Android Studio Tutorial

Introduction

In this tutorial, you will learn how to create and run your own basic Android application. You will be using Java as the programming language, and assumes you know the very

basics of Java. You'll learn:

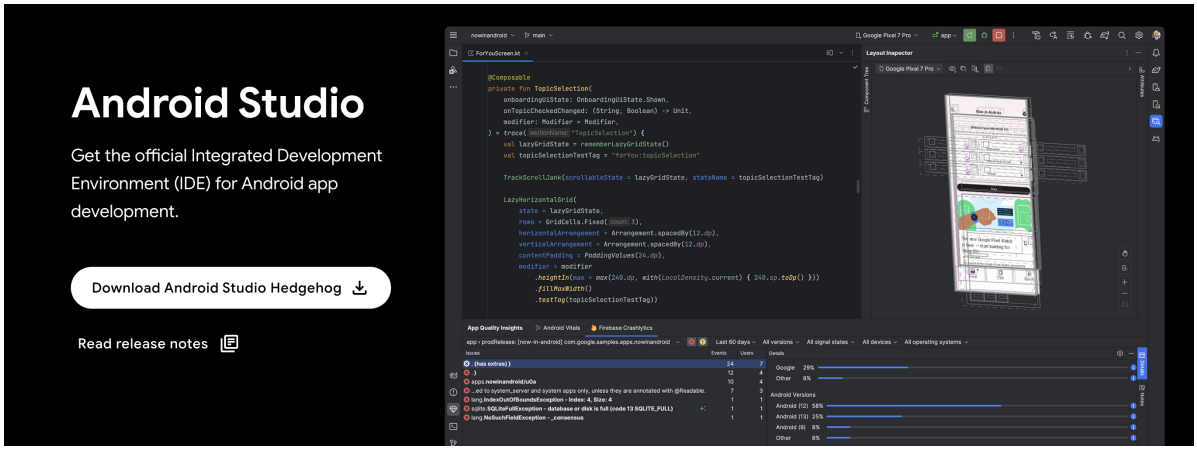
- How to use Android Studio
- How to run your new app on an emulator
- How to add interactive buttons
- How to pass information between fragments

Installing Android Studio

To complete this tutorial, you will need Android Studio Hedgehog.

To install Android Studio:

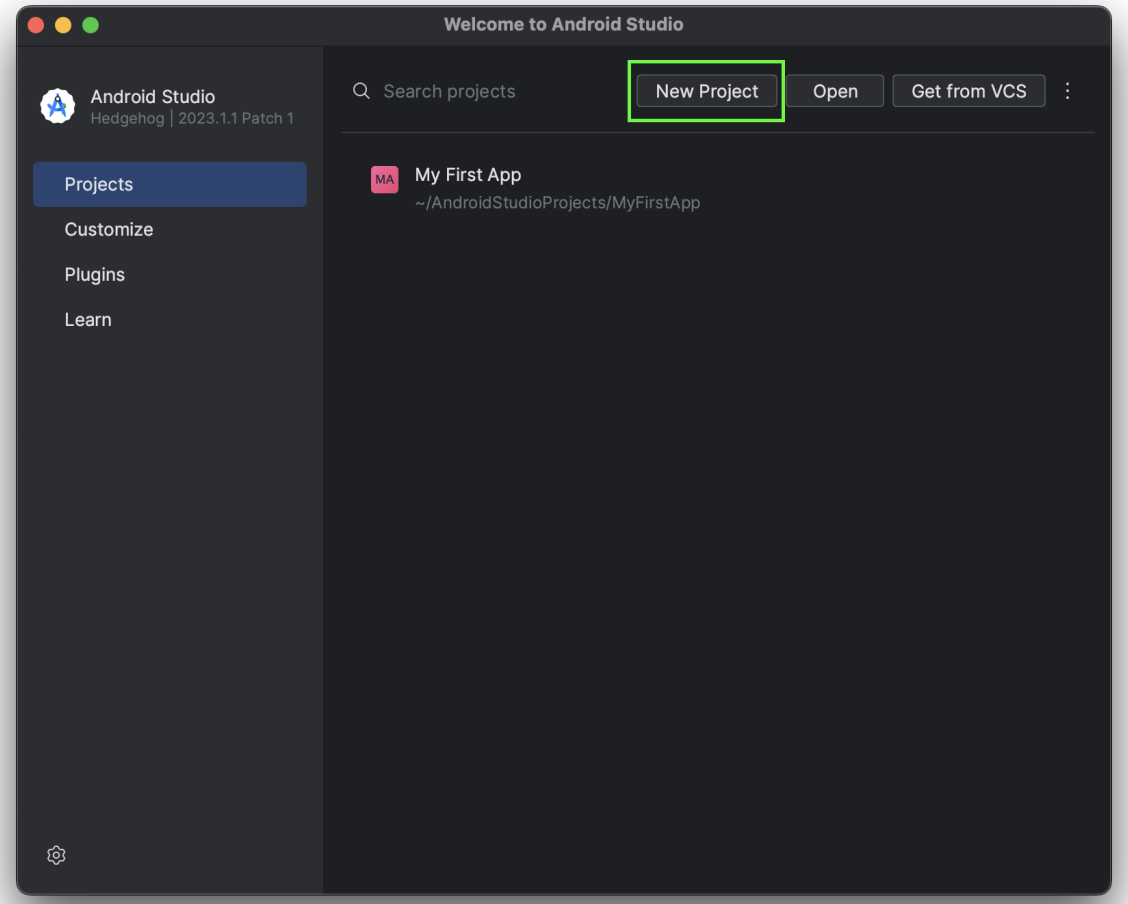
1. Navigate to the download page and click "**Download Android Studio Hedgehog**".
2. Accept the terms and conditions and download the version appropriate for your OS/Chip.



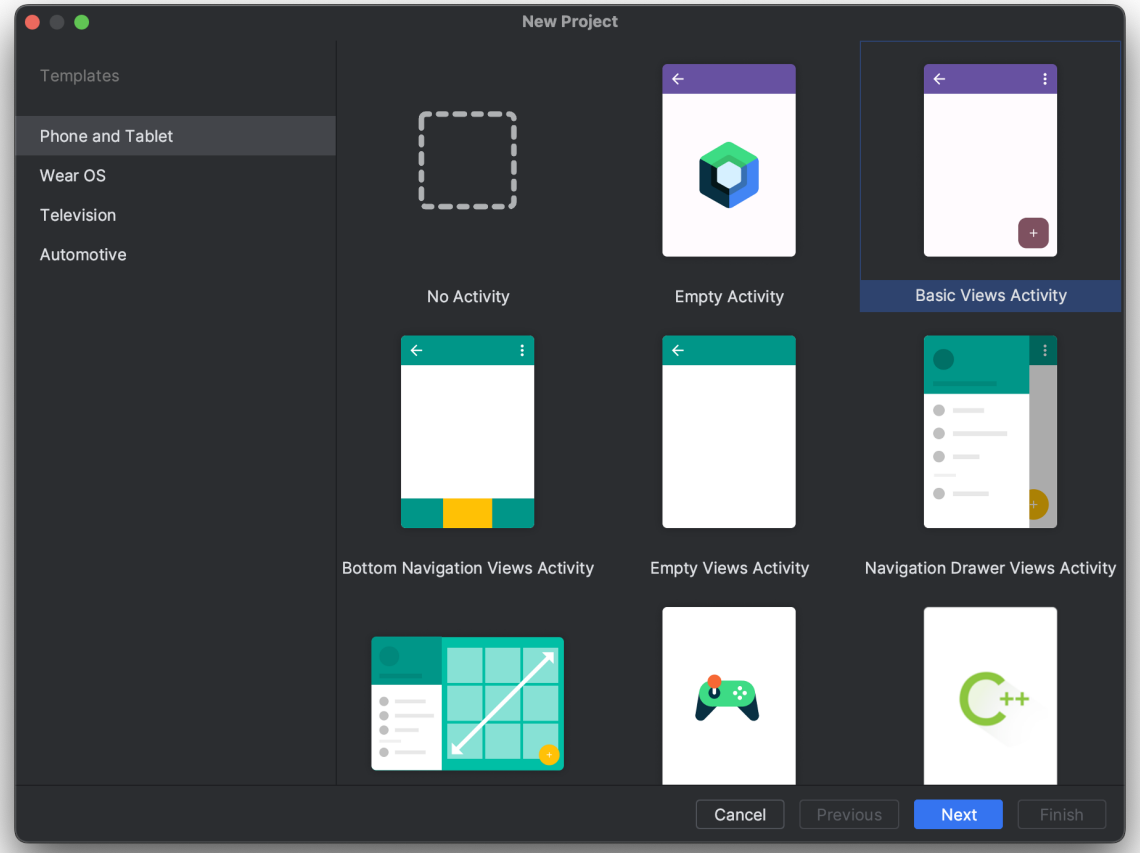
3. Once it has download, follow the instructions with the setup wizard to install components and Android SDK (If you need more help with installation, see here).
4. After installation, open **Android Studio**. You are ready to start building your app!

Creating your first project in Android Studio

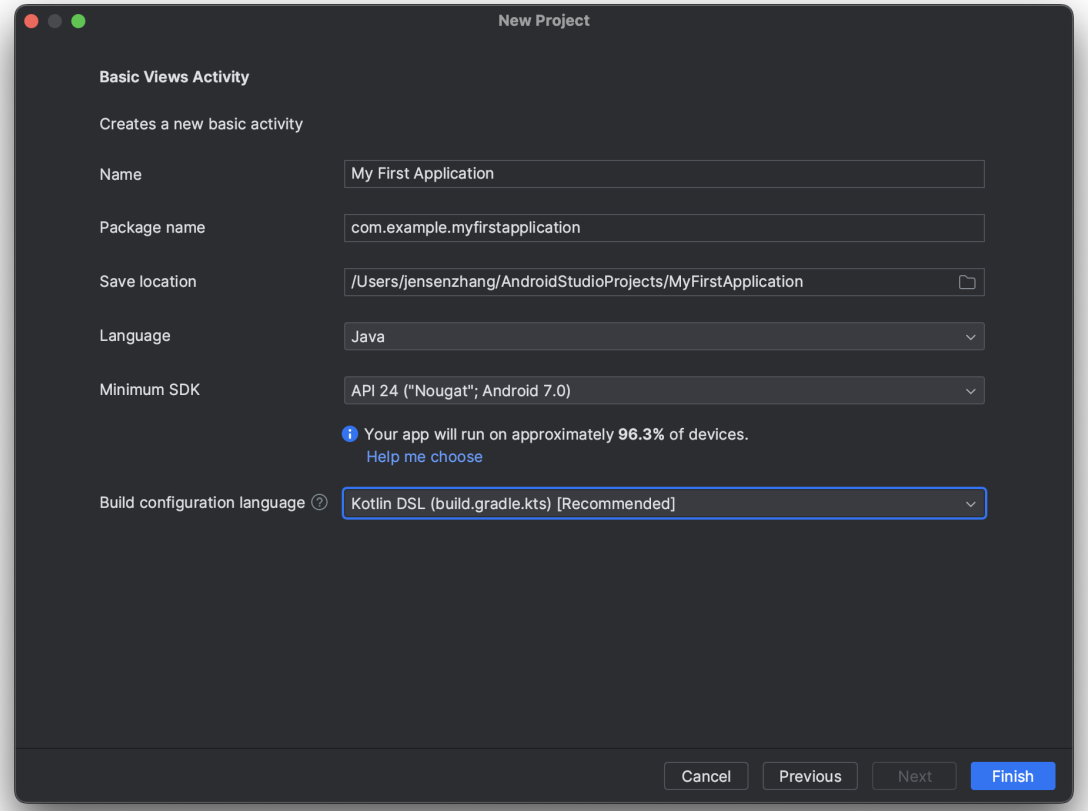
1. Open the Android Studio app
2. In the *Welcome to Android Studio* window that appears, click **New Project**



3. Select **Basic Views Activity**



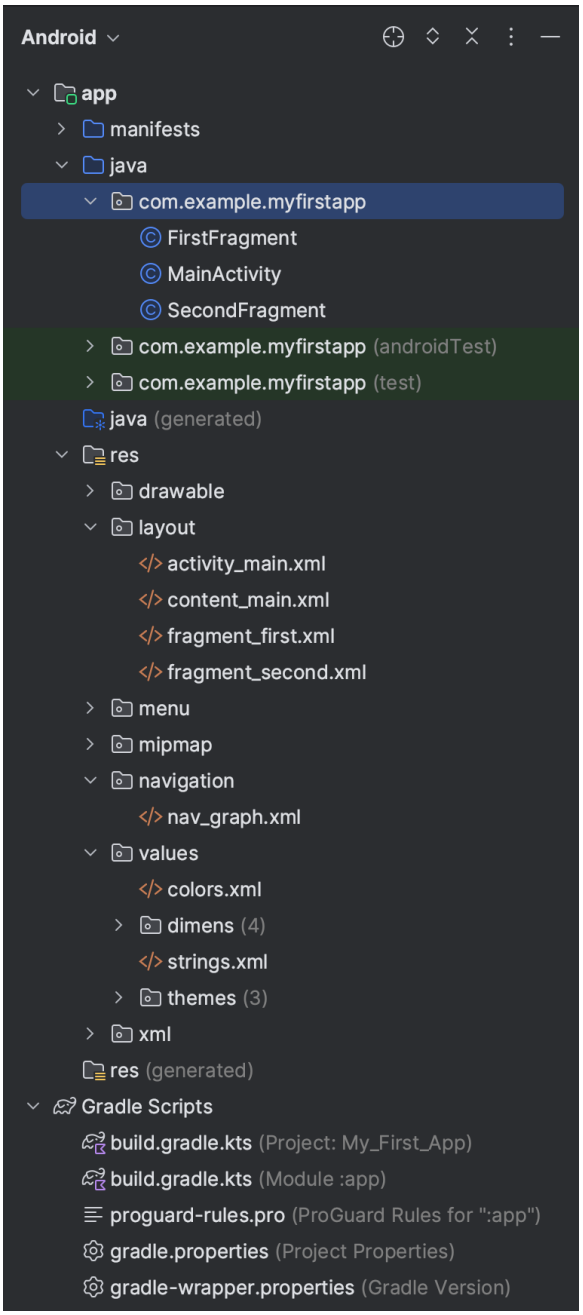
4. Name your project something like “My First Application”. Make sure that the Language is toggled to Java, and you can leave every other option as its default.



