marginEnd` allow setting the margin for individual sides.

In order to have consistency and uniformity throughout the app with these positioning values, you can define the margin and padding values as dimensions contained within a `dimens.xml` file so they can be used in multiple layouts. A dimension value of `<dimen name="grid_4">4dp</dimen>`

dimen> can then be used as a View attribute like this: `android:paddingStart="@dimen/grid_4"`. To position the content within a View, you can specify `android:gravity`. The center value constrains the content both vertically and horizontally within the View.

3. Next, add three `EditText` views below `header_text` for the user to add their first name, last name, and email:

```
<EditText
    android:id="@+id/first_name"
    android:textSize="20sp"
    android:layout_marginStart="24dp"
    android:layout_marginEnd="16dp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/first_name_label"
    android:inputType="text"
    app:layout_constraintTop_toBottomOf="@id/header_text"
    app:layout_constraintStart_toStartOf="parent"
/>

<EditText
    android:textSize="20sp"
    android:layout_marginEnd="24dp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/last_name_label"
    android:inputType="text"
    app:layout_constraintTop_toBottomOf="@id/header_text"
    app:layout_constraintStart_toEndOf="@id/first_name"
    app:layout_constraintEnd_toEndOf="parent" />
<!-- android:inputType="textEmailAddress" is not
enforced, but is a hint to the IME (Input Method
Editor) usually a keyboard to configure the
display for an email -typically by showing the '@'
symbol -->
<EditText
    android:id="@+id/email"
    android:textSize="20sp"
    android:layout_marginStart="24dp"
```

```

        android:layout_marginEnd="32dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/email_label"
        android:inputType="textEmailAddress"
        app:layout_constraintTop_toBottomOf="@id/first_name"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

```

The `EditText` fields have an `inputType` attribute to specify the type of input that can be entered into the form field. Some values, such as `number` on `EditText`, restrict the input that can be entered into the field, and on selecting the field, suggest how the keyboard is displayed. Others, such as `android:inputType="textEmailAddress"`, will not enforce an `@` symbol being added to the form field, but will give a hint to the keyboard to display it.

4. Finally, add a button for the user to press to generate a discount code, a `TextView` to display the discount code, and a `TextView` for the confirmation message:

```

<Button
    android:id="@+id/discount_button"
    android:textSize="20sp"
    android:layout_marginTop="12dp"
    android:gravity="center"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/discount_code_button"
    app:layout_constraintTop_toBottomOf="@id/email"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"/>
<TextView
    android:id="@+id/discount_code_confirmation"
    android:gravity="center"
    android:textSize="20sp"
    android:paddingStart="16dp"
    android:paddingEnd="16dp"
    android:layout_marginTop="8dp"
    android:layout_width="match_parent"

```



```

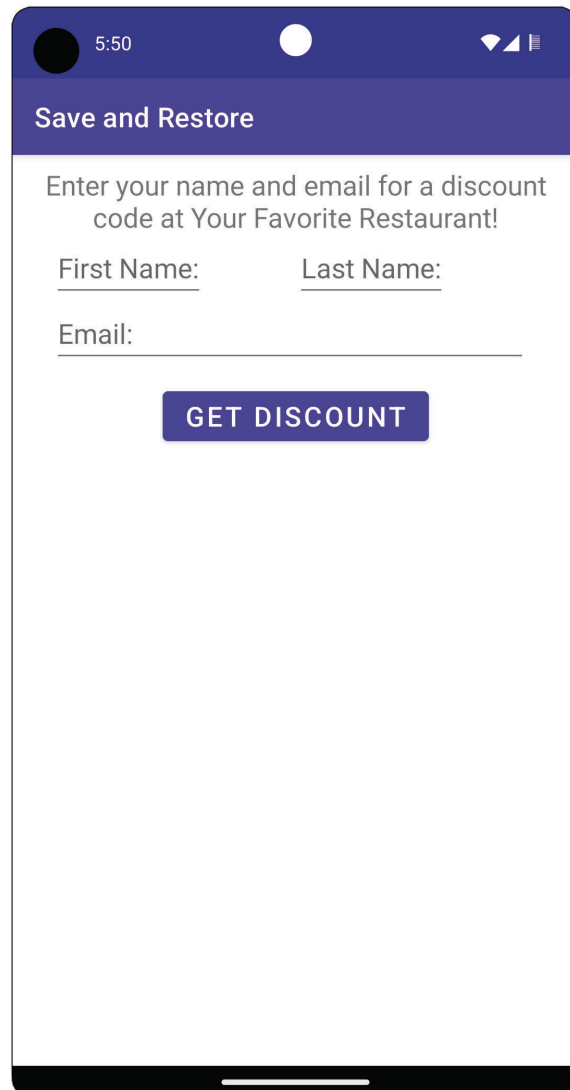
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/discount_
            button"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        tools:text="Hey John Smith! Here is your discount
            code" />
<TextView
    android:id="@+id/discount_code"
    android:gravity="center"
    android:textSize="20sp"
    android:textStyle="bold"
    android:layout_marginTop="8dp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toBottomOf="@id/discount_
        code_confirmation"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    tools:text="XHFG6H90" />

```

There are also some attributes that you haven't seen before. The `xmlns:tools="http://schemas.android.com/tools"` tools namespace, which was specified at the top of the XML layout file, enables certain features that can be used when creating your app to assist with configuration and design.

The attributes are removed when you build your app, so they don't contribute to the overall size of the app. You are using the `tools:text` attribute to show the text that will typically be displayed in the form fields. This helps when you switch to the **Design** view from viewing the XML in the **Code** view in Android Studio as you can see an approximation of how your layout displays on a device.

5. Run the app and you should see the output displayed in *Figure 4.7*. The **GET DISCOUNT** button has not been enabled and so currently will not do anything.



5:50

Save and Restore

Enter your name and email for a discount code at Your Favorite Restaurant!

First Name: Last Name:

Email:

GET DISCOUNT

Figure 4.7 - The Activity screen on the first launch

6. Enter some text into each of the form fields:

5:51

Save and Restore


Enter your name and email for a discount code at Your Favorite Restaurant!

