

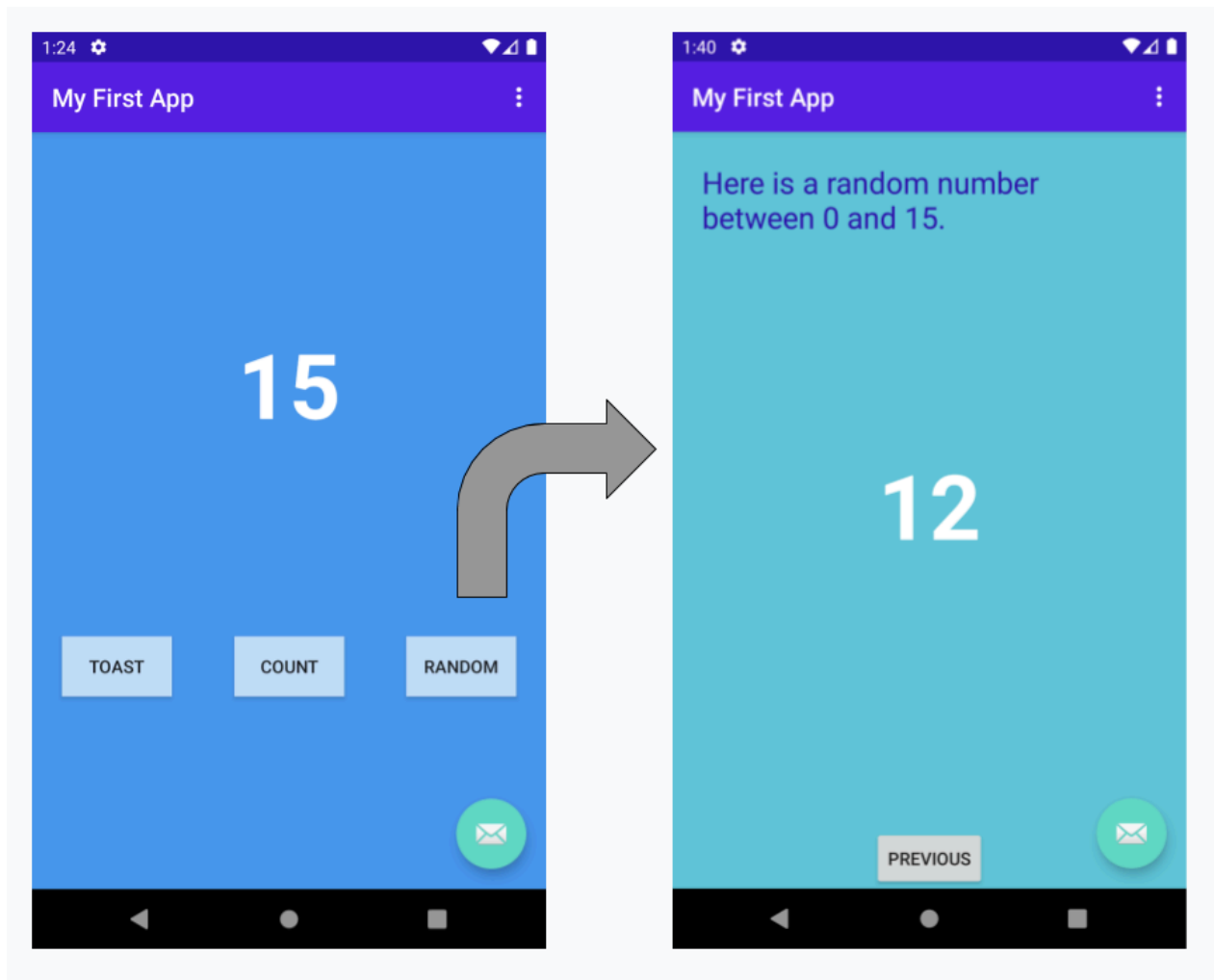
Source file:

<https://web.archive.org/web/20240324083233/https://developer.android.com/codelabs/build-your-first-android-app?hl=en#0>

Build Your First Android App in Java

This document has 10 sections.

1. Welcome



In this codelab, you'll learn how to build and run your first Android app in the Java programming language. (If you're looking for the [Kotlin version of this codelab](#), you can go [here](#).)