5. Click **Finish**

At this point, Android Studio will begin setting up and building your project for you. Notice in the bottom bar of Android Studio, it will show many Gradle files being downloaded.

Getting Familiar with the Project Structure



Based on the template selected, Android Studio will set up a number of folders and a project hierarchy for your application. In the previous step, you should have selected a **Basic Views Activity**. Now, there are a few ways to view your project structure but we will focus on the **Project** view. Double clicking the app folder at the top of the sidebar will expand this view.

The first folder we will look at is the **Java** folder. Upon expanding it, you will find three subfolders:

1. **com.example.myfirstapp:** This folder contains the Java source code files for your app.
2. **com.example.myfirstapp (androidTest):** This folder is where you would put your instrumented tests, which are tests that run on an Android device. It starts out with a skeleton test file.
3. **com.example.myfirstapp (test):** This folder is where you would put your unit tests. Unit tests don't need an Android device to run. It starts out with a skeleton unit test file.

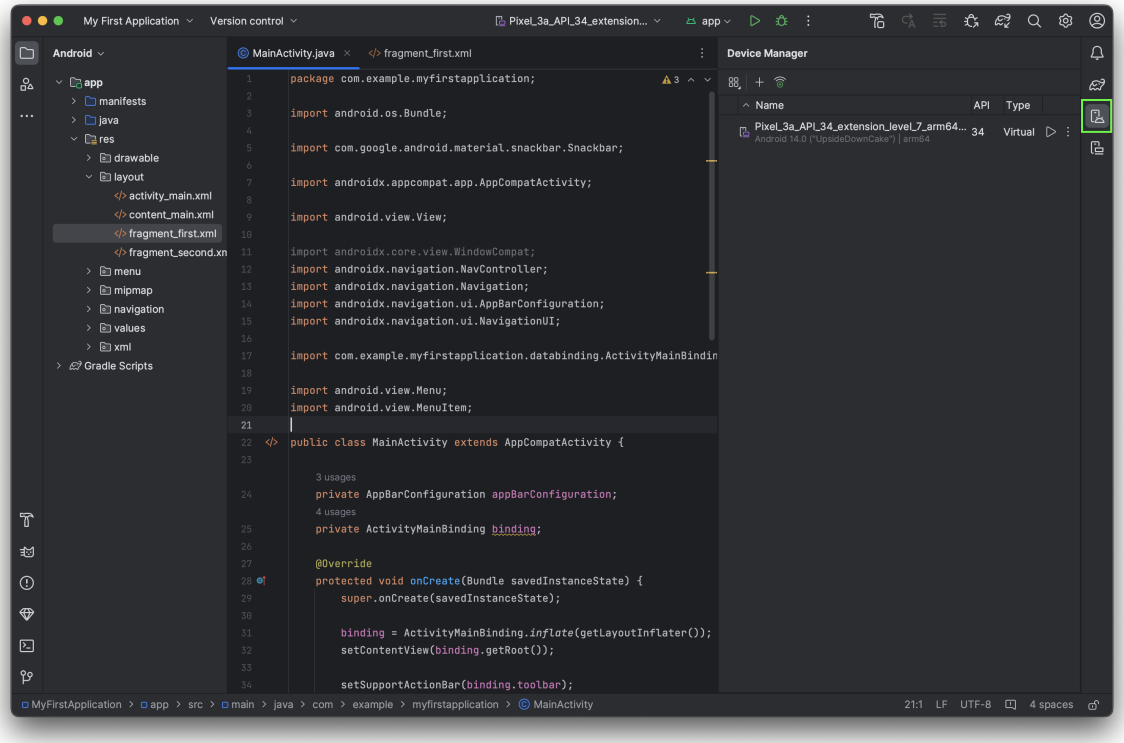
Next, let's take a look at the **res** folder. This folder contains all the resources for your app, including images, layout files, strings, icons, and styling. It includes these subfolders:

- **drawable**: All your app's images will be stored in this folder.
- **layout**: This folder contains the UI layout files for your activities. Currently, your app has one activity that has a layout file called `activity_main.xml`. It also contains `content_main.xml`, `fragment_first.xml`, and `fragment_second.xml`.
- **menu:** This folder contains XML files describing any menus in your app.
- **mipmap**: This folder contains the launcher icons for your app.
- **navigation:** This folder contains the navigation graph, which tells Android Studio how to navigate between different parts of your application.
- **values**: This folder contains resources, such as strings and colors, used in your app.

Creating a virtual device

Now that we've went over the basics of Android Studio's UI and the project structure, you might be wondering: where does this code actually run? We will use an Android emulator in this tutorial, so that you can experience your app right in front of you.

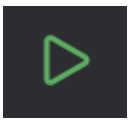
1. Click on the **Device Manager** on the right-hand side of Android Studio



2. If the list is empty, click on the **+** to add a device
3. From the popup that appears, select any device definition
4. For the system image, choose the first one under the recommended tab. If the next button is grayed out, click the download icon next to the release name. After the download is finished, the next button should be able to be clicked.
5. In the next dialog box, accept the defaults, and click Finish

Now, you can go ahead and unselect the device manager tab. We are ready to run the app.

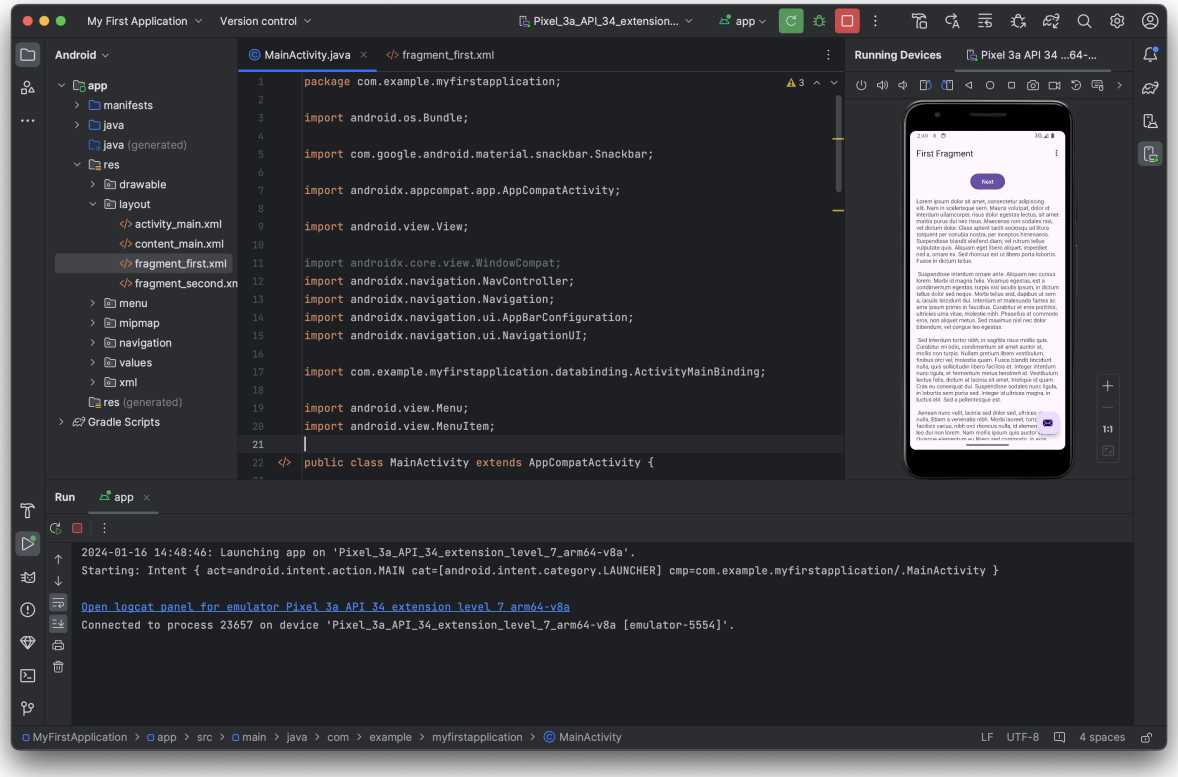
Running the app



1. In Android Studio, select **Run > Run 'app'** or click the **Run** icon in the toolbar. The icon will change when your app is already running

2. In **Run > Select Device**, under Available devices, select the virtual device that you just configured. This menu also appears in the toolbar.

Once your app builds and the emulator is ready, Android Studio uploads the app to the emulator and runs it. You should see your app as shown in the following screenshot:



Familiarizing with the layout editor

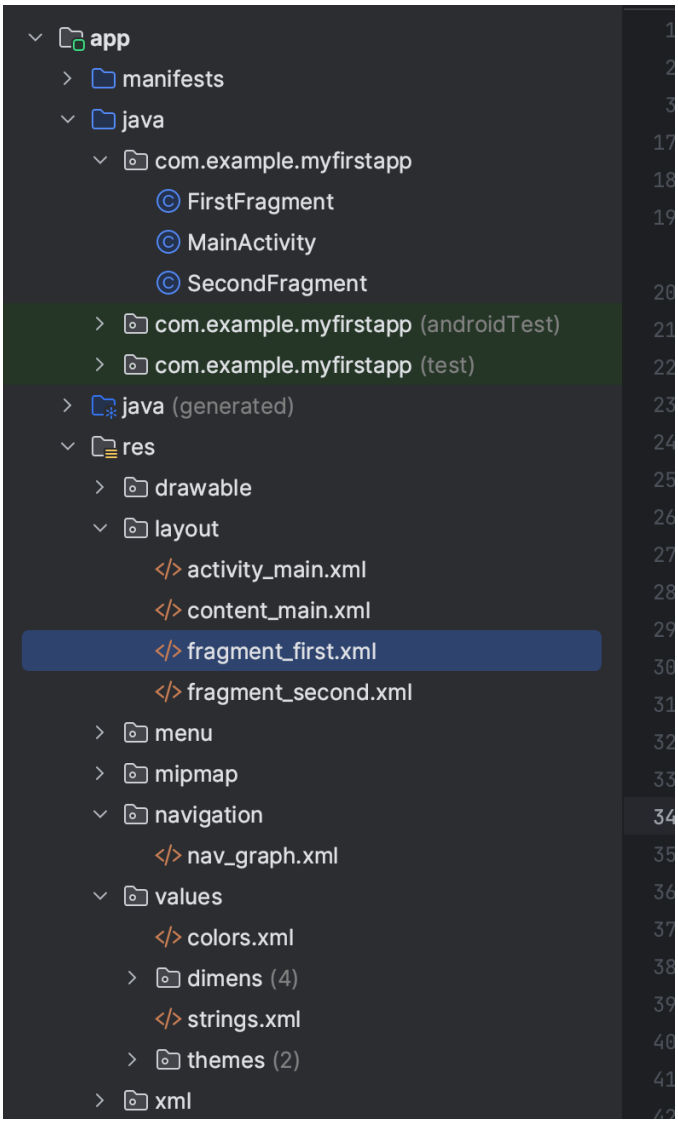
Each screen in your Android app is composed of one or more “fragments”. When you ran your app, the first screen you saw with “Hello first fragment” was created by **FirstFragment**. This fragment has an associated XML file (used for designing the user interface) and Java file (used for programming the functionality of the screen).

Layouts in Android Studio are written in
XML. However, there is also a layout editor which helps you visualize your design and

add/edit **Views**. A **View** is an element like a TextBox, Button, Image, etc. They are what make up our interface. You can create your interface by programming in XML or utilizing the layout editor. In this section of the tutorial, we will explore both.

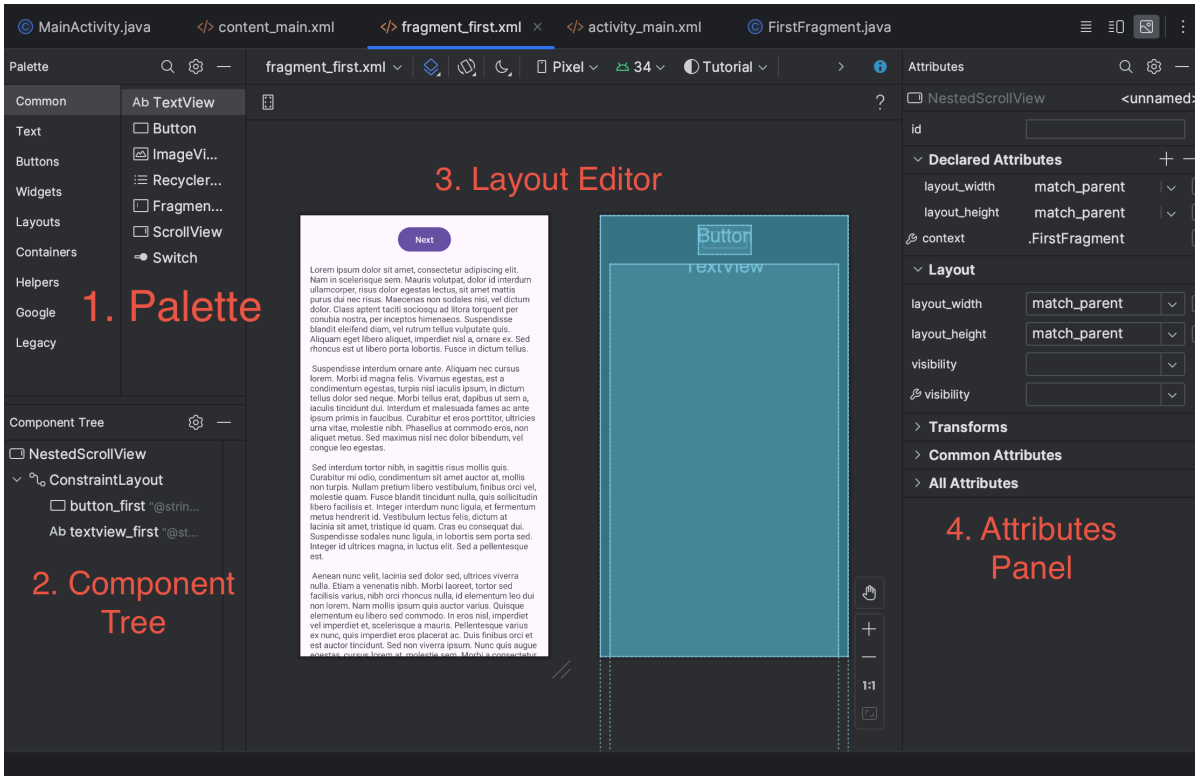
1) Exploring layout editor

1. We are going to be looking at `fragment_first.xml`. Navigate to the file using the project directory on the left. It should be under **app** → **res** → **layout** → **fragment_first.xml**