Alex Forrester

some_email@some_domain.com

GET DISCOUNT

Figure 4.8 - The EditText fields filled in

- Now, use the second rotate button in the virtual device controls () to rotate the phone 90 degrees to the right:

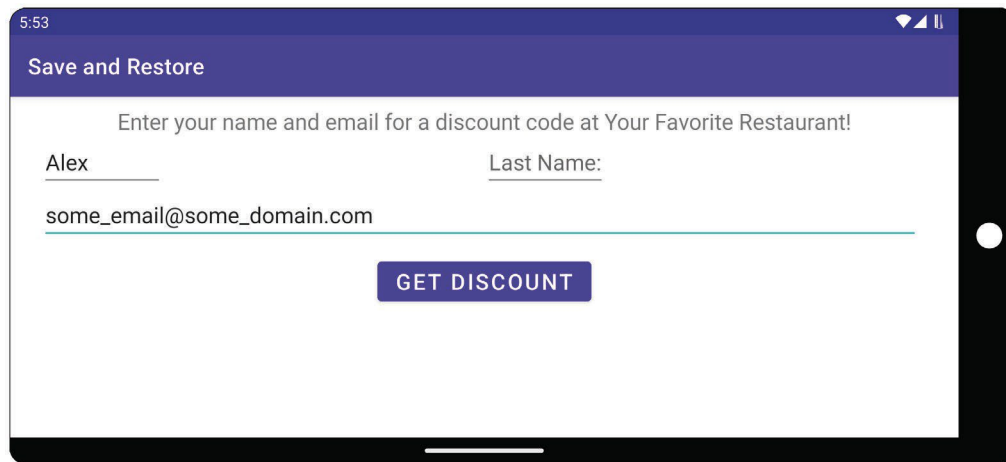


Figure 4.9 - The virtual device turned to landscape orientation

Can you spot what has happened? The **Last Name** field value is no longer set. It has been lost in the process of recreating the activity. Why is this? Well, in the case of the `EditText` fields, the Android framework will preserve the state of the fields if they have an ID set on them.

- Go back to the `activity_main.xml` layout file and add an ID to the Last Name `EditText` which appears below the First Name `EditText`:

```
<EditText
    android:id="@+id/first_name"

<EditText
    android:id="@+id/last_name"
...
```

When you run up the app again and rotate the device, it will preserve the value you have entered. You've now seen that you need to set an ID on the `EditText` fields to preserve the state. For the `EditText` fields, it's common to retain the state on a configuration change when the user is entering details into a form so that it is the default behavior if the field has an ID.

Obviously, you want to get the details of the `EditText` field once the user has entered some text, which is why you set an ID, but setting an ID for other field types, such as `TextView`, does not retain the state if you update them and you need to save the state yourself. Setting IDs for Views that enable scrolling, such as `RecyclerView`, is also important as it enables the scroll position to be maintained when the Activity is recreated.

Now, you have defined the layout for the screen, but you have not added any logic for creating and displaying the discount code. In the next exercise, we will work through this.

Exercise 4.03 – saving and restoring the state with Callbacks

The aim of this exercise is to bring all the UI elements in the layout together to generate a discount code after the user has entered their data. In order to do this, you will have to add logic to the button to retrieve all the `EditText` fields and then display a confirmation to the user, as well as generate a discount code:

1. Open up `MainActivity.kt` and replace the contents with the following:

```
package com.example.saveandrestore

import android.content.Context
import android.os.Bundle
import android.util.Log
import android.view.inputmethod.InputMethodManager
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import java.util.*

class MainActivity : AppCompatActivity() {

    private val discountButton: Button
        get() = findViewById(R.id.discount_button)
    private val firstName: EditText
        get() = findViewById(R.id.first_name)
    private val lastName: EditText
```

```

        get() = findViewById(R.id.last_name)
private val email: EditText
        get() = findViewById(R.id.email)
private val discountCodeConfirmation: TextView
        get() = findViewById(R.id.discount_code_
            confirmation)
private val discountCode: TextView
        get() = findViewById(R.id.discount_code)

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    Log.d(TAG, "onCreate")
    // here we handle the Button onClick event
    discountButton.setOnClickListener {
        val firstName = firstName.text.toString().
            trim()
        val lastName = lastName.text.toString().
            trim()
        val email = email.text.toString()
        if (firstName.isEmpty() || lastName.isEmpty()
            || email.isEmpty()) {
            Toast.makeText(this, getString(R.string.
                add_text_validation),
                Toast.LENGTH_LONG)
                .show()
        } else {
            val fullName = firstName.plus(" ")
                .plus(lastName)
            discountCodeConfirmation.text =
                getString(R.string.discount_code_
                    confirmation, fullName)
            // Generates discount code
            discountCode.text = UUID.randomUUID().
                toString().take(8).uppercase()
            hideKeyboard()
        }
    }
}

```

```

        }
    }
}
private fun hideKeyboard() {
    if (currentFocus != null) {
        val imm = getSystemService(Context.INPUT_
            METHOD_SERVICE) as InputMethodManager
        imm.hideSoftInputFromWindow(currentFocus?.
            windowToken, 0)
    }
}
companion object {
    private const val TAG = "MainActivity"
}
}

```

`get() = ...` is a custom accessor for a property.

Upon clicking the discount button, you retrieve the values from the `first_name` and `last_name` fields, concatenate them with a space, and then use a string resource to format the discount code confirmation text. The string you reference in the `strings.xml` file is as follows:

```

<string name="discount_code_confirmation">Hey %s! Here
is your discount code</string>

```

The `%s` value specifies a string value to be replaced when the string resource is retrieved. This is done by passing in the full name when getting the string:

```

getString(R.string.discount_code_confirmation,
    fullName)

```

The code is generated by using the **Universally Unique Identifier (UUID)** library from the `java.util` package. This creates a unique ID, and then the `take()` Kotlin function is used to get the first eight characters before setting these to uppercase. Finally, `discountCode` is set in the view, the keyboard is hidden, and all the form fields are set back to their initial values.

2. Run the app and enter some text into the name and email fields, and then click on GET DISCOUNT:

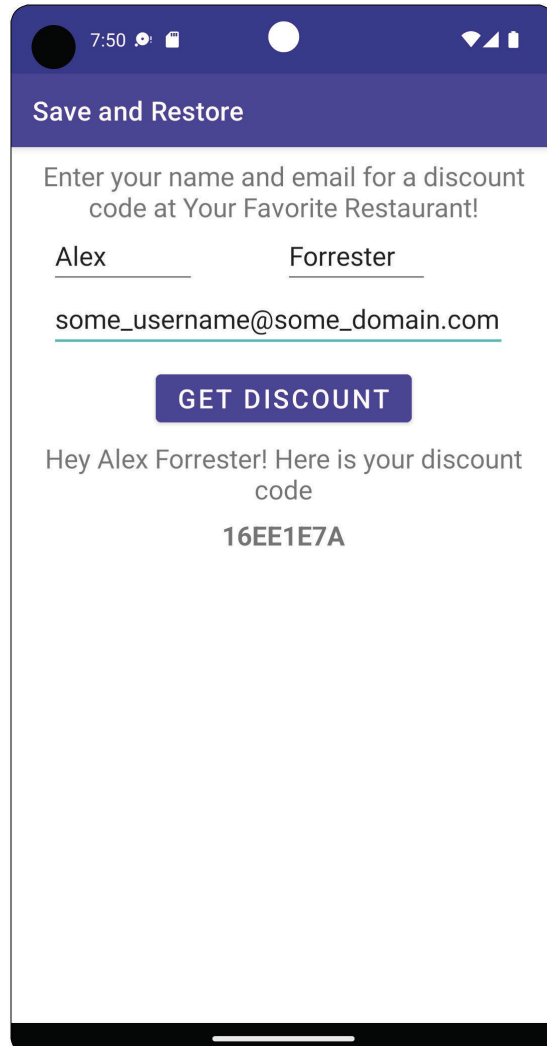


Figure 4.10 - Screen displayed after the user has generated a discount code

The app behaves as expected, showing the confirmation.