What you must know already

This codelab is written for programmers and assumes that you know either the Java or Kotlin programming language. If you are an experienced programmer and adept at reading code, you will likely be able to follow this codelab, even if you don't have much experience with Java.

What you'll learn

- How to use Android Studio to build your app.
- How to run your app on a device or in the emulator.
- How to add interactive buttons.
- How to display a second screen when a button is pressed.

Use Android Studio and Java to write Android apps

You write Android apps in the Java programming language using an IDE called Android Studio. Based on JetBrains' IntelliJ IDEA software, Android Studio is an IDE designed specifically for Android development.

Note: This version of the codelab requires Android Studio 3.6 or higher.

To work through this codelab, you will need a computer that can run [Android Studio 3.6](#) or higher (or already has Android Studio 3.6 or higher installed).

2. Install Android Studio

Note: This version of the codelab requires Android Studio 3.6 or higher.

You can download the latest Android Studio from the [Android Studio](#) page.

Android Studio provides a complete IDE, including an advanced code editor and app templates. It also contains tools for development, debugging, testing, and performance that make it faster and easier to develop apps. You can use Android Studio to test your apps with a large range of preconfigured emulators, or on your own mobile device. You can also build production apps and publish apps on the Google Play store.

Note: Android Studio is continually being improved. For the latest information on system requirements and installation instructions, see the [Android Studio download page](#).

Android Studio is available for computers running Windows or Linux, and for Macs running macOS. The OpenJDK (Java Development Kit) is bundled with Android Studio.

The installation is similar for all platforms. Any differences are noted below.

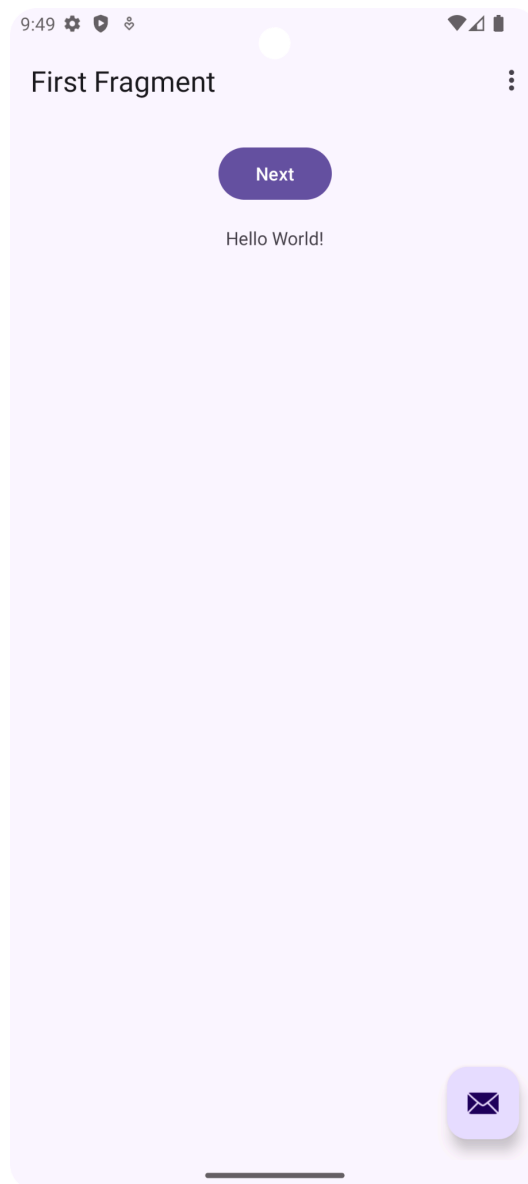
1. Navigate to the [Android Studio download page](#) and follow the instructions to download and [install Android Studio](#).
2. Accept the default configurations for all steps, and ensure that all components are selected for installation.
3. After the install is complete, the setup wizard downloads and installs additional components, including the Android SDK. Be patient, because this process might take some time, depending on your internet speed.
4. When the installation completes, Android Studio starts, and you are ready to create your first project.

Troubleshooting: If you run into problems with your installation, see the [Android Studio release notes](#) or [Troubleshoot Android Studio](#).