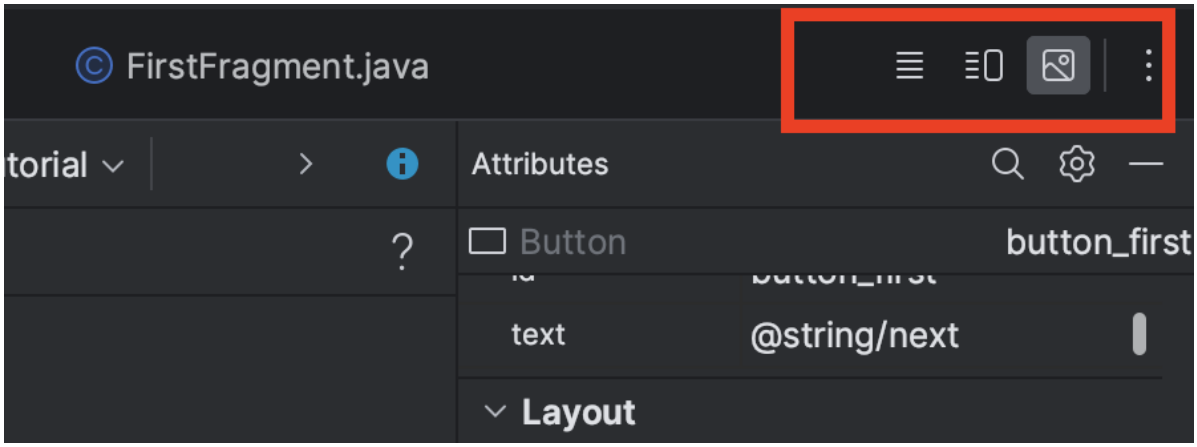


2. Double click to open it. Once you have opened the file, the layout editor will open up. It should look like the picture below. There is a lot going on here, so let's break it

down.



3. Let’s dive deeper into each section
    a. **Palette (1)** - This is where you can search for views to add to your layout. You can drag and drop from here onto the layout editor.
    b. **Component Tree (2)** - This is a hierarchy of the views in this file. Notice how the **Button** and **TextView** are tabbed in under the **ConstraintLayout.** This is because they are a subcomponent of this layout.
    c. **Layout Editor (3)** - This is a visual representation of the XML file. It gives you an idea of how your design will look like when viewed on a device.
    d. **Attributes Panel (4)** - When you click on a View, this panel displays all its “attributes”, like width, height, color, text, etc.
4. In the top right, above the Attributes Panel, there are 3 buttons. These allow you to toggle between the layout editor, the XML code, and a split screen of both.

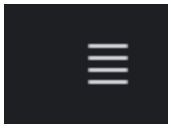


2) Learning about Component Tree and View Hierarchies

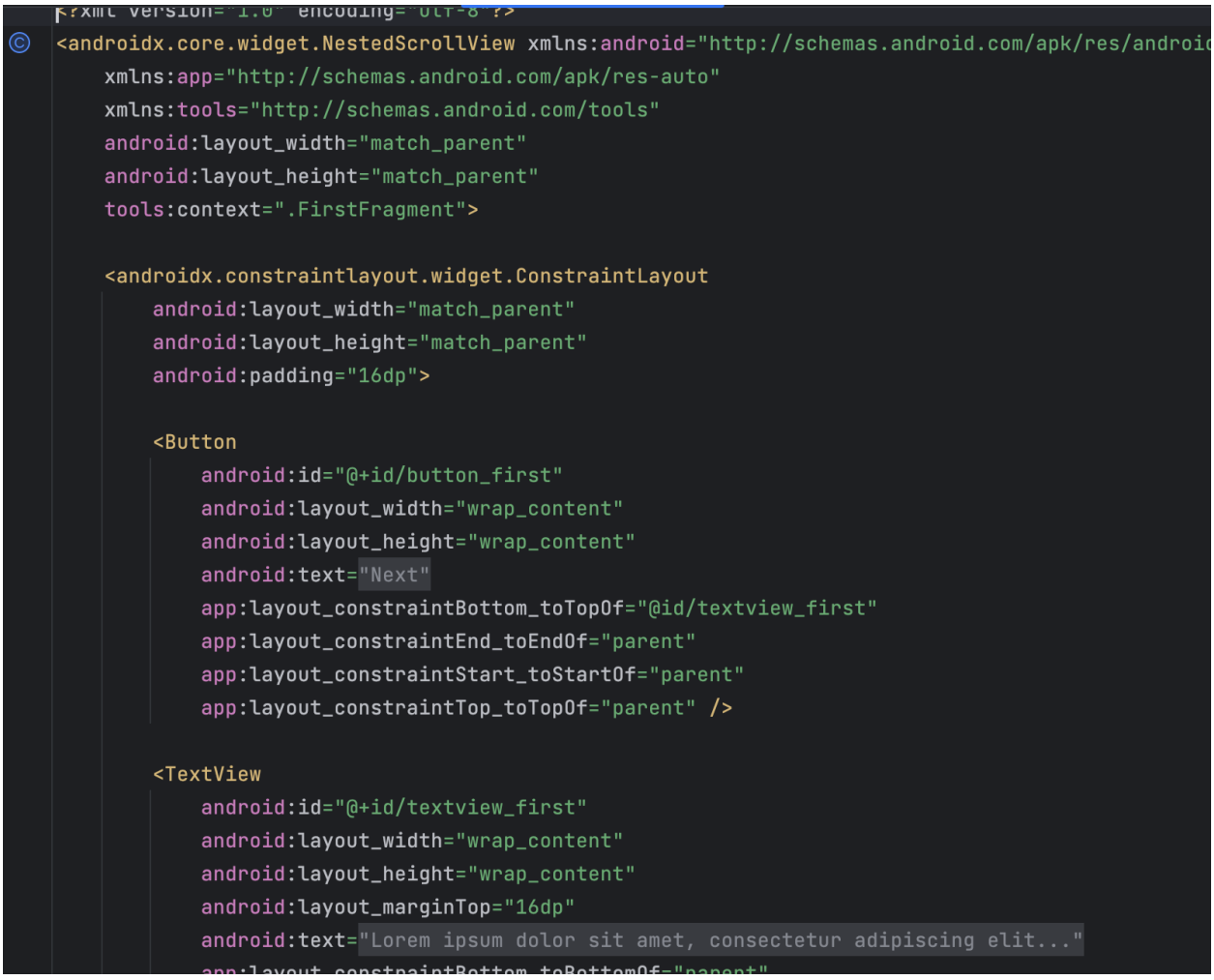
1. In our **Component Tree**, the view at the very top is the **NestedScrollView**. This view does exactly what it sounds like. It allows our screen to be scrollable.

Every layout has a root view that contains all the other views. You can add root views inside other root views. Like in our case, we can see that under our **NestedScrollView**, we have a **ConstraintLayout.** This layout contains our **Button** (`button _first`) and **TextView** (`textview_first`).

2. Switch to the XML view for our layout.



3. You can see this hierarchy in the XML code as well. If a View is “tabbed in”, it falls under that root view.



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.core.widget.NestedScrollView 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=".FirstFragment">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="16dp">

        <Button
            android:id="@+id/button_first"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Next"
            app:layout_constraintBottom_toTopOf="@id/textview_first"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/textview_first"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="16dp"
            android:text="Lorem ipsum dolor sit amet, consectetur adipiscing elit..."
            app:layout_constraintBottom_toBottomOf="parent"
```

3) Changing Property Values

1. Stay on the XML code editor.
2. Find the `TextView` element.

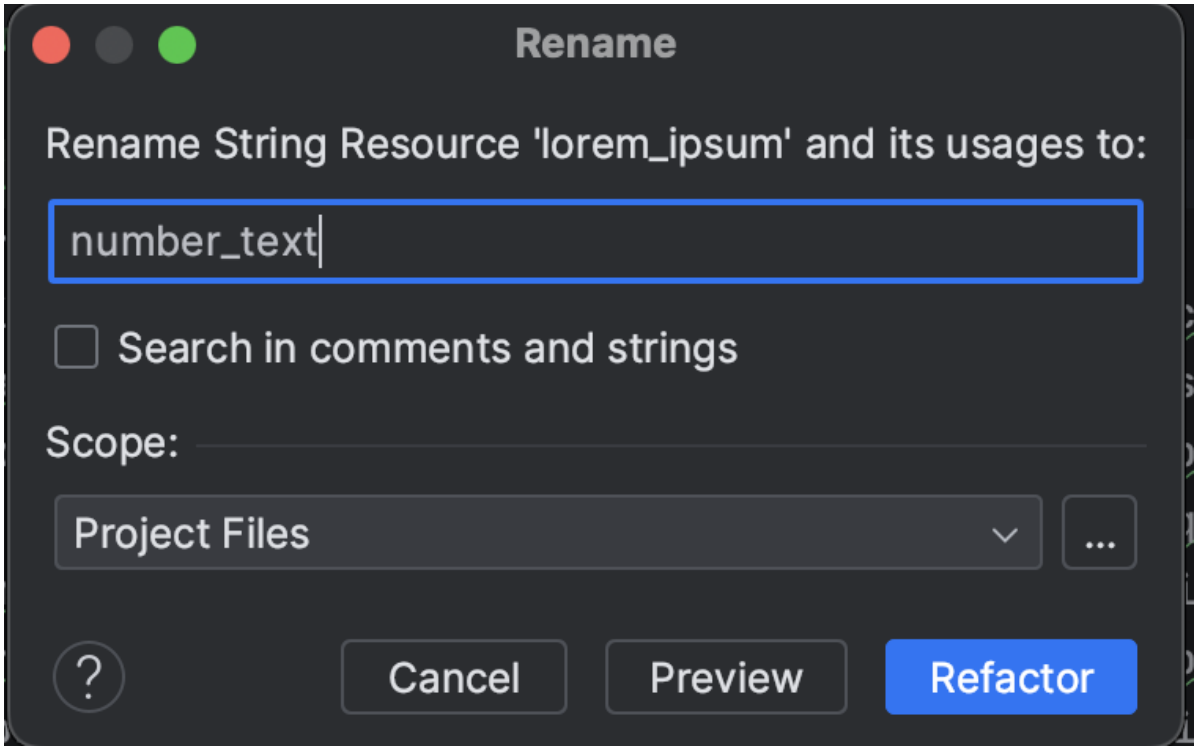
```
<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="Lorem ipsum dolor sit amet, consectetur adipiscing el
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/toast_button" />
```

3. Click on the string in the **text** property of the TextView. It should say something like `"@string/lorem_ipsum"`.
4. Right-click on this and click **Go To → Declaration or Usages**
5. This should bring you to a file called `values/string.xml`

```
<resources>
    <string name="app_name">Tutorial</string>
    <string name="action_settings">Settings</string>
    <!-- Strings used for fragments for navigation -->
    <string name="first_fragment_label">First Fragment</string>
    <string name="second_fragment_label">Second Fragment</string>
    <string name="next">Next</string>
    <string name="previous">Previous</string>

    <string name="lorem_ipsum">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam in scelerisque sem. Mauris
        volutpat, dolor id interdum ullamcorper, risus dolor egestas lectus, sit amet mattis purus
        dui nec risus. Maecenas non sodales nisi, vel dictum dolor. Class aptent taciti sociosqu ad
        litora torquent per conubia nostra, per inceptos himenaeos. Suspendisse blandit eleifend
        diam, vel rutrum tellus vulputate quis. Aliquam eget libero aliquet, imperdiet nisl a,
        ornare ex. Sed rhoncus est ut libero porta lobortis. Fusce in dictum tellus.\n\n
        Suspendisse interdum ornare ante. Aliquam nec cursus lorem. Morbi id magna felis. Vivamus
        egestas, est a condimentum egestas, turpis nisl iaculis ipsum, in dictum tellus dolor sed
        neque. Morbi tellus erat, dapibus ut sem a, iaculis tincidunt dui. Interdum et malesuada
        fames ac ante ipsum primis in faucibus. Curabitur et eros porttitor, ultricies urna vitae,
        molestie nibh. Phasellus at commodo eros, non aliquet metus. Sed maximus nisl nec dolor
        bibendum, vel congue leo egestas.\n\n
        Sed interdum tortor nibh, in sagittis risus mollis quis. Curabitur mi odio, condimentum sit
        amet auctor at, mollis non turpis. Nullam pretium libero vestibulum, finibus orci vel,
        molestie quam. Fusce blandit tincidunt nulla, quis sollicitudin libero facilisis et. Integer
        interdum nunc ligula, et fermentum metus hendrerit id. Vestibulum lectus felis, dictum at
        lacinia sit amet, tristique id quam. Cras eu consequat dui. Suspendisse sodales nunc ligula,
        in lobortis sem porta sed. Integer id ultrices magna, in luctus elit. Sed a pellentesque
```

`string.xml` is a file that contains static string constants. When elements like TextViews and Buttons have strings, it is important to put them in this file. This is because if you use this string in multiple places throughout your app, all you have to do is change it in this file instead of going through your entire app.

6. Right-click where it says `lorem_ipsum` . Click **Refactor** → **Rename.** Change the name to `number_text`. Hit Refactor.



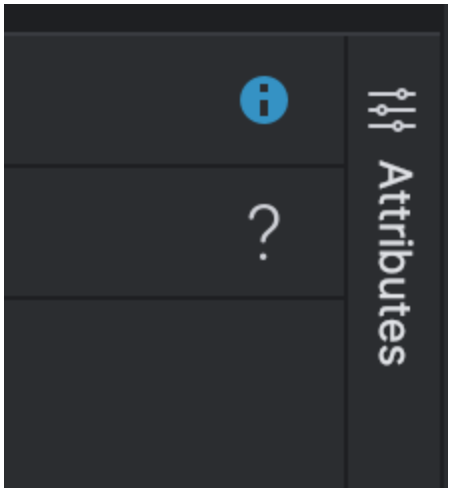
7. Then, delete all the words in between the string tags. Replace it with “0”.
8. It should look like this.