3. Now, rotate the phone by pressing the second rotate button () in the emulator controls and observe the result:

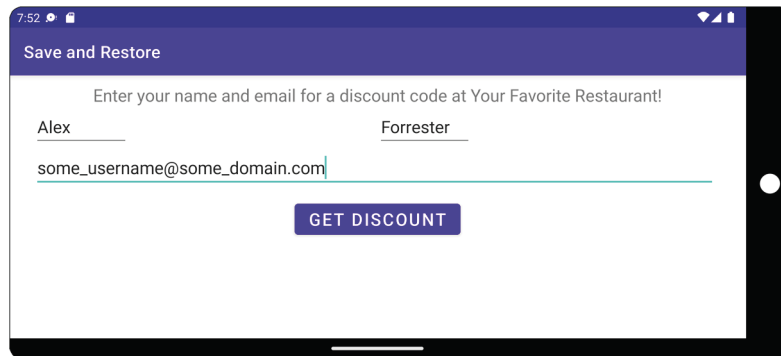


Figure 4.11 - Discount code no longer displaying on the screen

Oh, no! The discount code has gone. The `TextView` fields do not retain the state, so you will have to save the state yourself.

4. Go back into `MainActivity.kt` and add the following Activity callbacks:

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    Log.d(TAG, "onRestoreInstanceState")
}

override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.d(TAG, "onSaveInstanceState")
}
```

These callbacks, as the names declare, enable you to save and restore the instance state. `onSaveInstanceState(outState: Bundle)` allows you to add key-value pairs from your Activity when it is being backgrounded or destroyed, which you can retrieve in either `onCreate(savedInstanceState: Bundle?)` or `onRestoreInstanceState(savedInstanceState: Bundle)`.

So, you have two callbacks to retrieve the state once it has been set. If you are doing a lot of initialization in `onCreate(savedInstanceState: Bundle)`, it might be better to use `onRestoreInstanceState(savedInstanceState: Bundle)` to retrieve this instance state when your Activity is being recreated. In this way, it's clear which state is being recreated. However, you might prefer to use `onCreate(savedInstanceState: Bundle)` if there is minimal setup required.

Whichever of the two callbacks you decide to use, you will have to get the state you set in the `onSaveInstanceState(outState: Bundle)` call. For the next step in the exercise, you will use `onRestoreInstanceState(savedInstanceState: Bundle)`.

5. Add two constants to the MainActivity companion object, which is at the bottom of MainActivity:

```
private const val DISCOUNT_CONFIRMATION_MESSAGE =
    "DISCOUNT_CONFIRMATION_MESSAGE"
private const val DISCOUNT_CODE = "DISCOUNT_CODE"
```

6. Now, add these constants as keys for the values you want to save and retrieve and make the following changes to the Activity in the onSaveInstanceState(outState: Bundle) and onRestoreInstanceState(savedInstanceState: Bundle) functions.

```
override fun onRestoreInstanceState(savedInstanceState:
Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    Log.d(TAG, "onRestoreInstanceState")
    // Get the discount code or an empty string if it
    hasn't been set
    discountCode.text = savedInstanceState.
        getString(DISCOUNT_CODE, "")
    // Get the discount confirmation message or an empty
    string if it hasn't been set
    discountCodeConfirmation.text = savedInstanceState.
        getString(DISCOUNT_CONFIRMATION_MESSAGE, "")
}
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.d(TAG, "onSaveInstanceState")
    outState.putString(DISCOUNT_CODE, discountCode.text.
        toString())
    outState.putString(DISCOUNT_CONFIRMATION_MESSAGE,
        discountCodeConfirmation.text.toString())
}
```

7. Run the app, enter the values into the EditText fields, and then generate a discount code. Then, rotate the device and you will see that the discount code is restored in *Figure 4.12*:

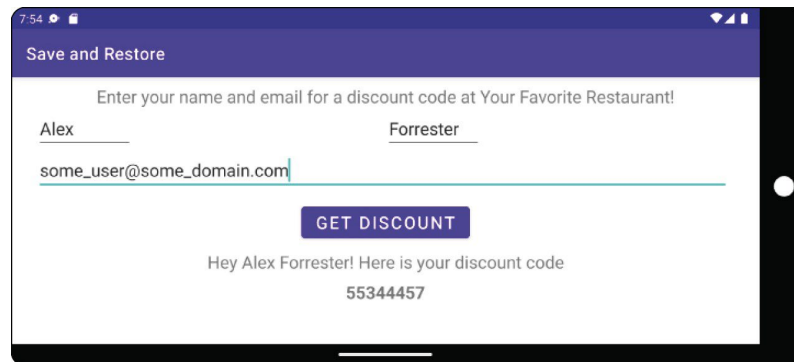


Figure 4.12 - Discount code continues to be displayed on the screen

In this exercise, you first saw how the state of the `EditText` fields is maintained on configuration changes. You also saved and restored the instance state using the Activity lifecycle `onSaveInstanceState(outState: Bundle)` and `onCreate(savedInstanceState: Bundle?)/onRestoreInstanceState(savedInstanceState: Bundle)` functions. These functions provide a way to save and restore simple data. The Android framework also provides `ViewModel`, an Android architecture component that is lifecycle-aware. The mechanisms of how to save and restore this state (with `ViewModel`) are managed by the framework, so you don't have to explicitly manage it as you have done in the preceding example.

In the next section, you will add another Activity to an app and navigate between the activities.

Activity interaction with Intents

An intent in Android is a communication mechanism between components. Within your own app, a lot of the time, you will want another specific Activity to start when some action happens in the current activity. Specifying exactly which Activity will start is called an **explicit intent**. On other occasions, you will want to get access to a system component, such as the camera. As you can't access these components directly, you will have to send an intent, which the system resolves in order to open the camera. These are called **implicit intents**. An intent filter has to be set up in order to register to respond to these events. Go to the `AndroidManifest.xml` file from the previous exercise and you will see an example of two intent filters set within the `<intent-filter>` XML element of the `MainActivity`:

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

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

The one specified with `<action android:name="android.intent.action.MAIN" />` means that this is the main entry point into the app. Depending on which category is set, it governs which Activity first starts when the app is started. The other intent filter that is specified is `<category android:name="android.intent.category.LAUNCHER" />`, which defines that the app should appear in the launcher. When combined, the two intent filters define that when the app is started from the launcher, `MainActivity` should be started. Removing the `<action android:name="android.intent.action.MAIN" />` intent filter results in the "Error running 'app': Default Activity not found" message. As the app has not got a main entry point, it can't be launched. If you remove `<category android:name="android.intent.category.LAUNCHER" />` then there is nowhere that it can be launched from.

For the next exercise, you will see how intents work to navigate around your app.

Exercise 4.04 – an introduction to Intents

The goal of this exercise is to create a simple app that uses intents to display text to the user based on their input:

1. Create a new project in Android Studio called `Intents Introduction` and select an empty Activity. Once you have set up the project, go to the toolbar and select `File | New | Activity | Empty Activity`. Call it `WelcomeActivity` and leave all the other defaults as they are. It will be added to the `AndroidManifest.xml` file, ready to use. The issue you have now that you've added `WelcomeActivity` is knowing how to do anything with it. `MainActivity` starts when you launch the app, but you need a way to launch `WelcomeActivity` and then, optionally, pass data to it, which is when you use intents.
2. In order to work through this example, add the following strings to the `strings.xml` file.