3. Create your first Project

In this step, you will create a new Android project for your first app. This simple app displays the string "Hello World" on the screen of an Android virtual or physical device.

Here's what the finished app will look like:

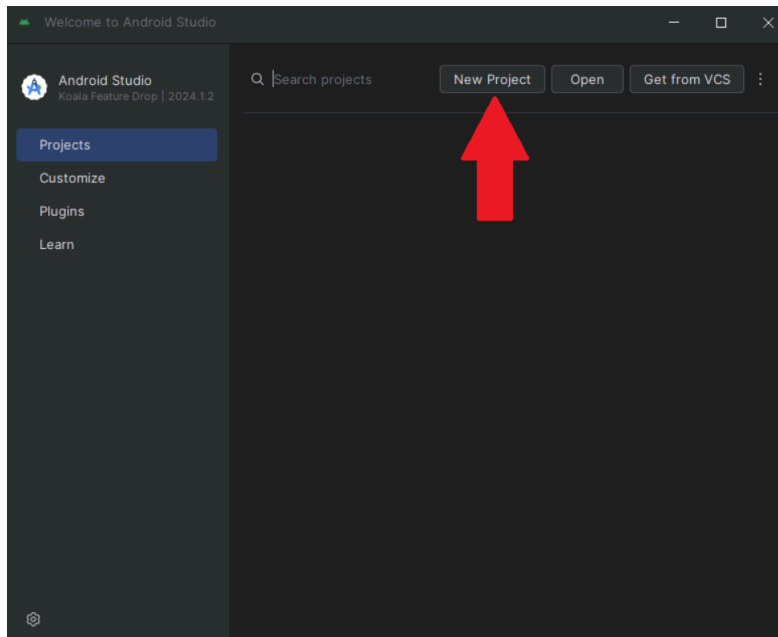


What you'll learn

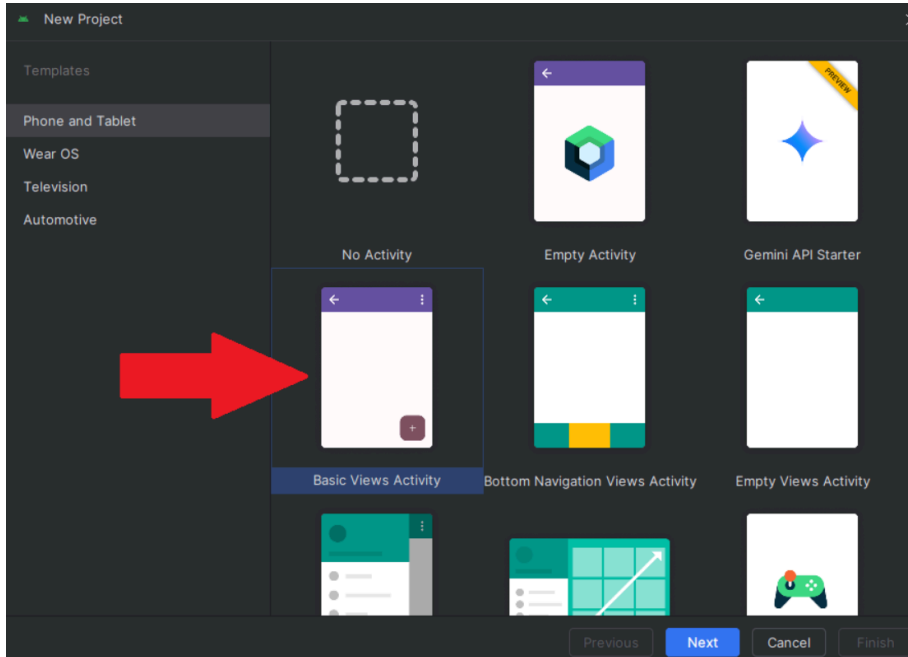
- How to create a project in Android Studio.
- How to create an emulated Android device.
- How to run your app on the emulator.
- How to run your app on your own physical device, if you have one.