```
<string name="number_text">0</string>
```

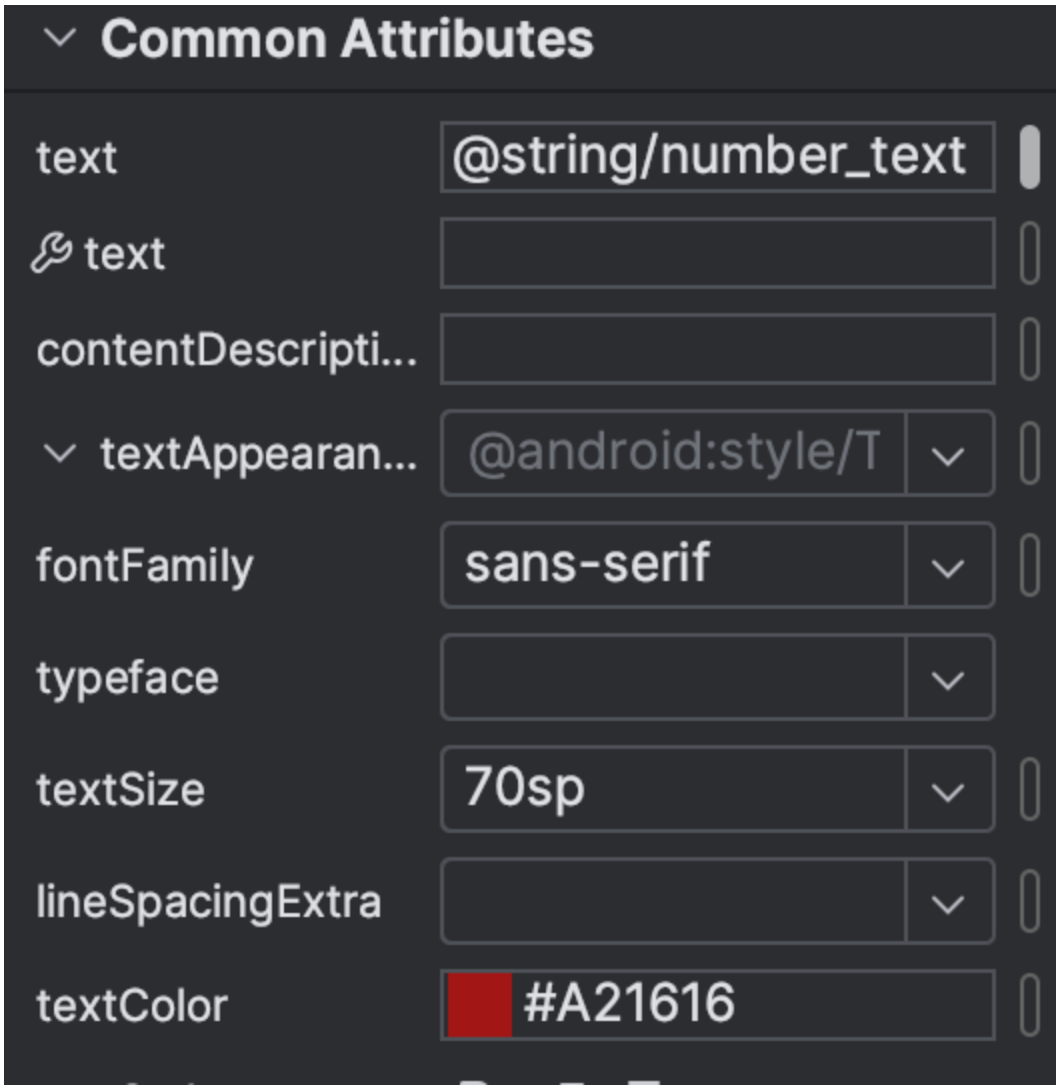
4) Changing text properties

1. Navigate back to `fragment_first.xml` .
2. Switch to the layout editor.
3. In the Component Tree on the left, click on `textview_first`. Look at the Attributes.

Note: If your attribute panel is missing, click this in the top right to reopen it.



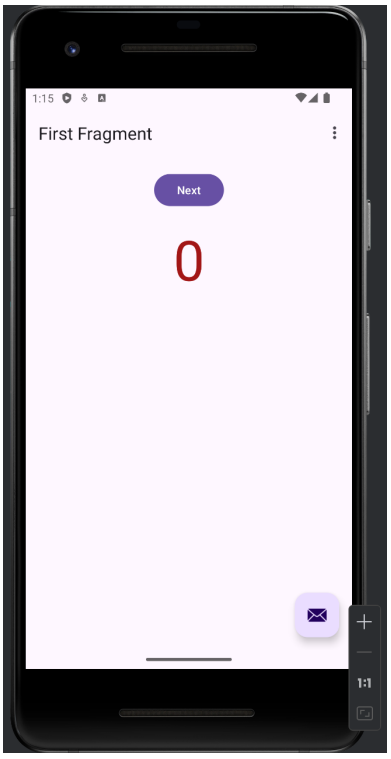
4. Scroll down in the Attributes panel until you see **Common Attributes**
5. Mess around with some of the text appearance properties like fontFamily, textSize, textColor, etc.



6. When you navigate back to XML view, you'll see that these properties have been added under TextView.

```
<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:fontFamily="sans-serif"
    android:text="@string/number_text"
    android:textColor="#A21616"
    android:textSize="70sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
```

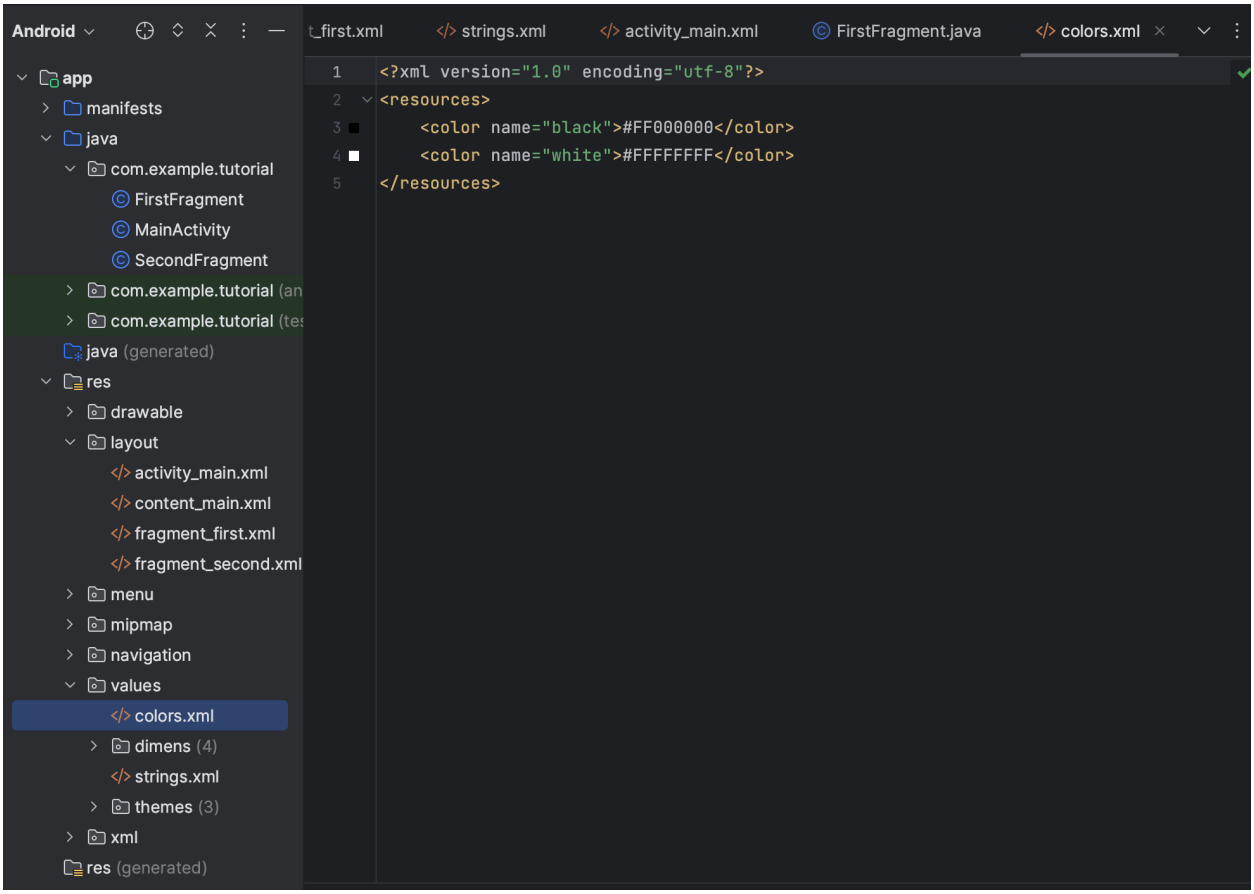
7. Run your app now to see your changes.



Designing your App

1) Adding the Color Resources

1. In the Project panel, go to **res → values → colors.xml**



2. Add this new color resource called **screenBackground.**

```
<color name="screenBackground">#2196F3</color>
```

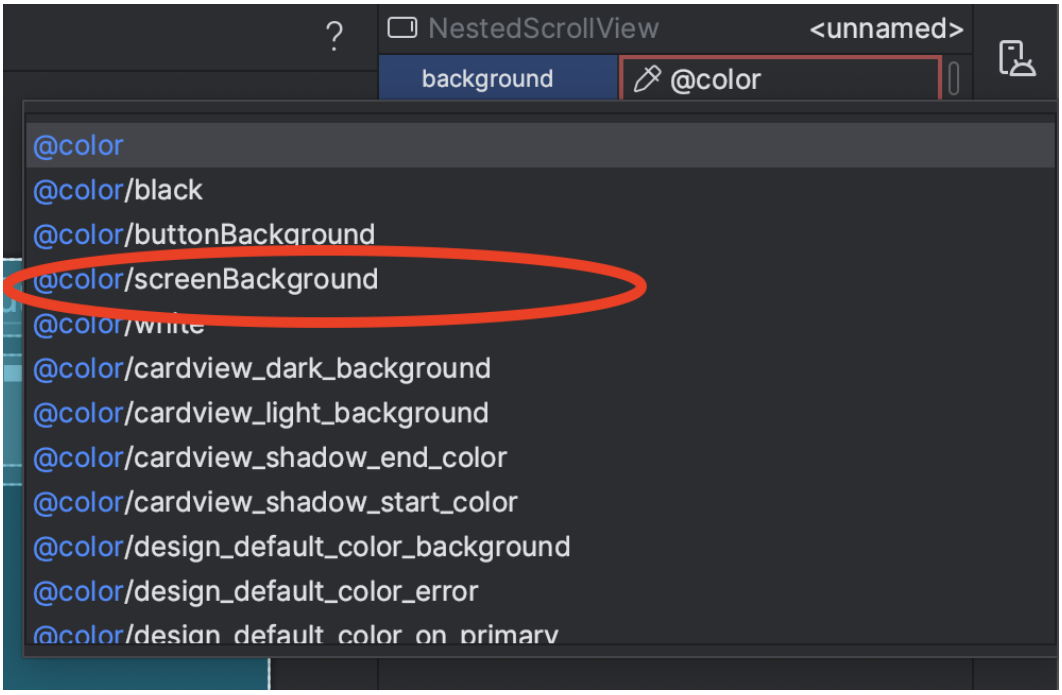
Note: the **#2196F3** is hexadecimal for the red-green-blue components of the color

3. Your `color.xml` file should look like this.

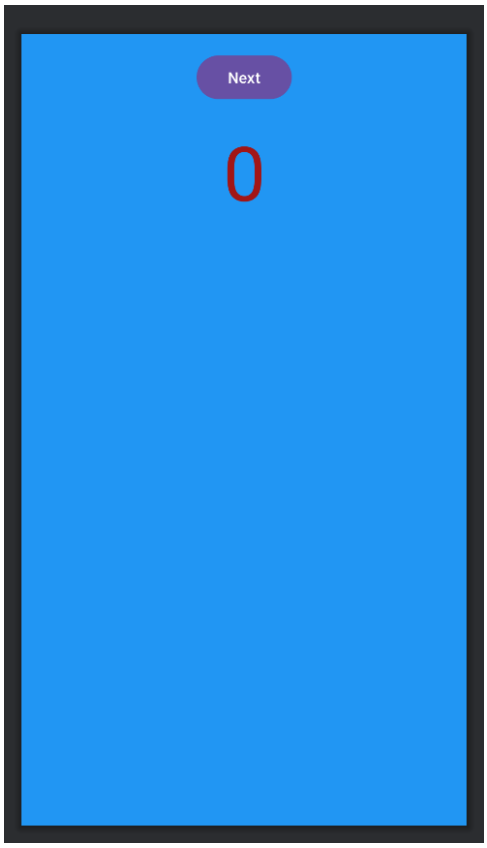
```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="screenBackground">#2196F3</color>
</resources>
```

Notice that the color defined by the hexadecimal shows up in a little square on the left.

4. Navigate back to the `fragment_first.xml` . Switch to layout editor view if you are not already on it.
5. Click on **NestedScrollView** in the **ComponentTree.**
6. Locate the “background” attribute in the Attribute panel. You can use the search bar to make this easier.
7. Type in @color and select the one that says screenBackground.



8. Your screen should be blue like this.



9. One quick adjustment we need to make is to get rid our **NestedScrollView**, and make our **ConstraintLayout** the highest root view. Switch to XML view.
10. At the top, where it says