```
<string name="header_text">Please enter your name and
    then we'll get started!</string>
<string name="welcome_text">Hello %s, we hope you
    enjoy using the app!</string>
<string name="full_name_label">Enter your full
    name:</string>
<string name="submit_button_text">SUBMIT</string>
```

3. Next, change the MainActivity layout in activity_main.xml and replace the content with the following code to add an EditText and a Button:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:padding="28dp"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/full_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="28sp"
        android:hint="@string/full_name_label"
        android:layout_marginBottom="24dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>
    <Button
        android:id="@+id/submit_button"
        android:textSize="24sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/submit_button_text"
        app:layout_constraintTop_toBottomOf="@id/full_
            name"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

The app, when run, looks as in *Figure 4.13*:

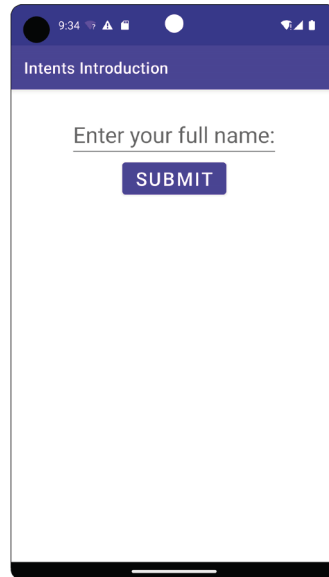


Figure 4.13 - The app display after adding the EditText full name field and SUBMIT button

You now need to configure the button so that when it's clicked, it retrieves the user's full name from the EditText field and then sends it in an intent, which starts WelcomeActivity.

4. Update the `activity_welcome.xml` layout file to prepare to do this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".WelcomeActivity">
    <TextView
        android:id="@+id/welcome_text"
        android:textSize="24sp"
        android:padding="24sp"
        android:layout_width="wrap_content"
```

```

        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        tools:text="Welcome John Smith we hope you enjoy
            using the app!"/>
    </androidx.constraintlayout.widget.ConstraintLayout>

```

You are adding a `TextView` field to display the full name of the user with a welcome message. The logic to create the full name and welcome message will be shown in the next step.

5. Now, open `MainActivity` and add a constant value above the class header and update the imports:

```

package com.example.intentsintroduction

import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
const val FULL_NAME_KEY = "FULL_NAME_KEY"

class MainActivity : AppCompatActivity() {
    ...
}

```

You will use the constant to set the key to hold the full name of the user by setting it in the intent.

6. Then, add the following code to the bottom of `onCreate` (`savedInstanceState: Bundle?`):

```

        findViewById<Button>(R.id.submit_button).
        setOnClickListener {
            val fullName = findViewById<EditText>(R.id.full_
                name).text.toString()

            if (fullName.isNotEmpty()) {
                // Set the name of the Activity to launch
            }
        }

```

```

        val welcomeIntent = Intent(this,
            WelcomeActivity::class.java)
        welcomeIntent.putExtra(FULL_NAME_KEY,
            fullName)
        startActivity(welcomeIntent)
    } else {
        Toast.makeText(this, getString(
            R.string.full_name_label),
            Toast.LENGTH_LONG).show()
    }
}

```

There is logic to retrieve the value of the full name and verify that the user has filled this in; otherwise, a pop-up toast message will be shown if it's blank. The main logic, however, takes the `fullName` value of the `EditText` field and creates an explicit intent to start `WelcomeActivity`, and then puts an `Extra` key with a value into the `Intent`. The last step is to use the intent to start `WelcomeActivity`.

7. Now, run the app, enter your name, and press **SUBMIT**, as shown in *Figure 4.14*:

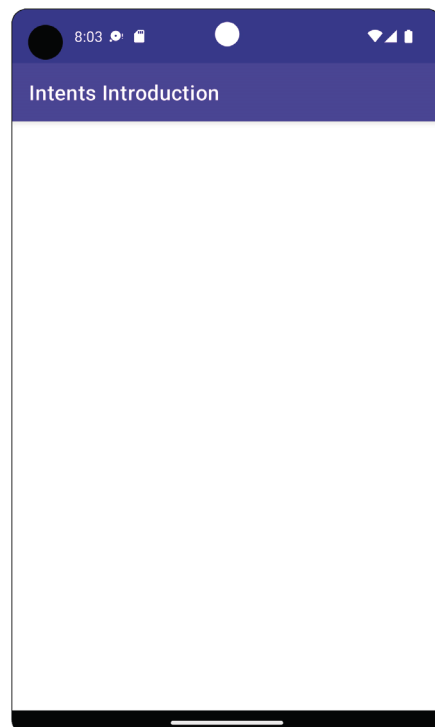


Figure 4.14 - The default screen displayed when the intent extras data is not processed

Well, that's not very impressive. You've added the logic to send the user's name, but not to display it.

8. To enable this, please open `WelcomeActivity` and add the import `import android.widget.TextView` to the imports list and add the following to the bottom of `onCreate(savedInstanceState: Bundle?)`:

```
if (intent != null) {  
    val fullName = intent.getStringExtra(FULL_NAME_KEY)  
    findViewById<TextView>(R.id.welcome_text).text =  
        getString(R.string.welcome_text, fullName)  
}
```

We check that the intent that started the Activity is not null and then retrieve the string value that was passed from the `MainActivity` intent by getting the string `FULL_NAME_KEY` extra key. We then format the `<string name="welcome_text">Hello %s, we hope you enjoy using the app!</string>` resource string by getting the string from the resources and passing in the `fullName` value retrieved from the intent. Finally, this is set as the text of `TextView`.

9. Run the app again, and a simple greeting will be displayed, as in *Figure 4.15*:

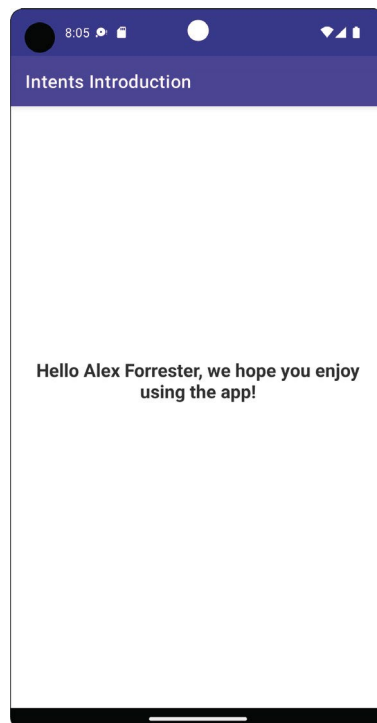


Figure 4.15 - User welcome message displayed

This exercise, although very simple in terms of layouts and user interaction, allows the demonstration of some core principles of intents. You will use them to add navigation and create user flows from one section of your app to another. In the next section, you will see how you can use intents to launch an Activity and receive a result back from it.