Step 1: Create a new project

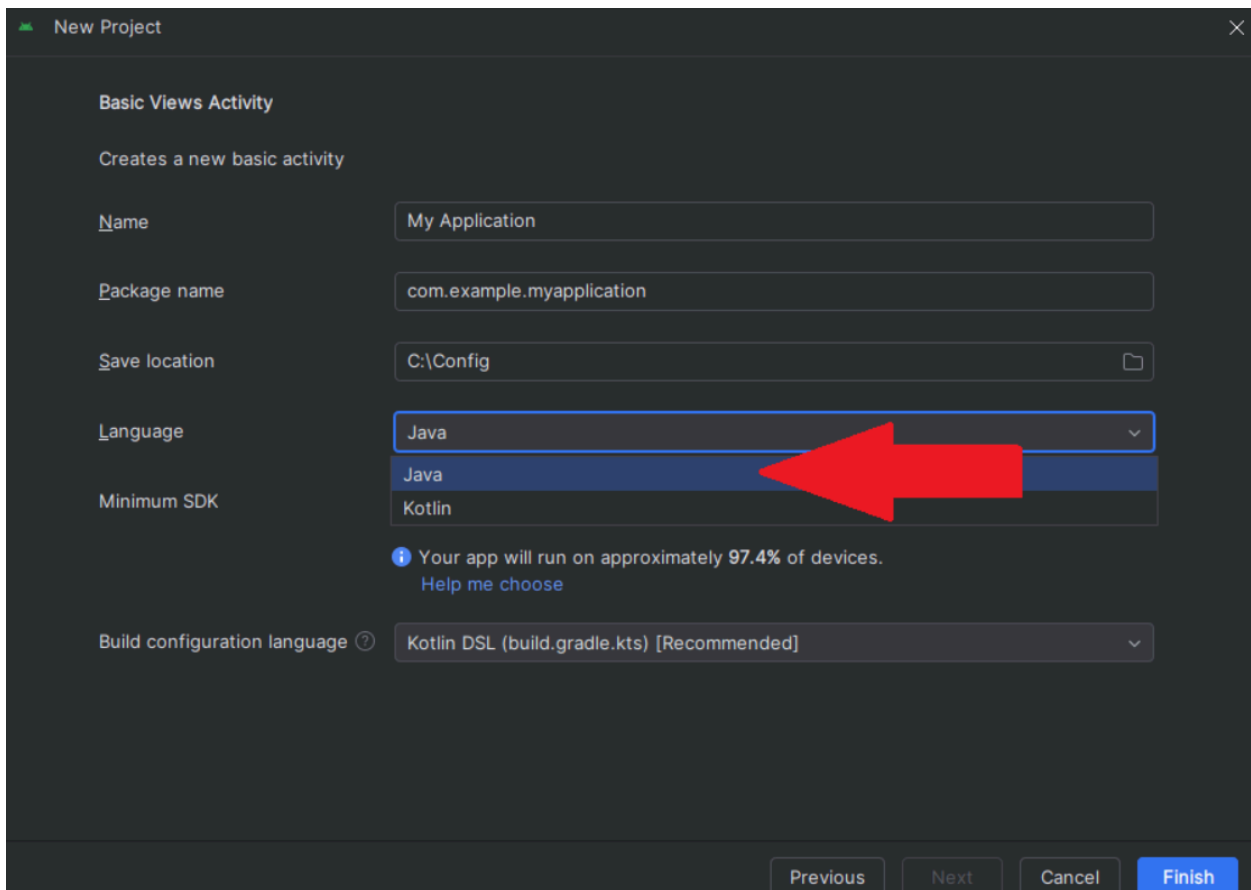
1. Open Android Studio.
2. In the **Welcome to Android Studio** dialog, click **New Project**.