```
<androidx.core.widget.NestedScrollView xmlns:android="http://sch
    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:background="@color/screenBackground"
    tools:context=".FirstFragment">
```

11. Replace `androidx.core.widget.NestedScrollView` with `androidx.constraintlayout.widget.ConstraintLayout` . It should now look like this

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:androi
    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:background="@color/screenBackground"
    tools:context=".FirstFragment">
```

12. Make sure to change the bottom ScrollView tag from `</androidx.core.widget.NestedScrollView>` to `</androidx.constraintlayout.widget.ConstraintLayout>`

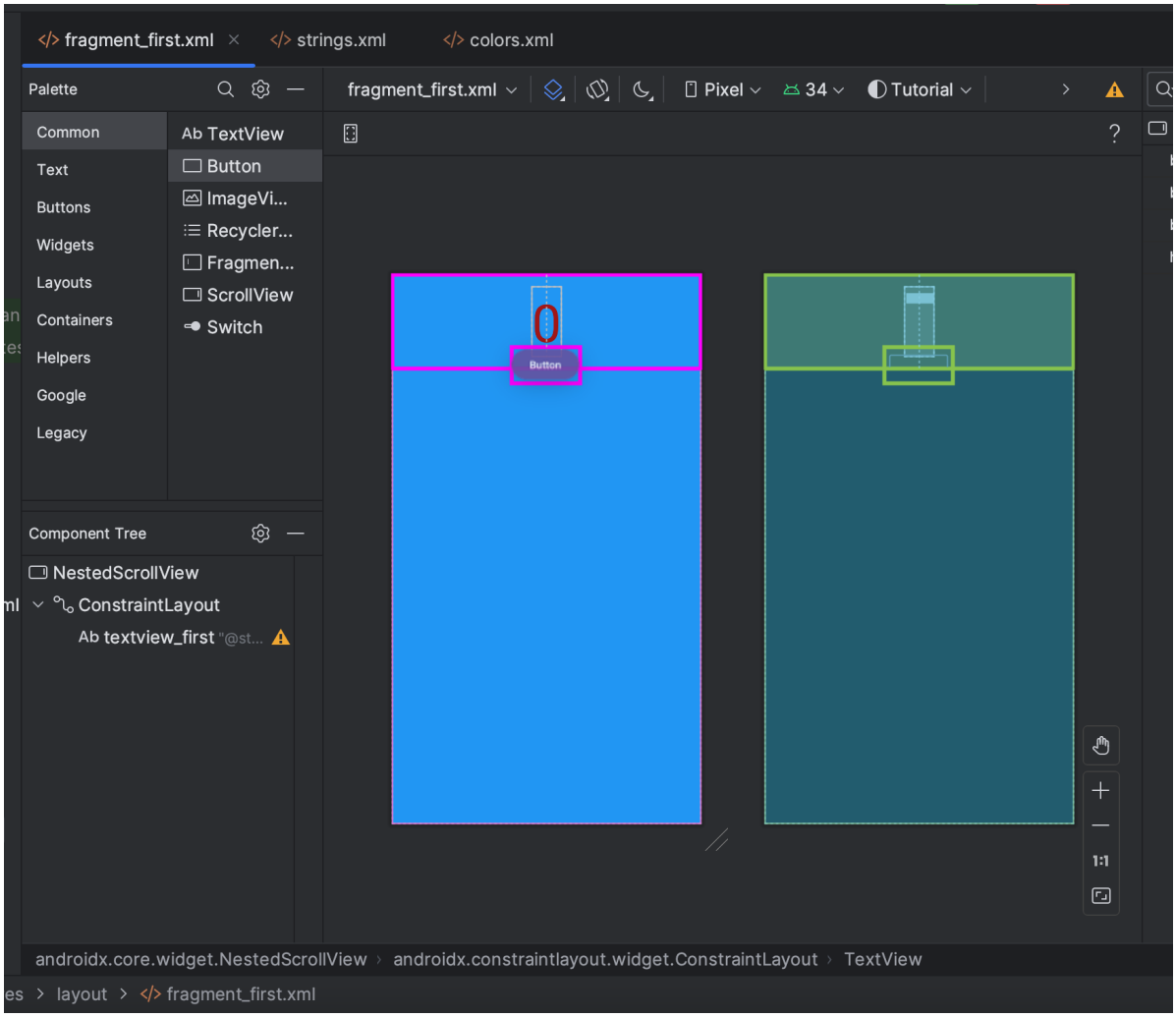
13. Now get rid of the inside ConstraintLayout altogether (not the views inside of it). That was a lot of steps, so to clarify. This is what your `first_fragment.xml` file should look like.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:androi
    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:background="@color/screenBackground"
    tools:context=".FirstFragment">


    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="64dp"
        android:layout_marginBottom="508dp"
        android:text="Button"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

    <TextView
```

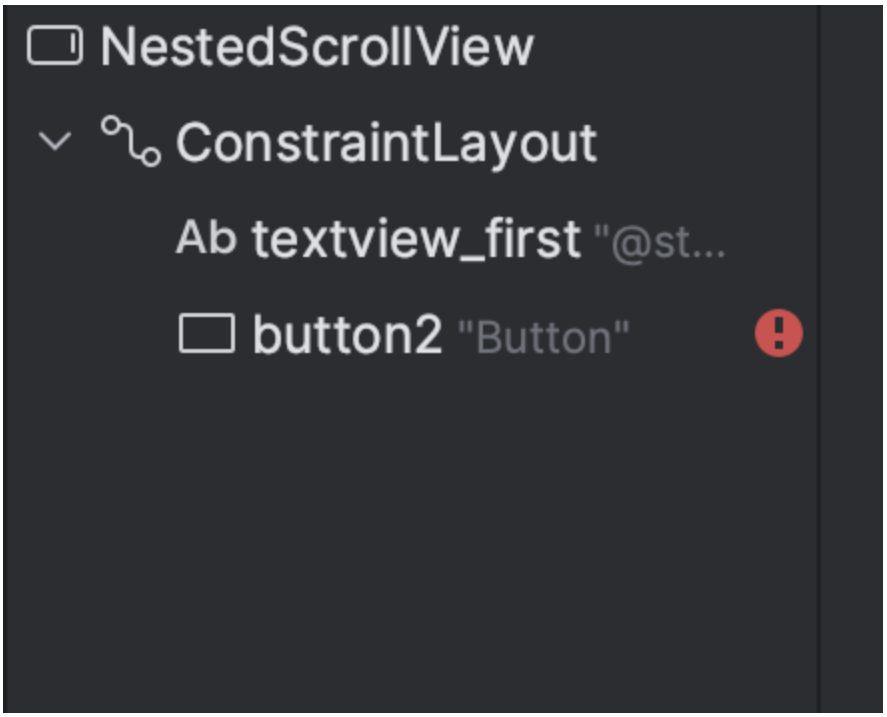
```
        android:id="@+id/textview_first"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:fontFamily="sans-serif"
        android:text="@string/number_text"
        android:textColor="#A21616"
        android:textSize="70sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        tools:layout_editor_absoluteY="16dp" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

3) Adding all our elements

1. Right-click on the button in the **Component Tree.** Click **Delete.** We will be adding three new buttons.
2. From the Palette on the left, drag and drop a **Button** right below the **TextView**

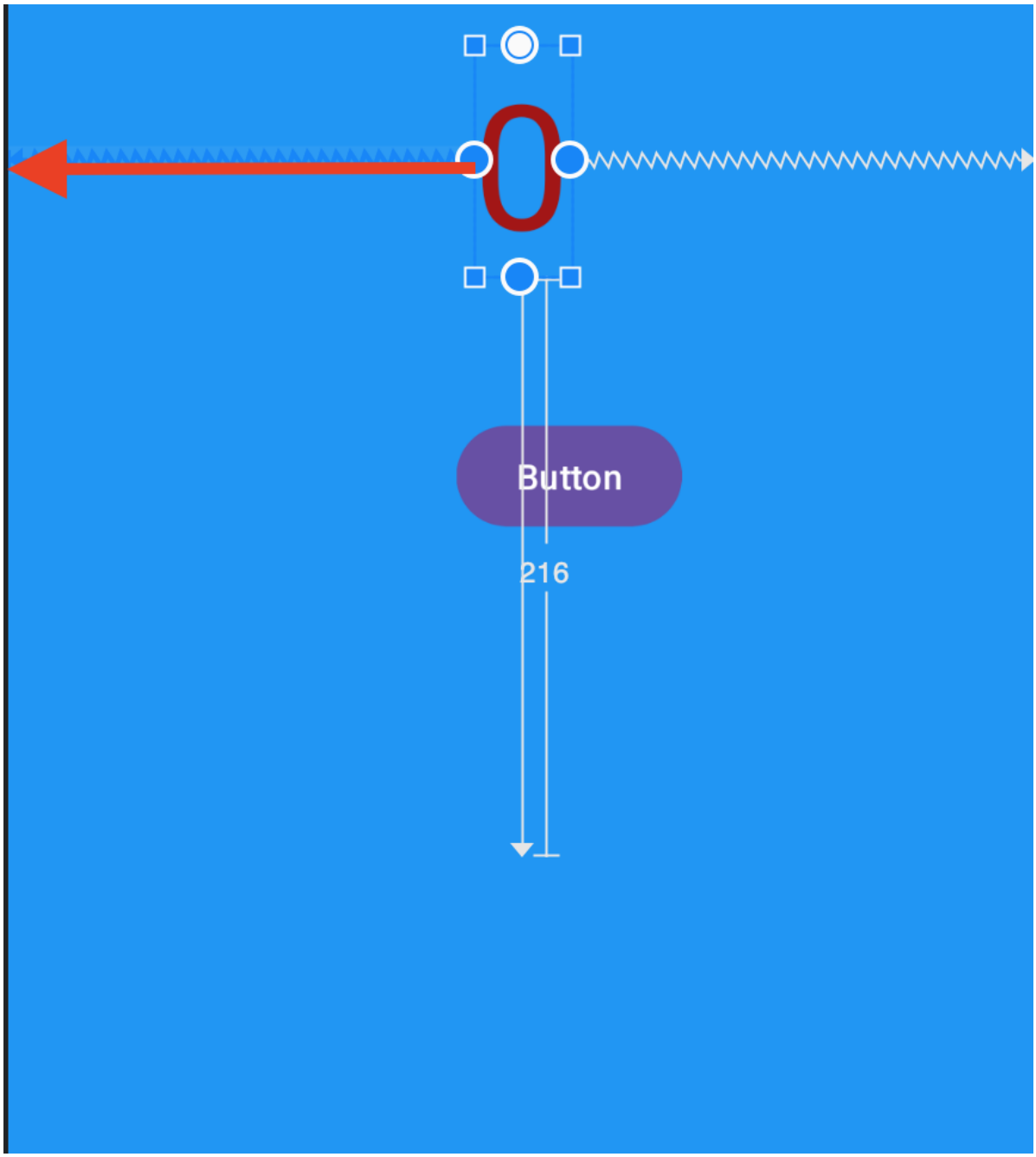


3. Notice that a Button has been added to the Component Tree under ConstraintLayout.

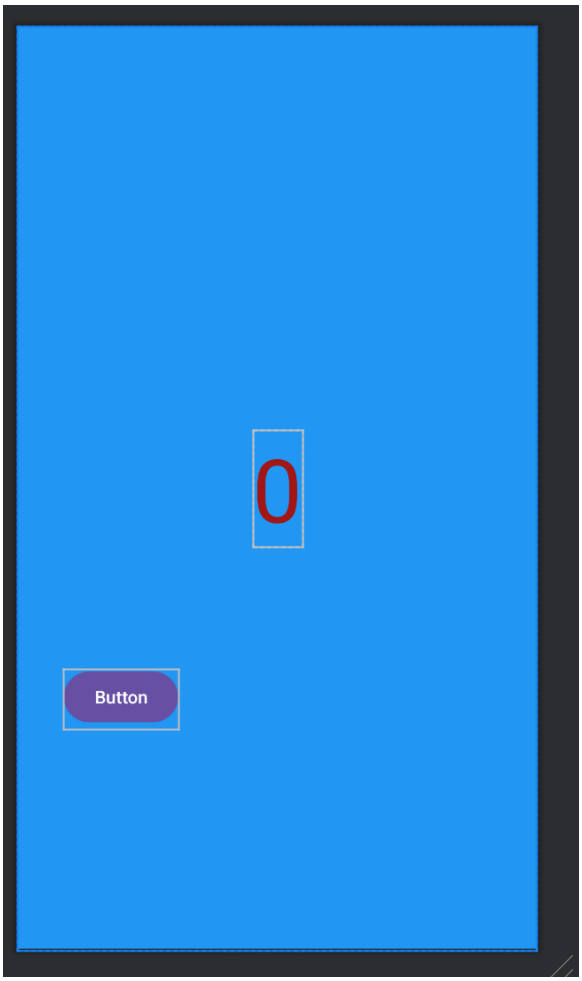


ConstraintLayouts are a little confusing. Here is a simplified explanation. When you add elements to the layout, you have to constrain them to either other views or to the sides of the screens.