Exercise 4.05 – retrieving a result from an Activity

For some user flows, you will only launch an Activity for the sole purpose of retrieving a result back from it. This pattern is often used to ask permission to use a particular feature, popping up a dialog with a question about whether the user gives their permission to access contacts, the calendar, and so on, and then reporting the result of yes or no back to the calling Activity. In this exercise, you will ask the user to pick their favorite color of the rainbow, and then once that is chosen, display the result in the calling activity:

1. Create a new project named `Activity Results` with an empty activity and add the following strings to the `strings.xml` file:

```
<string name="header_text_main">Please click the button
below to choose your favorite color of the rainbow!</
string>
<string name="header_text_picker">Rainbow Colors</string>
<string name="footer_text_picker">Click the button above
which is your favorite color of the rainbow.</string>
<string name="color_chosen_message">%s is your favorite
color!</string>
<string name="submit_button_text">CHOOSE COLOR</string>
<string name="red">RED</string>
<string name="orange">ORANGE</string>
<string name="yellow">YELLOW</string>
<string name="green">GREEN</string>
<string name="blue">BLUE</string>
<string name="indigo">INDIGO</string>
<string name="violet">VIOLET</string>
<string name="unexpected_color">Unexpected color</string>
```

2. Add the following colors to `colors.xml`:

```
<!--Colors of the Rainbow -->
<color name="red">#FF0000</color>
<color name="orange">#FF7F00</color>
<color name="yellow">#FFFF00</color>
<color name="green">#00FF00</color>
```

```
<color name="blue">#0000FF</color>
<color name="indigo">#4B0082</color>
<color name="violet">#9400D3</color>
```

3. Now, you have to set up the Activity that will set the result you receive in MainActivity. Go to **File | New | Activity | EmptyActivity** and create an Activity called RainbowColorPickerActivity.
4. Update the activity_main.xml layout file to display a header, a button, and then a hidden android:visibility="gone" View, which will be made visible and set with the user's favorite color of the rainbow when the result is reported:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/header_text"
        android:textSize="20sp"
        android:padding="10dp"
        android:gravity="center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/header_text_main"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"/>
    <Button
        android:id="@+id/submit_button"
        android:textSize="18sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/submit_button_text"
```

```

        app:layout_constraintTop_toBottomOf="@id/header_
            text"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"/>
<TextView
    android:id="@+id/rainbow_color"
    android:layout_width="320dp"
    android:layout_height="50dp"
    android:layout_margin="12dp"
    android:textSize="22sp"
    android:textColor="@android:color/white"
    android:gravity="center"
    android:visibility="gone"
    tools:visibility="visible"
    app:layout_constraintTop_toBottomOf="@id/submit_
        button"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    tools:text="BLUE is your favorite color"
    tools:background="@color/blue"/>
</androidx.constraintlayout.widget.ConstraintLayout>

```

5. You'll be using the `registerForActivityResult` (`ActivityResultContracts.StartActivityForResult()`) function to get a result back from the Activity you launch. Add two constant keys for the values we want to use in the intent, as well as a default color constant above the class header in `MainActivity`, and update the imports so it is displayed as follows with the package name and imports:

```

package com.example.activityresults

import android.content.Intent
import android.graphics.Color
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import androidx.activity.result.contract.
ActivityResultContracts
import androidx.appcompat.app.AppCompatActivity

```

```
import androidx.core.content.ContextCompat
import androidx.core.view.isVisible

const val RAINBOW_COLOR_NAME = "RAINBOW_COLOR_NAME" //
Key to return rainbow color name in intent
const val RAINBOW_COLOR = "RAINBOW_COLOR" // Key to
return rainbow color in intent
const val DEFAULT_COLOR = "#FFFFFF" // White

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

6. Then, create a property below the class header that is used to both launch the new activity and return a result from it:

```
private val startForResult =
    registerForActivityResult(ActivityResultContracts.
        StartActivityForResult()) { activityResult ->
        val data = activityResult.data
        val backgroundColor = data?.getIntExtra(RAINBOW_
            COLOR, Color.parseColor(DEFAULT_COLOR))
            ?: Color.parseColor(DEFAULT_COLOR)
        val colorName = data?.getStringExtra(RAINBOW_
            COLOR_NAME) ?: ""
        val colorMessage = getString(R.string.color_
            chosen_message, colorName)

        val rainbowColor = findViewById<TextView>(R.
            id.rainbow_color)
        rainbowColor.setBackgroundColor(ContextCompat.
            getColor(this, backgroundColor))
        rainbowColor.text = colorMessage
        rainbowColor.isVisible = true
    }
```

Once the result is returned, you can proceed to query the intent data for the values you are expecting. For this exercise, we want to get the background color name (`colorName`) and the hexadecimal value of the color (`backgroundColor`) so that we can display it. The `?` operator checks whether the value is null (that is, not set in the intent), and if so, the Elvis operator (`?:`) sets the default value. The color message uses string formatting to set a message,

replacing the placeholder in the resource value with the color name. Now that you've got the colors, you can make the `rainbow_color` `TextView` field visible and set the background color of the View to `backgroundColor` and add text displaying the name of the user's favorite color of the rainbow.

7. First, add the logic to launch the Activity from the property defined previously by adding the following to the bottom of `onCreate (savedInstanceState: Bundle?)::`

```
findViewById<Button>(R.id.submit_button)
    .setOnClickListener {
        startForResult.launch(Intent(this,
            RainbowColorPickerActivity::class.java)
        )
    }
}
```

This creates an `Intent` that is launched for its result: `Intent (this, RainbowColorPickerActivity::class.java).`

8. For the layout of the `RainbowColorPickerActivity` activity, you are going to display a button with a background color and color name for each of the seven colors of the rainbow: RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, and VIOLET. These will be displayed in a `LinearLayout` vertical list. For most of the layout files in the modules, you will be using `ConstraintLayout`, which provides fine-grained positioning of individual Views. For situations where you need to display a vertical or horizontal list of a small number of items, `LinearLayout` is also a good choice. If you need to display a large number of items, then `RecyclerView` is a better option as it can cache layouts for individual rows and recycle views that are no longer displayed on the screen. You will learn about `RecyclerView` in *Week 5*.
9. The first thing you need to do in `RainbowColorPickerActivity` is create the layout. This will be where you present the user with the option to choose their favorite color of the rainbow.
10. Open `activity_rainbow_color_picker.xml` and replace the layout, inserting the following:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android=
    "http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
</ScrollView>
```

We are adding `ScrollView` to allow the contents to scroll if the screen height cannot display all of the items. `ScrollView` can only take one child View, which is the layout to scroll.

11. Next, add `LinearLayout` within `ScrollView` to display the contained views in the order that they are added with a header and a footer. The first child View is a header with the title of the page and the last View that is added is a footer with instructions for the user to pick their favorite color:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    tools:context=".RainbowColorPickerActivity">
    <TextView
        android:id="@+id/header_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:padding="10dp"
        android:text="@string/header_text_picker"
        android:textAllCaps="true"
        android:textSize="24sp"
        android:textStyle="bold"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <TextView
        android:layout_width="380dp"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="10dp"
        android:text="@string/footer_text_picker"
        android:textSize="20sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</LinearLayout>
```