3. Select **Basic Views Activity** (not the default). Click **Next**.



4. Give your application a **Name** such as **My First App**.
5. Make sure the **Language** is set to **Java**.



6. Leave the defaults for the other fields.

7. Click **Finish**.

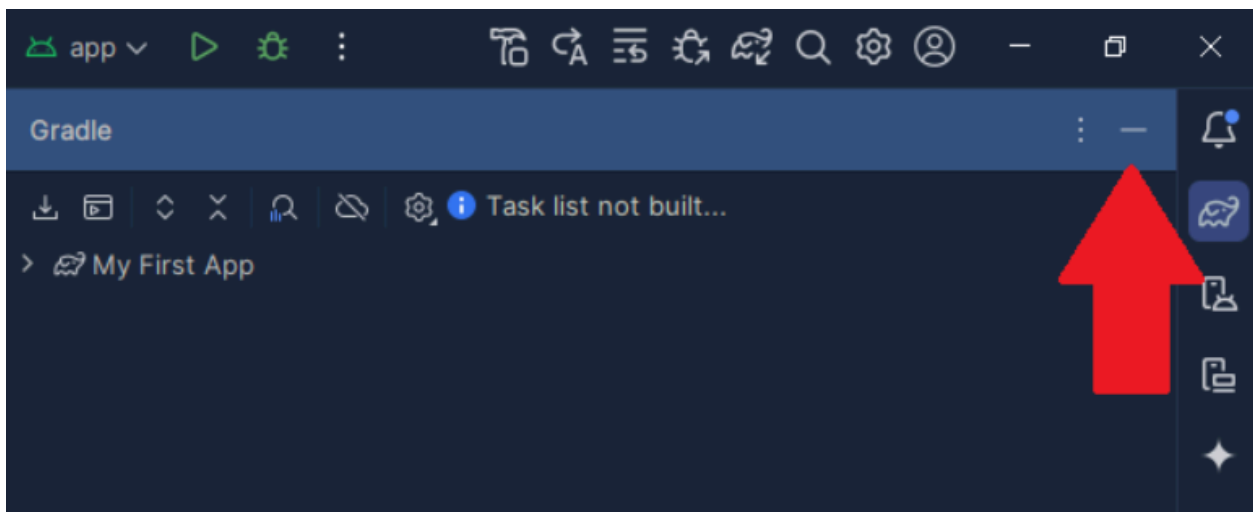
After these steps, Android Studio:

- Creates a folder for your Android Studio project called **My First App**. This is usually in a folder called **AndroidStudioProjects** below your home directory.
- Builds your project (this may take a few moments). Android Studio uses [Gradle](#) as its build system. You can follow the build progress at the bottom of the Android Studio window.
- Opens the code editor showing your project.

Step 2: Get your screen set up

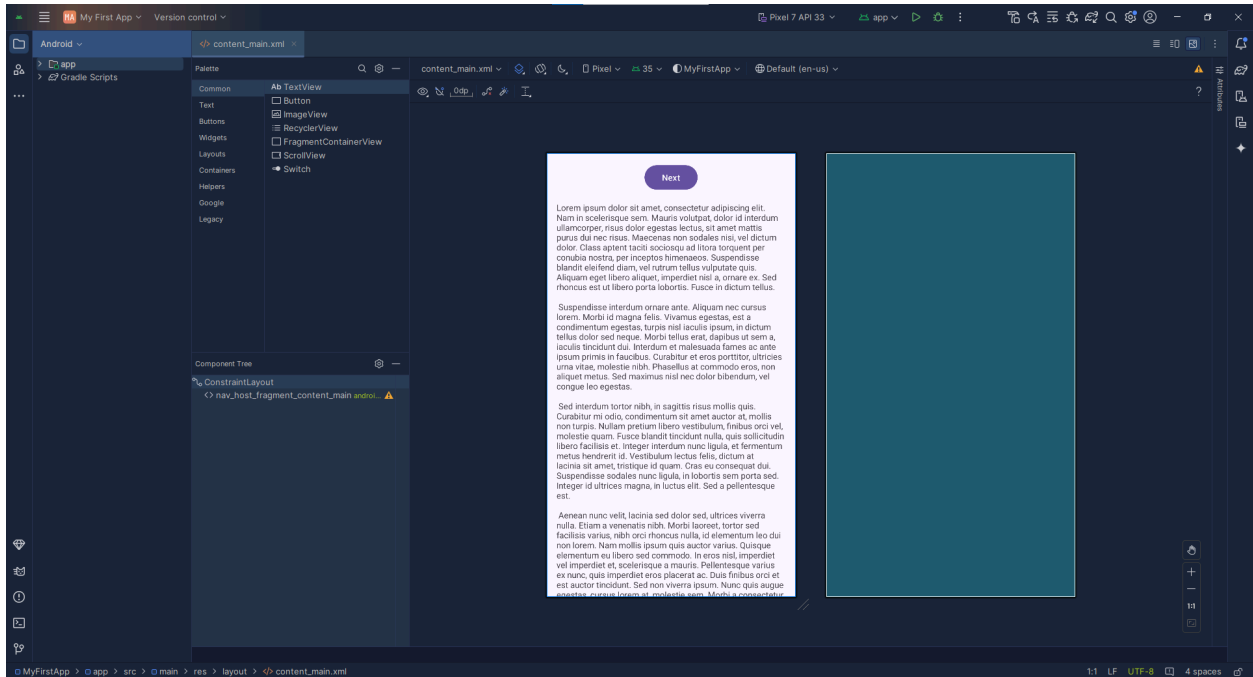
When your project first opens in Android Studio, there may be a lot of windows and panes open. To make it easier to get to know Android Studio, here are some suggestions on how to customize the layout.