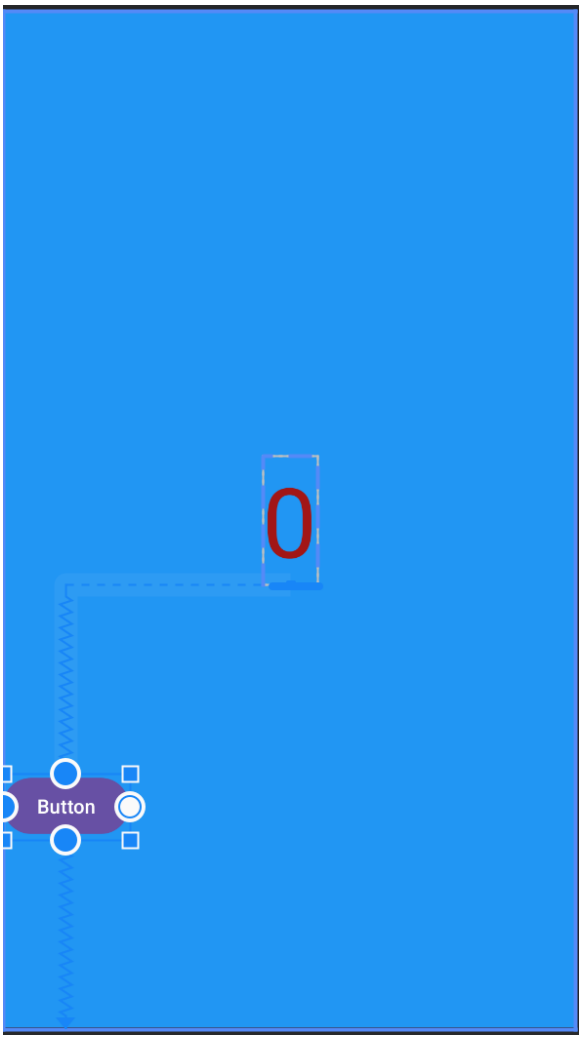
1. We need to add constraints to our Views.
2. If you haven't already, constrain our **TextView** to the left, right, top, and bottom of our screen. You can do this by clicking on the **TextView**, and clicking and dragging the circular buttons to their respective side. For instance, the **Button** on the top should be clicked and dragged to the top of the screen.



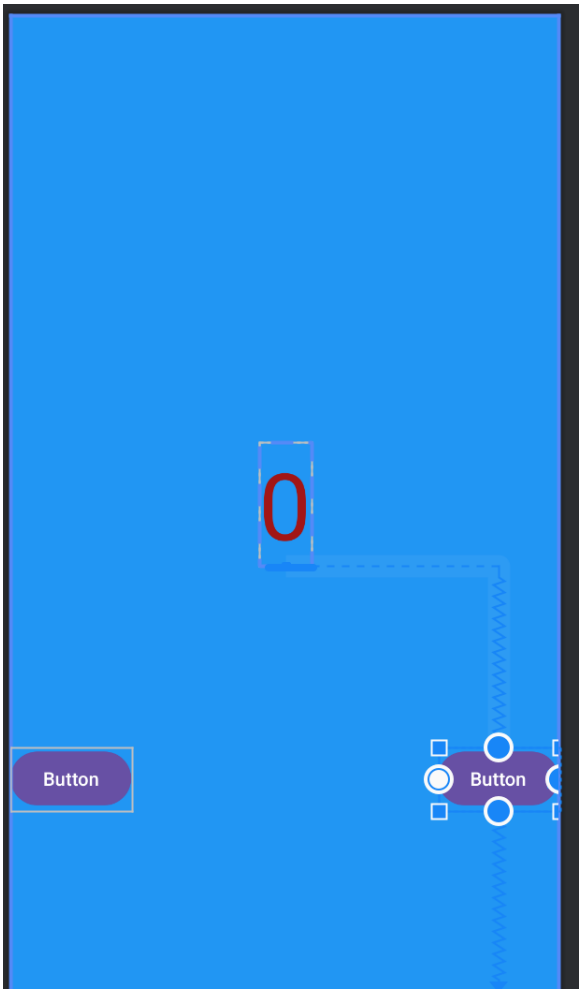
3. If you incorrectly add a constraint, you can delete it by holding the `Ctrl` key (`Cmd` key for Mac), hovering your cursor over the constraint, and clicking it.
4. Now that you have constrained your TextView to all the sides of the screen, it should be in the center like the picture below. If it doesn't look exactly like this, don't worry. We can correct it at the end.



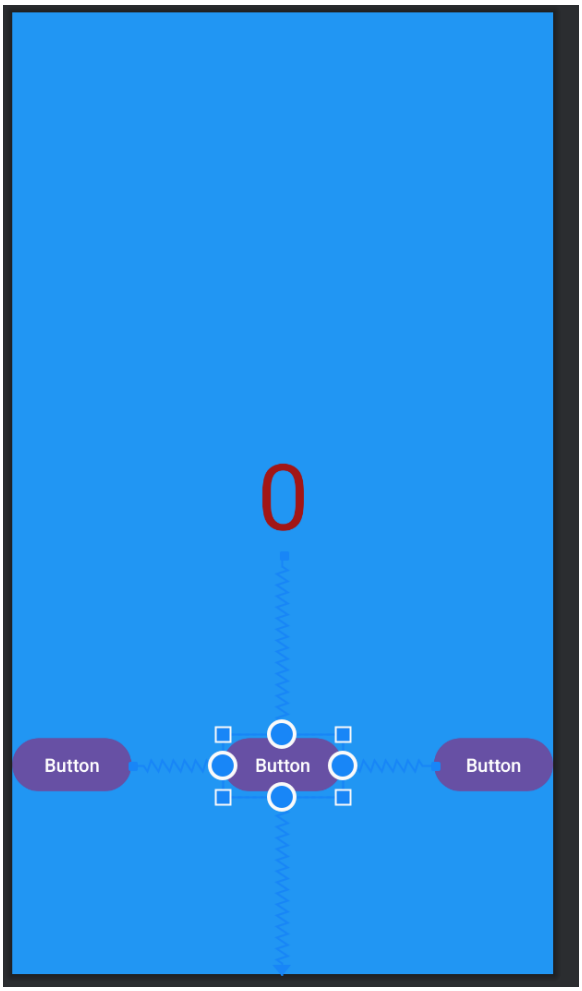
5. Now we need to add constraints for our **Button**. Constrain the top of the **Button** to the bottom of the **TextView**. Constrain the left and bottom side of the button to their respective sides of the screen. It should look something like the picture below. If it doesn't you can drag the button around until it does.



6. Now let's add another **Button**. Drag it from the palette, and add it below the **TextView**. It will go on the right side of the screen.
7. Constrain the top of the **Button** to the bottom of the **TextView**. Constrain the right and bottom side of the **Button** to their respective sides of the screen. Should look like this. Again, make adjustments as needed.



8. Finally, add your third button. This will go directly in the middle of our other two buttons.
9. Constrain the top of the **Button** to the bottom of the **TextView**. Constrain the bottom side of the **Button** to the bottom of the screen. Constrain the left and right side of the **Button** to the sides of the other two buttons. Our screen will now look like this.



4) Naming our elements

1. Now let's name our buttons to make it easier to differentiate them. Switch to XML view.
2. Right-click the value in the "id" field of the first **Button**. Click **Refactor** → **Rename** and change the name to `toast_button`.
3. Do the same for the second **Button**, but change the name to `count_button` .
4. Once again, do the same for the third **Button**, but change the name to `random_button` .
5. Let's rename our buttons now. Navigate back to the `strings.xml` file. Add the following in between the resource tags.

```
<string name="toast_button_text">Toast</string>
<string name="count_button_text">Count</string>
<string name="random_button_text">Random</string>
```

6. Go back to `fragment_first.xml`. Change the “text” field of each **Button** to its respective string resource. For instance,

```
<Button
        android:id="@+id/toast_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/toast_button_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textview_first
```

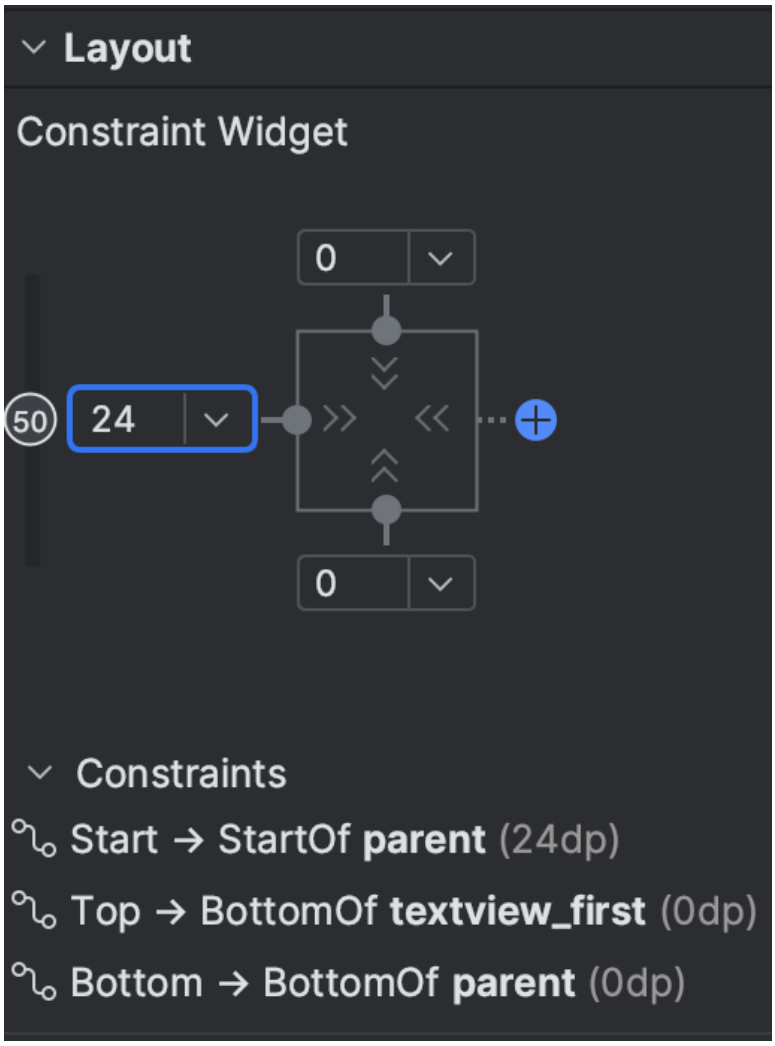
7. Let’s now put the finishing touches!

5) Finishing Touches

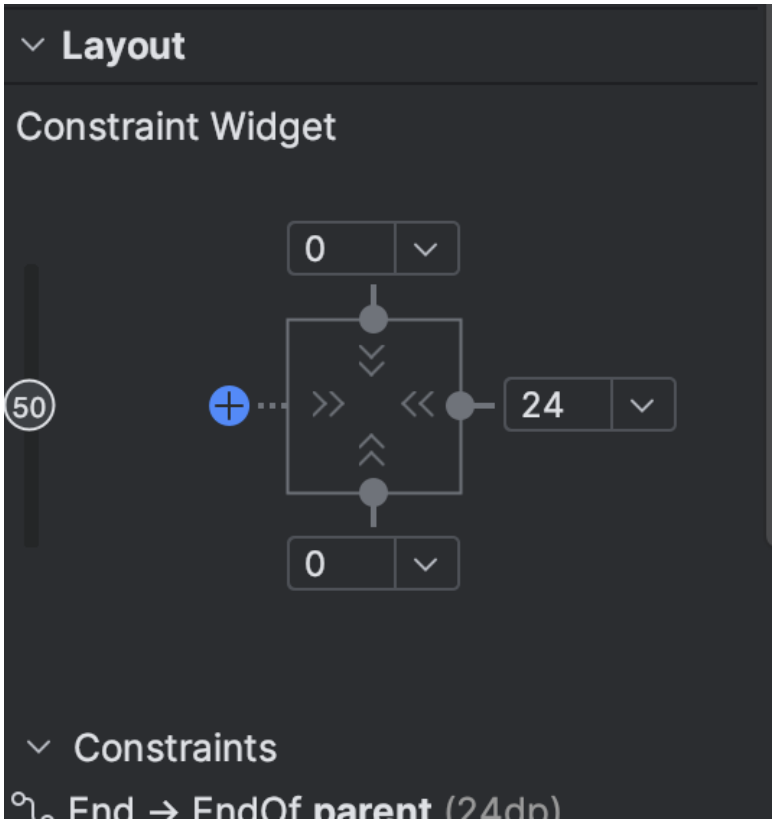
1. Ensure that the `textSize` of the TextView is 72sp, the `textColor` is white, the `fontFamily` is sans-serif, and the `textStyle` is bold.

```
android:textSize="72sp"
android:textColor="@color/white"
android:fontFamily="sans-serif"
android:textStyle="bold"
```

2. Our **Toast** and **Random** buttons are a little too close to the edges of the screen. Switch back to layout editor view.
3. Click the **Toast** button, and open the **Attributes** panel.
4. In the **Constraint Widget,** give the Toast button a left margin of 24dp. You can do this by typing 24 in the box on the left like below.