The layout should now look as in *Figure 4.16* in the app:

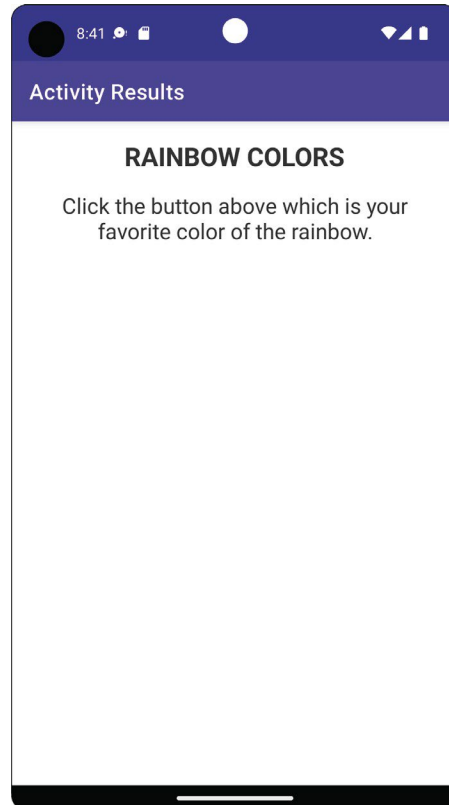


Figure 4.16 - Rainbow colors screen with a header and footer

12. Now, finally, add the button views between the header and the footer to select a colour of the rainbow, and then run the app (the following code only displays the first button).

```
<Button
    android:id="@+id/red_button"
    android:layout_width="120dp"
    android:layout_height="wrap_content"
    android:backgroundTint="@color/red"
    android:text="@string/red" />
```

These buttons are displayed in the order of the colors of the rainbow with the color text and background. The XML `id` attribute is what you will use in the Activity to prepare the result of what is returned to the calling activity.

13. Now, open `RainbowColorPickerActivity` and replace the content with the following:

```
package com.example.activityresults

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

class RainbowColorPickerActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_rainbow_color_picker)
    }
    private fun setRainbowColor(colorName: String, color: Int) {
        Intent().let { pickedColorIntent ->
            pickedColorIntent.putExtra(RAINBOW_COLOR_NAME, colorName)
            pickedColorIntent.putExtra(RAINBOW_COLOR, color)
            setResult(Activity.RESULT_OK, pickedColorIntent)
            finish()
        }
    }
}
```

The `setRainbowColor` function creates an intent and adds the rainbow color name and the rainbow color hex value as String extras. The result is then returned to the calling Activity, and as you have no further use for this Activity, you call `finish()` so that the calling Activity is displayed. The way that you retrieve the rainbow color that the user has chosen is by adding a listener for all the buttons in the layout.

14. Now, add the following to the bottom of onCreate (savedInstanceState: Bundle?):

```
val colorPickerClickListener = View.OnClickListener {
    view ->
        when (view.id) {
            R.id.red_button -> setRainbowColor(getString(R.string.red), R.color.red)
            R.id.orange_button -> setRainbowColor(getString(R.string.orange), R.color.orange)
            R.id.yellow_button -> setRainbowColor(getString(R.string.yellow), R.color.yellow)
            R.id.green_button -> setRainbowColor(getString(R.string.green), R.color.green)
            R.id.blue_button -> setRainbowColor(getString(R.string.blue), R.color.blue)
            R.id.indigo_button -> setRainbowColor(getString(R.string.indigo), R.color.indigo)
            R.id.violet_button -> setRainbowColor(getString(R.string.violet), R.color.violet)
            else -> {
                Toast.makeText(this, getString(R.string.unexpected_color), Toast.LENGTH_LONG)
                    .show()
            }
        }
}
```

The `colorPickerClickListener` added in the preceding code determines which colors to set for the `setRainbowColor(colorName: String, color: Int)` function by using a `when` statement. The `when` statement is the equivalent of the `switch` statement in Java and languages based on C. It allows multiple conditions to be satisfied with one branch and is more concise. In the preceding example, `view.id` is matched against the IDs of the rainbow layout buttons and, when found, executes the branch, setting the color name and hex value from the string resources to pass into `setRainbowColor(colorName: String, color: Int)`.

15. Now, add this click listener to the buttons from the layout below the preceding code:

```
findViewById<View>(R.id.red_button).
setOnClickListener(colorPickerClickListener)
```

```
findViewById<View>(R.id.orange_button) .  
setOnClickListener(colorPickerClickListener)  
findViewById<View>(R.id.yellow_button) .  
setOnClickListener(colorPickerClickListener)  
findViewById<View>(R.id.green_button) .  
setOnClickListener(colorPickerClickListener)  
findViewById<View>(R.id.blue_button) .  
setOnClickListener(colorPickerClickListener)  
findViewById<View>(R.id.indigo_button) .  
setOnClickListener(colorPickerClickListener)  
findViewById<View>(R.id.violet_button) .  
setOnClickListener(colorPickerClickListener)
```

Every button has a `ClickListener` interface attached, and as the operation is the same, they have the same `ClickListener` interface attached. Then, when the button is pressed, it sets the result of the color that the user has chosen and returns it to the calling activity.