1. If there's a **Gradle** window open on the right side, click on the minimize button (—) in the upper right corner to hide it.



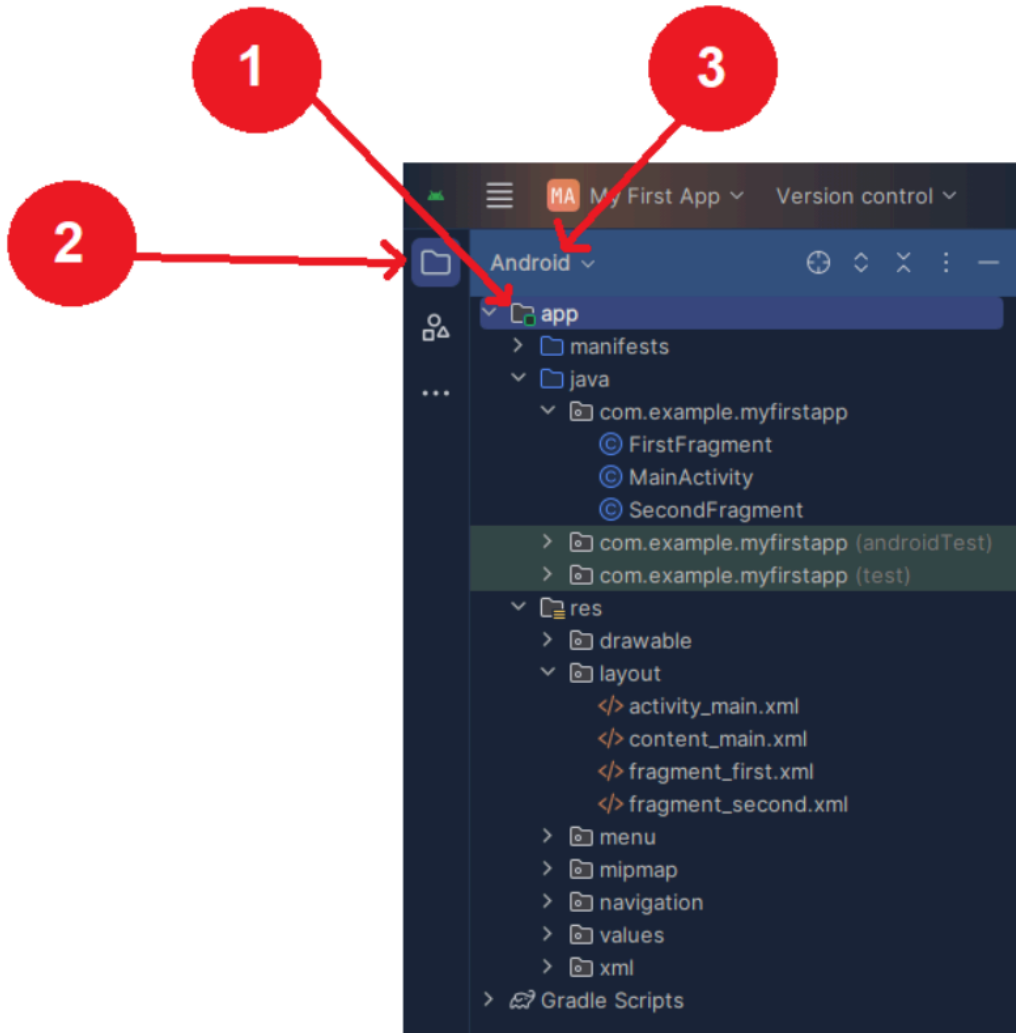
2. Depending on the size of your screen, consider resizing the pane on the left showing the project folders to take up less space.