8. Do the same for the **Random** button, except give it a right margin of 24dp, like so



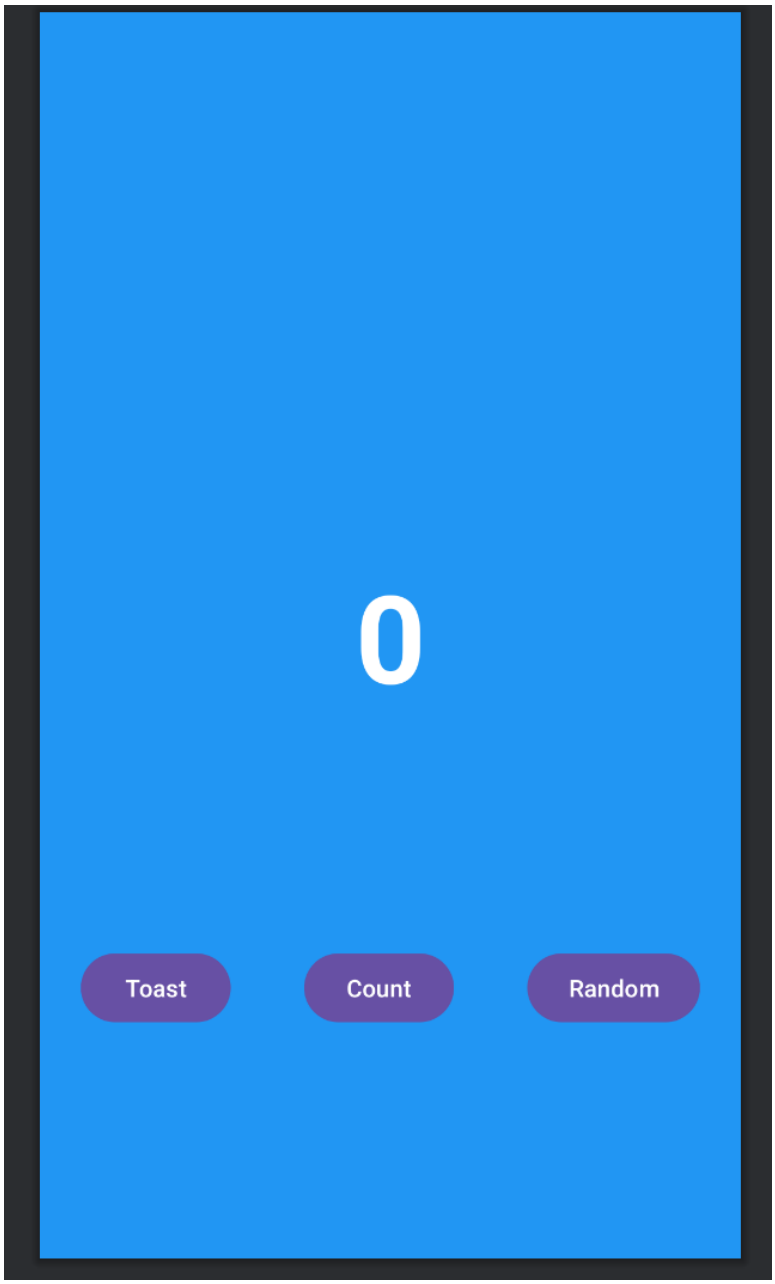
9. Whew! That was a lot of steps. Let’s regroup. Here is what your XML file should look like.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:androi
    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:background="@color/screenBackground"
    tools:context=".FirstFragment">

    <TextView
        android:id="@+id/textview_first"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/number_text"
        android:textSize="72sp"
```

```
        android:textColor="@color/white"
        android:fontFamily="sans-serif"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/toast_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:text="@string/toast_button_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textview_first


    <Button
        android:id="@+id/random_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="24dp"
        android:text="@string/random_button_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textview_first

    <Button
        android:id="@+id/count_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/count_button_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toStartOf="@+id/random_button"
```

```
        app:layout_constraintStart_toEndOf="@+id/toast_button"
        app:layout_constraintTop_toBottomOf="@+id/textview_first

</androidx.constraintlayout.widget.ConstraintLayout>
```

7. And here is what your screen should look like. You are one step closer to completing your first app!



Making the App Interactive

So far we have added buttons to our app's main screen, but as of right now, they do not do anything. Let's change that and make them interactive!

First, let's enable **Auto Imports** in our settings so that Android Studio will automatically enable imports for any classes used by your Java code:

1. In Android Studio, click the Settings icon in the top right next to your profile.
2. Select **Auto Imports**. In the **Java** section, make sure **Add Unambiguous Imports on the fly** is checked. Press **Apply** and **OK**.

Now, let's attach a Java method to our **Toast** button to make it give a toast when the user presses the button.

1. Open **FirstFragment.java** (**app** > **java** > **com.example.android.myfirstapp** > **FirstFragment**).

   This class has three methods, `onCreateView()` , `onViewCreated()` , and `onDestroyView()` . The first two methods execute when the fragment starts.

   The **id** for a view helps us identify that view distinctly from other views. So, it is a good idea to use good naming conventions. Currently, the basic activity template uses the binding class for setting click listeners. We will use the `findViewByID()` method to single in on a specific **view** (in this case, our `toast_button`). In code, we reference the `toast_button` using its id, `R.id.toast_button`.

   We then set up a click listener using `setOnClickListener`

2. Take a look at `onViewCreated()`. It sets up a click listener for the `random_button`, which was originally created as the **Next** button.

```
view.findViewById(R.id.random_button).setOnClickListener(new
View.OnClickListener() {
   @Override
   public void onClick(View view) {
       NavHostFragment.findNavController(FirstFragment.this)
               .navigate(R.id.action_FirstFragment_to_SecondF
ragment);
```

```
    }
     });
```

Let’s quickly explain this code:

- Use the `findViewById()` method with the id of the desired view as an argument, then set a click listener on that view.
- In the body of the click listener, use an action, which, currently, has not been changed and navigates to another fragment. (We will learn about that later.)

3. Let’s now replace the action in our click listener to display a simple, pop-up toast message. Here is the code:

```
view.findViewById(R.id.toast_button).setOnClickListener(new V
iew.OnClickListener() {
   @Override
   public void onClick(View view) {
       Toast myToast = Toast.makeText(getActivity(), "Hello t
oast!", Toast.LENGTH_SHORT);
       myToast.show();
   }
});
```

Now, rerun the app and press the **Toast** button. Did you see the toast message? If you did, great! If not, make sure that your code matches ours from above.

4. Finally, if you’d like to extract the toast message string into a resource as you did for the button texts, feel free to do so!

Now, let’s make our count button update the number on screen!

Our previous toast method did not interact with any other views. Let’s now take it up a notch and add some behavior to our **Count** button to update other views in our layout!

1. In our `fragment_first.xml` layout file, note the `id` for the `TextView`.

2. Head back to our **FirstFragment.java**, and add a click listener for our `count_button` in the `onViewCreated()` method like we did during out implementation of the **Toast** button. Then, for our action, call a not-yet defined method, `countMe()` like below:

```
view.findViewById(R.id.count_button).setOnClickListener(new V
iew.OnClickListener() {
   @Override
   public void onClick(View view) {
       countMe(view);
   }
});
```

3. In the `FirstFragment` class (outside of the `onViewCreated()` method), add the private method `countMe()` that will take a single **View** object argument.

```
private void countMe(View view) {

}
```

4. Now, get the value of the `showCountTextView` variable we created before and convert the value to a number:

```
...
    // Get the value of the text view
    String countString = showCountTextView.getText().toString()
    // Convert value to a number and increment it
    Integer count = Integer.parseInt(countString);
    count++;
```

5. Display the new value in the `TextView` by programmatically setting the `text` property of the `TextView`

```
   ...
   // Display the new value in the text view.
```

```
    showCountTextView.setText(count.toString());
```

Your final implementation of `countMe()` should look like so:

```
private void countMe(View view) {
    // Get the value of the text view
    String countString = showCountTextView.getText().toString
();
    // Convert value to a number and increment it
    Integer count = Integer.parseInt(countString);
    count++;
    // Display the new value in the text view.
    showCountTextView.setText(count.toString());
}
```

Let's now Cache the TextView for repeated use

Unfortunately our current implementation calls `findViewById()` every time `countMe()` is called. `findViewById()` is a relatively time consuming method. So, instead let's find the view once on creation and cache it for future use.

1. In the `FirstFragment` class before any methods, add a member variable for `showCountTxtView` of type `TextView`.

```
TextView showCountTextView;
```

2. In `onCreateView()`, let's call `findViewById()` to get the `TextView` that shows the count. The `findViewById()` method must be called on a `View` where the search for the requested ID should start, so assign the layout view that is currently returned to a new variable, `fragmentFirstLayout`, instead. Note that we use a view object rather than the `FragmentFirstBinding` **binding** member variable autoimplemented from the **Basic View Activity** template.

```
// Inflate the layout for this fragment
View fragmentFirstLayout = inflater.inflate(R.layout.fragment
```

```
_first, container, false);
```

3. Call `findViewById()` on `fragmentFirstLayout`, and specify the `id` of the view to find, `textview_first`. Cache that value in `showCountTextView`.

```
   ...
    // Get the count text view
    showCountTextView = fragmentFirstLayout.findViewById(R.i
d.textview_first);
```

4. Return `fragmentFirstLayout` from `onCreateView()`. Again, note we are not using **binding**, so you may remove `return binding.getRoot();` and replace it with the following code.

```
...
return fragmentFirstLayout;
```

5. Here is the final implementation including the declaration of `showCountTextView`:

```
TextView showCountTextView;

@Override
public View onCreateView(
       LayoutInflater inflater, ViewGroup container,
       Bundle savedInstanceState
) {
   // Inflate the layout for this fragment
   View fragmentFirstLayout = inflater.inflate(R.layout.fragm
ent_first, container, false);
   // Get the count text view
   showCountTextView = fragmentFirstLayout.findViewById(R.id.
textview_first);
```

```
    return fragmentFirstLayout;
}
```

6. Now, let's run the app and see the result!

Implementing the Second Fragment

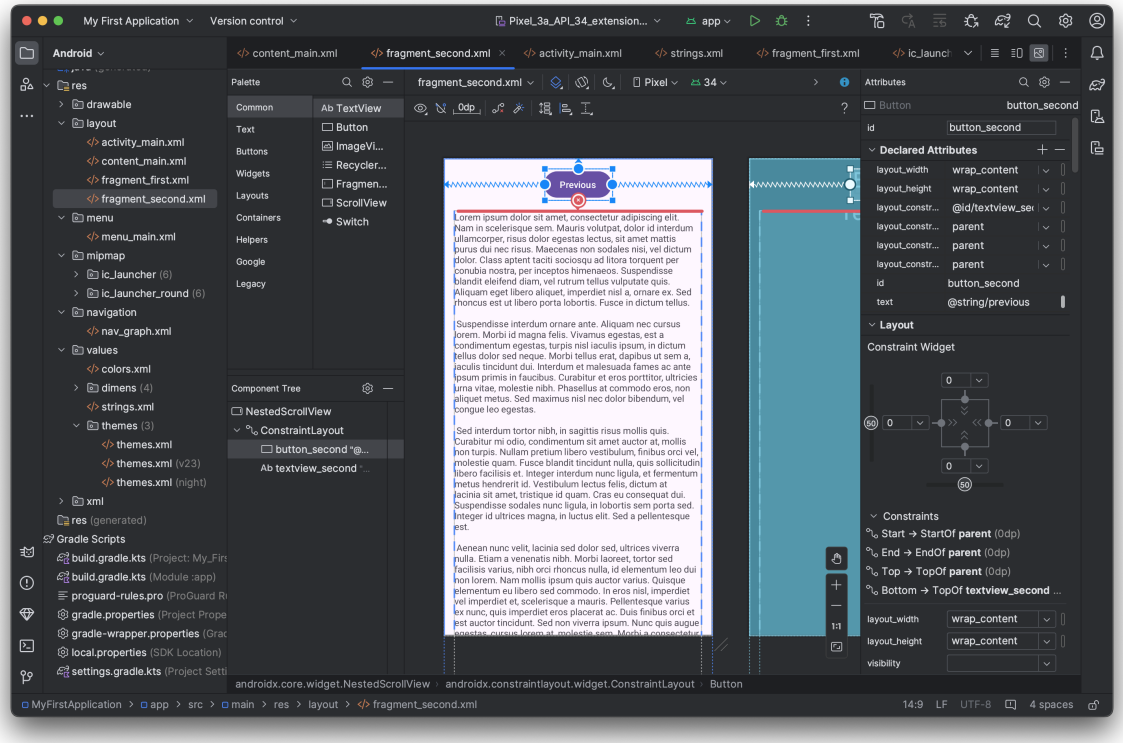
Up until now, you have been focusing on only the first fragment of your app. In this section, you will work on creating a **Random** button that will display a number between 0 and your count from the first screen.

Adding a TextView for the Random Number

1. Open `fragment_second.xml` (**app > res > layout > fragment_second.xml**) and switch to **Design View** if needed. Notice that it has a `ConstraintLayout` that contains a `TextView` and a `Button`.
2. Open the `xml` view of `fragment_second.xml`. Under the metadata for the `NestedScrollView`, add the following line of code: `android:fillViewport="true"`. Your current `fragment_second.xml` file should begin like this:

```
<androidx.core.widget.NestedScrollView xmlns:android="http://sch
    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:fillViewport="true"
    tools:context=".SecondFragment">
            ...
</androidx.core.widget.NestedScrollView>
```

3. Go ahead and remove the chain constraints between the `TextView` and the `Button` (in the design view, click on the element, and then hold **CTRL** or **CMD** on Mac while clicking the circles on the sides)



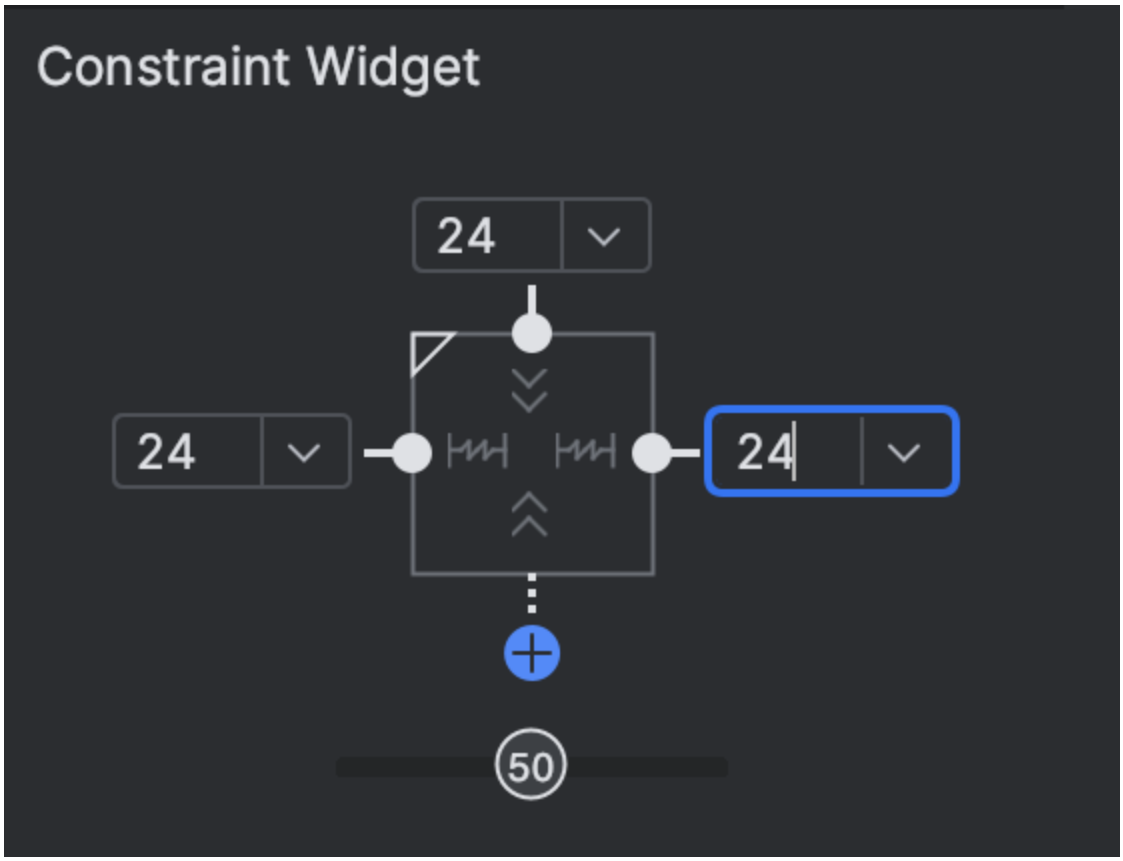
4. Go to `strings.xml`, and create a new string with name `random_heading`. Give it the value of: `Here is a random number between 0 and %d`
5. Go back to `fragment_second.xml`. Select `textview_second` in the component tree. In the attributes panel, set the id to `textview_header`, and make sure to refactor it.
6. Set the text to `@string/random_heading`.
7. Constrain the top of `textview_header` to the top of the screen, and constrain the bottom of `button_second` to the bottom of the screen.
8. Add another `TextView` from the palette and drop it near the middle of the screen. This `TextView` will be used to display a random number between 0 and the current count from the first `Fragment`.
9. Set the `id` to `@+id/textview_random` (`textview_random` in the **Attributes** panel), and refactor if you need to
10. Constrain the top edge of the new `TextView` to the bottom of the first `TextView`, the left edge to the left of the screen, and the right edge to the right of the screen, and the bottom to the top of the **Previous** button.
11. Set both width and height to **wrap_content**.

12. Set the **textColor** to `@android:color/white`, set the **textSize** to **72sp**, and the **textStyle** to **bold**.
13. Set the text to "`R`". This text is just a placeholder until the random number is generated.
14. Set the **layout_constraintVertical_bias** to **0.45**.

This `TextView` is constrained on all edges, so it's better to use a vertical bias than margins to adjust the vertical position, to help the layout look good on different screen sizes and orientations.

Updating the Header Formatting and Text

1. Select `textview_header` in the component tree
2. Set the width to **match_constraint**, but set the height to **wrap_content**, so the height will change as needed to match the height of the content
3. Set top, left and right margins to `24dp`. Left and right margins may also be referred to as "start" and "end" to support localization for right to left languages.



4. Remove any bottom constraint.
5. Set the text color to `@color/colorPrimaryDark` and the text size to `24sp`.

Change the Background Color of the Layout

Let's give your new activity a different background color than the first activity:

1. In `colors.xml`, add a new color resource:

```
<color name="screenBackground2">#26C6DA</color>
```

2. In the layout for the second activity, `fragment_second.xml`, set the background of the `ConstraintLayout` to the new color, either through the Attributes panel or in XML

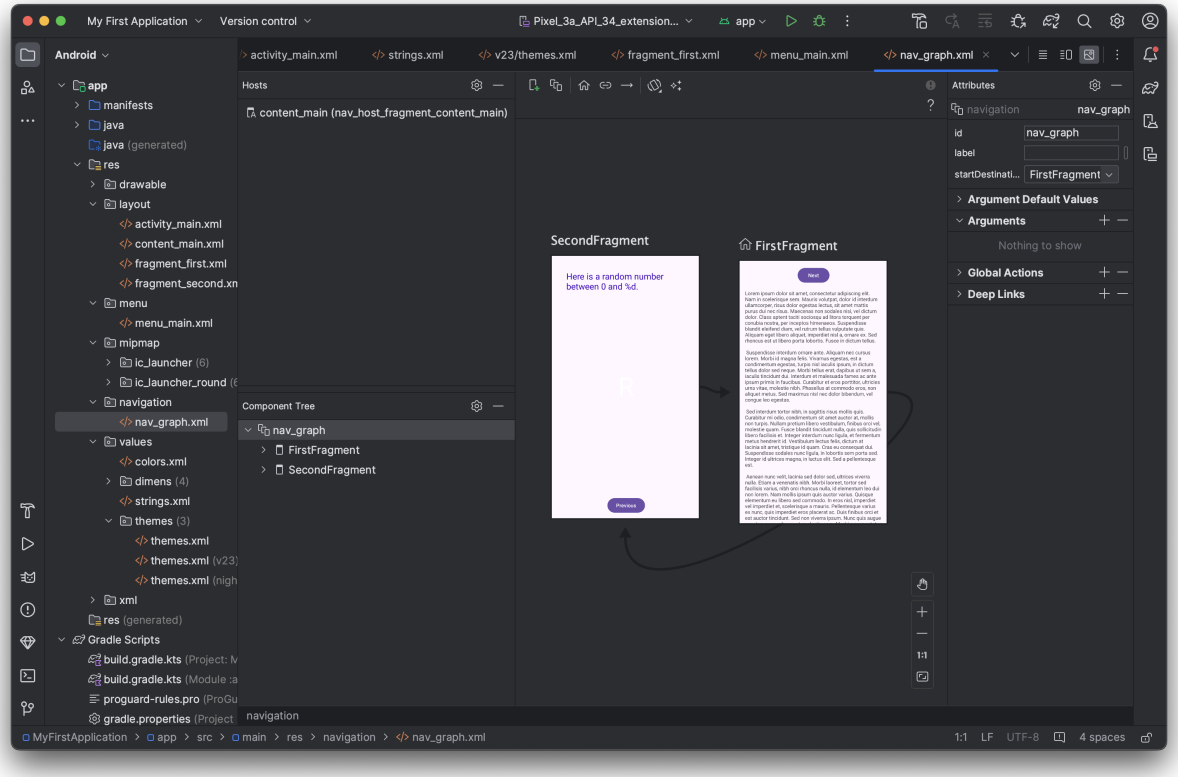
Your app now has a completed layout for the second fragment. But if you run your app and press the **Random** button, it may crash. The click handler that Android Studio set up for that button needs some changes. In the next task, you will explore and fix this error.

Examining the Navigation Graph

When you first created your project, you chose **Basic Views Activity** as the template for the new project. When Android Studio uses the **Basic Views Activity** template for a new project, it sets up two fragments, and a navigation graph to connect the two. It also set up a button to send a string argument from the first fragment to the second. This is the button you changed into the Random button. And now you want to send a number instead of a string.

1. Open `nav_graph.xml` (**app > res > navigation > nav_graph.xml**). Select the **Design View**.

A screen similar to the **Layout Editor** in **Design** view appears. It shows the two fragments with some arrows between them. You can zoom with **+** and **-** buttons in the lower right, as you did with the **Layout Editor**.



2. You can freely move the elements in the navigation editor. For example, if the fragments appear with `SecondFragment` to the left, drag `FirstFragment` to the left

of `SecondFragment` so they appear in the order you work with them.

Enable SafeArgs

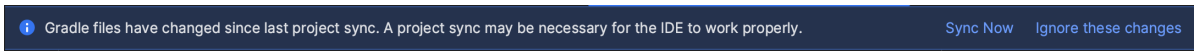
1. Expand the `GradleScripts` folder, and navigate to `build.gradle.kts (Project: My_First_Application)`. Here, paste the following code:

```
buildscript {
    repositories {
        google()
    }
    dependencies {
        val nav_version = "2.7.6"
        classpath("androidx.navigation:navigation-safe-args-grad
    }
}
```

2. Add the following lines of code to `build.gradle.kts (Module :app)`:

```
plugins {
    id("androidx.navigation.safeargs.kotlin")
}
```

3. Android Studio should display a message about the Gradle files being changed. Click **Sync Now** on the notification that appears at the top.

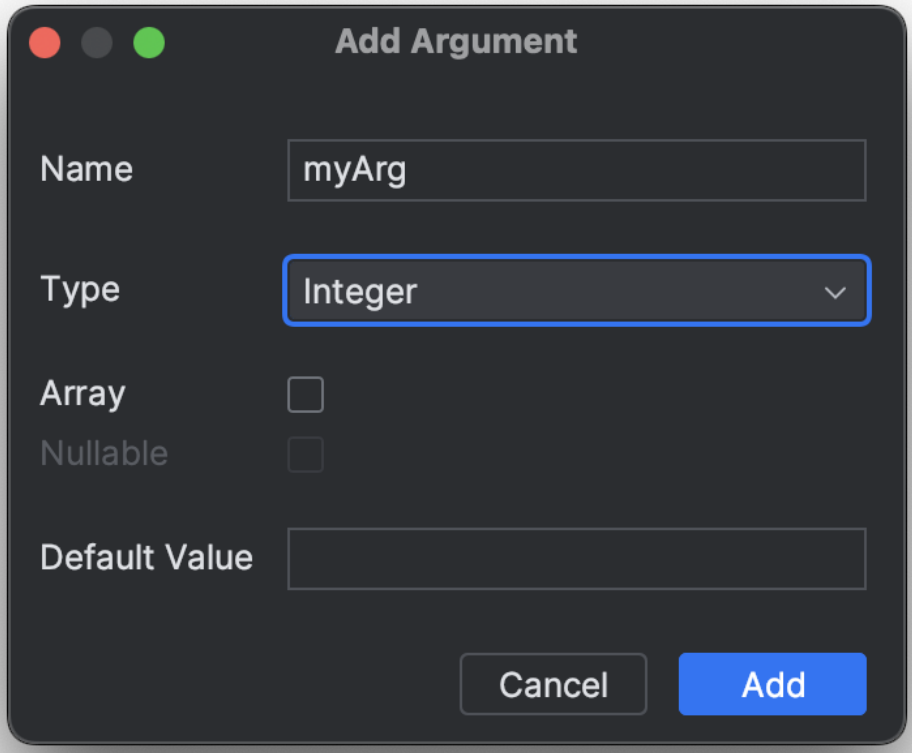


4. Choose **Build > Make Project**. This should rebuild everything so that Android Studio can find `FirstFragmentDirections`.

Create the Argument for the Navigation Action

1. In the navigation graph, click on `FirstFragment`, and look at the **Attributes** panel to the right. (If the panel isn't showing, click on the vertical **Attributes** label to the right.)

2. In the **Actions** section, it shows what action will happen for navigation, namely going to `SecondFragment`.
3. Click on `SecondFragment`, and look at the **Attributes** panel. The **Arguments** section shows `Nothing to show`.
4. Click on the **+** in the **Arguments** section.
5. In the **Add Argument** dialog, enter `myArg` for the name and set the type to **Integer**, then click the **Add** button.



Send the Count to the Second Fragment

The **Next/Random** button was set up by Android Studio to go from the first fragment to the second, but it doesn't send any information. In this step you'll change it to send a

number for the current count. You will get the current count from the text view that displays it, and pass that to the second fragment.

1. Open `FirstFragment.java` (**app > java > com.example.myfirstapp > FirstFragment**)
2. Find the method `onViewCreated()` and notice the code that sets up the click listener to go from the first fragment to the second.
3. Replace the code in that click listener with a line to find the count text view, `textview_first`:

```
int currentCount = Integer.parseInt(showCountTextView.getText()
```

4. Create an action with `currentCount` as the argument to `actionFirstFragmentToSecondFragment()`:

```
FirstFragmentDirections.ActionFirstFragmentToSecondFragment acti
```

5. Add a line to find the nav controller and navigate with the action you created:

```
NavHostFragment.findNavController(FirstFragment.this).navigate(a
```

Your `onViewCreated` method should look like the following:

```
public void onViewCreated(@NonNull View view, Bundle savedInstan
        super.onViewCreated(view, savedInstanceState);

        binding.randomButton.setOnClickListener(new View.OnClicl
            @Override
            public void onClick(View view) {
                int currentCount = Integer.parseInt(showCountTex
                FirstFragmentDirections.ActionFirstFragmentToSec
                NavHostFragment.findNavController(FirstFragment
                        .navigate(action);
            }
        });
```

```
        view.findViewById(R.id.toast_button).setOnClickListener(
            @Override
            public void onClick(View v) {
                Toast toast = Toast.makeText(getActivity(), "Hel
                toast.show();
            }
        });

        view.findViewById(R.id.count_button).setOnClickListener(
            @Override
            public void onClick(View view) {
                countMe(view);
            }
        });
    }
```

Now, run your app. Click the **Count** button a few times. Now when you press the **Random** button, the second screen shows the correct string in the header, but still no count or random number, because you need to write some code to do that.

Update SecondFragment.java to Compute/Display Random Number

You have written the code to send the current count to the second fragment. The next step is to add code to `SecondFragment.java` to retrieve and use the current count.

1. In the `onViewCreated()` method below the line that starts with `super`, add code to get the current count, get the string and format it with the count, and then set it for textview_header.

```
Integer count = SecondFragmentArgs.fromBundle(getArguments()).ge
String countText = getString(R.string.random_heading, count);
```

```
TextView headerView = view.getRootView().findViewById(R.id.textv
headerView.setText(countText);
```

2. Get a random number between 0 and the count:

```
Random random = new java.util.Random();
Integer randomNumber = 0;
if (count > 0) {
   randomNumber = random.nextInt(count + 1);
}
```

3. Add code to convert that number into a string and set it as the text for `textview_random`:

```
TextView randomView = view.getRootView().findViewById(R.id.textv
randomView.setText(randomNumber.toString());
```

4. Run the app. Press the **Count** button a few times, then press the **Random** button.

Congratulations!

You have successfully built your first Android app! For more resources/information on Android Studio, visit this link for more information.