16. Now, run the app and press the `CHOOSE COLOR` button, as shown in *Figure 4.17*:

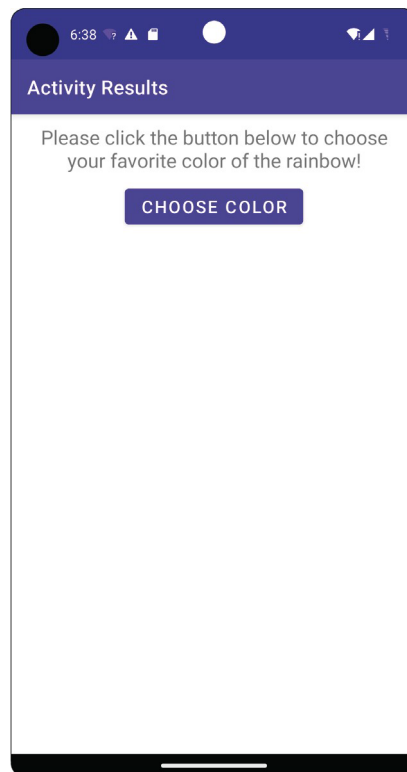


Figure 4.17 - The rainbow colors app start screen

17. Now, select your favorite color of the rainbow:

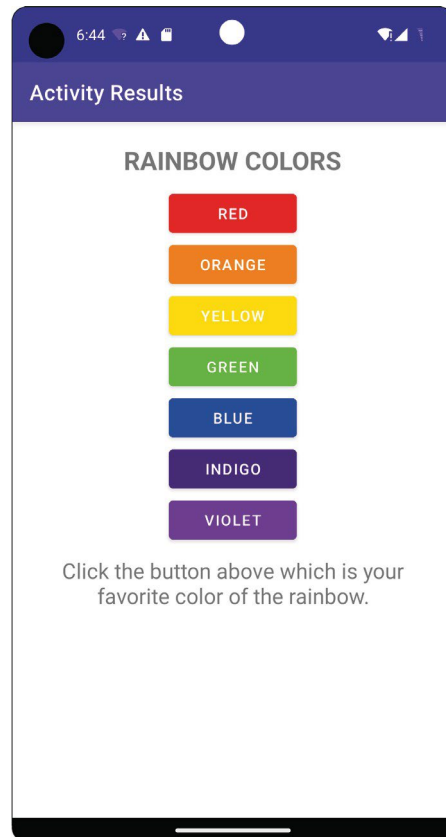


Figure 4.18 - The rainbow colors selection screen

18. Once you've chosen your favorite color, a screen with your favorite color will be displayed, as shown in *Figure 4.19*:

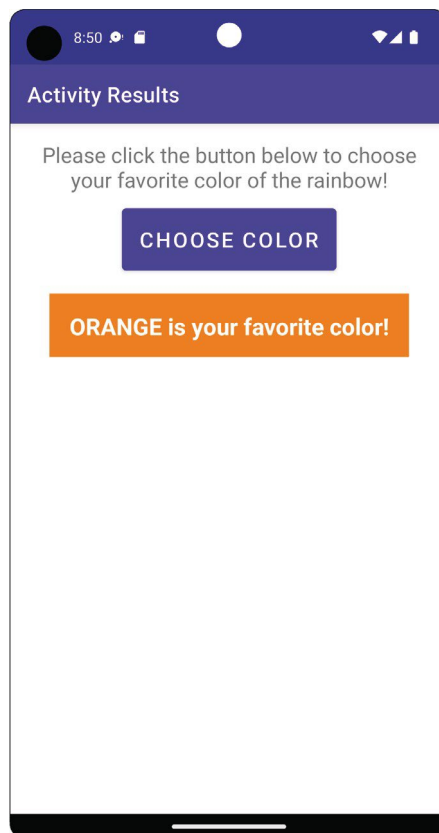


Figure 4.19 - The app displaying the selected color

As you can see, the app displays the color that you've selected as your favorite color in *Figure 4.19*.

This exercise introduced you to another way of creating user flows using `registerForActivityResult`. This can be very useful for carrying out a dedicated Task where you need a result before proceeding with the user's flow through the app. Next, you will explore launch modes and how they impact the flow of user journeys when building apps.

Intents, Tasks, and Launch Modes

Up until now, you have been using the standard behavior for creating Activities and moving from one Activity to the next. When you open the app from the launcher with the default behavior, it creates its own Task, and each Activity you create is added to a back stack, so when you open three Activities one after the other as part of your user's journey, pressing the back button three times will move the user back through the previous screens/Activities and then go back to the device's home screen, while still keeping the app open.

The launch mode for this type of Activity is called `Standard`; it is the default and doesn't need specifying in the Activity element of `AndroidManifest.xml`. Even if you launch the same Activity three times, one after the other, there will be three instances of the same activity that exhibit the behavior described previously.

For some apps, you may want to change this behavior so the same instance is used. The launch mode that can help here is called `singleTop`. If a `singleTop` Activity is the most recently added, when the same `singleTop` Activity is launched again, then instead of creating a new Activity, it uses the same Activity and runs the `onNewIntent` callback. In this callback, you receive an intent, and you can then process this intent as you have done previously in `onCreate`.

There are three other launch modes to be aware of called `singleTask`, `singleInstance` and `singleInstancePerTask`. These are not for general use and are only used for special scenarios. Detailed documentation of all launch modes can be viewed here: <https://developer.android.com/guide/topics/manifest/activity-element#lmode>.

You'll explore the differences in behavior of the `Standard` and `singleTop` launch modes in the next exercise.

Exercise 4.06 – setting the Launch Mode of an Activity

This exercise has many different layout files and Activities to illustrate the two most commonly used launch modes.

1. Open up the `activity_main.xml` file and examine it.

This illustrates a new concept when using layout files. If you have a layout file and you would like to include it in another layout, you can use the `<include>` XML element (have a look at the following snippet of the layout file):

```
<include layout="@layout/letters"
    android:id="@+id/letters_layout"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/launch_mode_
        standard"/>
<include layout="@layout/numbers"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```

app:layout_constraintLeft_toLeftOf="parent"
app:layout_constraintRight_toRightOf="parent"
app:layout_constraintTop_toBottomOf="@id/launch_mode_
single_top"/>

```

The preceding layout uses the `include` XML element to include the two layout files: `letters.xml` and `numbers.xml`.

2. Open up and inspect the `letters.xml` and `numbers.xml` files found in the `res | layout` folder. These are very similar and are only differentiated from the buttons they contain by the ID of the buttons themselves and the text label they display.
3. Run the app and you will see the following screen:

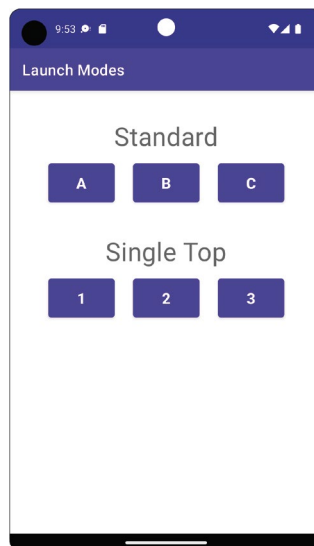


Figure 4.20 - App displaying both the standard and single top modes

In order to demonstrate/illustrate the difference between the `standard` and `singleTop` activity launch modes, you have to launch two or three activities one after the other.

4. Open up `MainActivity` and examine the contents of the code block (truncated) in `onCreate(savedInstanceState: Bundle?)`:

```

val buttonClickListener = View.OnClickListener { view ->
    when (view.id) {
        R.id.letterA -> startActivity(Intent(this,
            ActivityA::class.java))
    }
}

```

```

        // Other letters and numbers follow the same
        pattern/flow
    else -> {
        Toast.makeText(
            this, getString(
                R.string.unexpected_button_pressed
            ),
            Toast.LENGTH_LONG
        )
        .show()
    }
}

findViewById<View>(R.id.letterA).
setOnClickListener(buttonClickListener)
// The buttonClickListener is set on all the number and
letter views

```

The logic contained in the main Activity and the other activities is basically the same. It displays an Activity and allows the user to press a button to launch another Activity using the same logic of creating a `ClickListener` and setting it on the button you saw in *Exercise 4.05, Retrieving a result from an Activity*.

5. Open the `AndroidManifest.xml` file and you will see the following activities displayed:

```

<activity android:name=".ActivityA"
    android:launchMode="standard"/>
<activity android:name=".ActivityB"
    android:launchMode="standard"/>
<activity android:name=".ActivityC"
    android:launchMode="standard"/>
<activity android:name=".ActivityOne"
    android:launchMode="singleTop"/>
<activity android:name=".ActivityTwo"
    android:launchMode="singleTop"/>
<activity android:name=".ActivityThree"
    android:launchMode="singleTop"/>

```

You launch an Activity based on a button pressed on the main screen, but the letter and number activities have a different launch mode, which you can see specified in the `AndroidManifest.xml` file.

The standard launch mode is specified here to illustrate the difference between standard and singleTop, but standard is the default and would be how the Activity is launched if the android:launchMode XML attribute was not present.

6. Press one of the letters under the Standard heading and you will see the following screen (with **A**, **B**, or **C**):

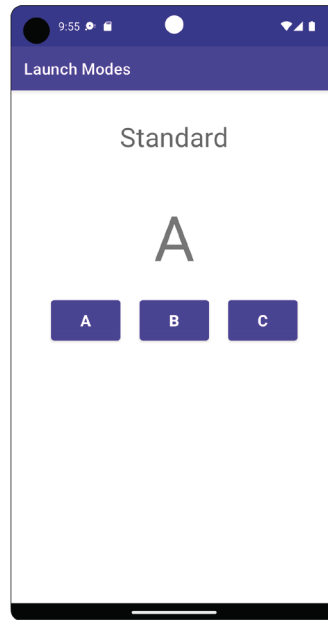


Figure 4.21 - The app displaying standard activity

7. Keep on pressing any of the letter buttons, which will launch another Activity. Logs have been added to show this sequence of launching activities. Here is the log after pressing 10 letter Activities randomly:

```
MainActivity com.example.launchmodes onCreate
Activity A   com.example.launchmodes onCreate
Activity A   com.example.launchmodes onCreate
Activity B   com.example.launchmodes onCreate
Activity B   com.example.launchmodes onCreate
Activity C   com.example.launchmodes onCreate
Activity B   com.example.launchmodes onCreate
Activity B   com.example.launchmodes onCreate
Activity A   com.example.launchmodes onCreate
Activity C   com.example.launchmodes onCreate
```

```
Activity B    com.example.launchmodes onCreate
Activity C    com.example.launchmodes onCreate
```

If you observe the preceding log, every time the user presses a character button in launch mode, a new instance of the character Activity is launched and added to the back stack.

8. Close the app, making sure it is not backgrounded (or in the recents/overview menu) but is actually closed, and then open the app again and press one of the number buttons under the **Single Top** heading:

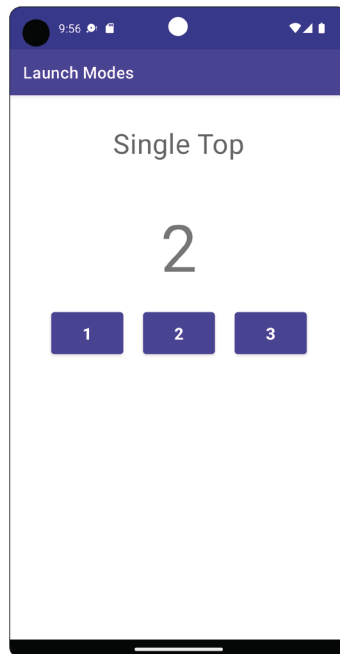


Figure 4.22 - The app displaying the Single Top activity

9. Press the number buttons 10 times, but make sure you press the same number button at least twice sequentially before pressing another number button.

The logs you should see in the **Logcat** window (**View** | **Tool Windows** | **Logcat**) should be similar to the following:

```
MainActivity com.example.launchmodes onCreate
Activity 1    com.example.launchmodes onCreate
Activity 1    com.example.launchmodes onNewIntent
Activity 2    com.example.launchmodes onCreate
Activity 2    com.example.launchmodes onNewIntent
```

Activity 3	com.example.launchmodes	onCreate
Activity 2	com.example.launchmodes	onCreate
Activity 3	com.example.launchmodes	onCreate
Activity 3	com.example.launchmodes	onNewIntent
Activity 1	com.example.launchmodes	onCreate
Activity 1	com.example.launchmodes	onNewIntent

You'll notice that instead of calling `onCreate` when you pressed the same button again at least twice sequentially, the Activity is not created, but a call is made to `onNewIntent`. If you press the back button, you'll notice that it will take you less than 10 clicks to back out of the app and return to the home screen, reflecting the fact that 10 activities have not been created.

Activity 4.01 – creating a login form

The aim of this activity is to create a login form with username and password fields. Once the values in these fields are submitted, check these entered values against the hardcoded values and display a welcome message if they match, or an error message if they don't, and return the user to the login form. The steps needed to achieve this are the following:

1. Create a form with username and password `EditText` Views and a `LOGIN` button.
2. Add a `ClickListener` interface to the button to react to a button press event.
3. Validate that the form fields are filled in.
4. Check the submitted username and password fields against the hardcoded values.
5. Display a welcome message with the username if successful and hide the form.
6. Display an error message if not successful and redirect the user back to the form.

There are a few possible ways that you could go about trying to complete this activity. Here are three ideas for approaches you could adopt:

- Use a `singleTop` Activity and send an intent to route to the same Activity to validate the credentials
- Use a standard Activity to pass a username and password to another Activity and validate the credentials
- Use `registerForActivityResult` to carry out the validation in another Activity and then return the result

The completed app, upon its first loading, should look as in *Figure 4.23*:

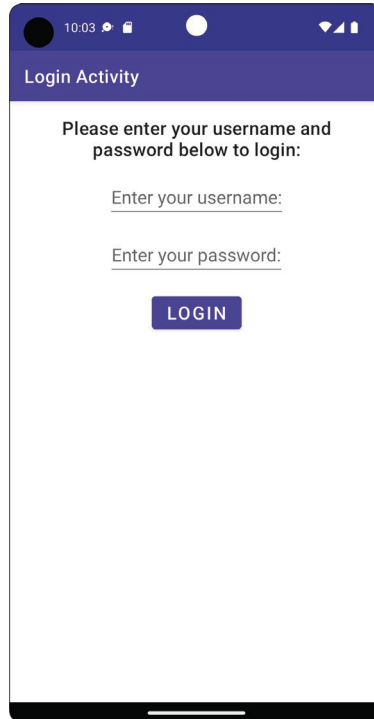


Figure 4.23 - The app display when first loaded

Summary

In this week, you have covered a lot of the groundwork of how your application interacts with the Android framework, from the Activity lifecycle callbacks to retaining the state in your activities, navigating from one screen to another, and how intents and launch modes make this happen. These are core concepts that you need to understand in order to move on to more advanced topics.

This is part one of this week. In Part2, you will be introduced to fragments and how they fit into the architecture of your application, as well as exploring more of the Android resources framework.