At this point, your screen should look a bit less cluttered, similar to the screenshot shown below.



Step 3: Explore the project structure and layout

The upper left of the Android Studio window should look similar to the following diagram:



Based on you selecting the **Basic Views Activity** template for your project, Android Studio has set up a number of files for you. You can look at the hierarchy of the files for your app in multiple ways, one is in **Project** view. **Project** view shows your files and folders structured in a way that is convenient for working with an Android project. (This does not always match the file hierarchy! To see the file hierarchy, choose the **Project files** view by clicking **(3)**.)

1. Double-click the **app (1)** folder to expand the hierarchy of app files. (See **(1)** in the screenshot.)
2. If you click **Project (2)**, you can hide or show the **Project** view. You might need to select **View > Tool Windows** to see this option.
3. The current **Project** view selection (3) is **Project > Android**.

In the **Project > Android** view you see three or four top-level folders below your **app** folder: **manifests**, **java**, **java (generated)** and **res**. You may not see **java (generated)** right away.

1. Expand the **manifests** folder. This folder contains **AndroidManifest.xml**. This file describes all the components of your Android app and is read by the Android runtime system when your app is executed.
2. Expand the **java** folder. All your Java language files are organized here. The java folder contains three subfolders:

com.example.myfirstapp: This folder contains the Java source code files for your app.

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.

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.

3. Expand 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.

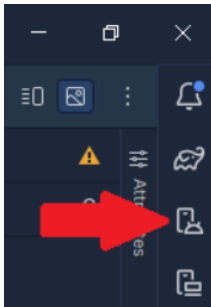
values: This folder contains resources, such as strings and colors, used in your app.

Step 4: Create a virtual device (emulator)

In this task, you will use the [Android Virtual Device \(AVD\)](#) manager to create a virtual device (or emulator) that simulates the configuration for a particular type of Android device.

The first step is to create a configuration that describes the virtual device.

1. In Android Studio, select **Tools > Device Manager**, or click the Device Manager icon in the toolbar.



2. Click the **(+) button > Create Virtual Device**. (If you have created a virtual device before, the window shows all of your existing devices). The **Select Hardware** window shows a list of pre-configured hardware device definitions.
3. Choose a device definition, such as **Pixel 2**, and click **Next**. (For this codelab, it really doesn't matter which device definition you pick).
4. In the **System Image** dialog, from the **Recommended** tab, choose the latest release. (This does matter.)
5. If a **Download** link is visible next to a latest release, it is not installed yet, and you need to download it first. If necessary, click the link to start the download, and click **Next** when it's done. This may take a while depending on your connection speed.

Note: System images can take up a large amount of disk space, so just download what you need.

6. In the next dialog box, accept the defaults, and click **Finish**.

The Device Manager now shows the virtual device you added.

7. If your **Device Manager** window is still open, go ahead and close it.

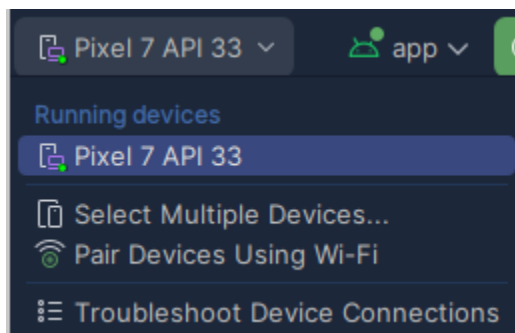
Step 5: Run your app on your new emulator

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.



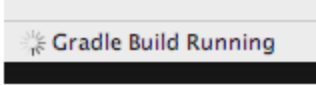
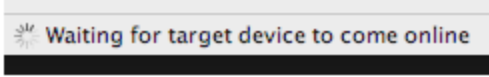
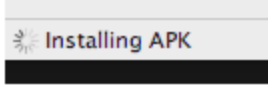
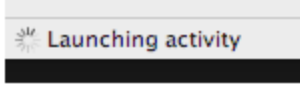
If you get a dialog box stating "Instant Run requires that the platform corresponding to your target device (Android N...) is installed" go ahead and click Install and continue.

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

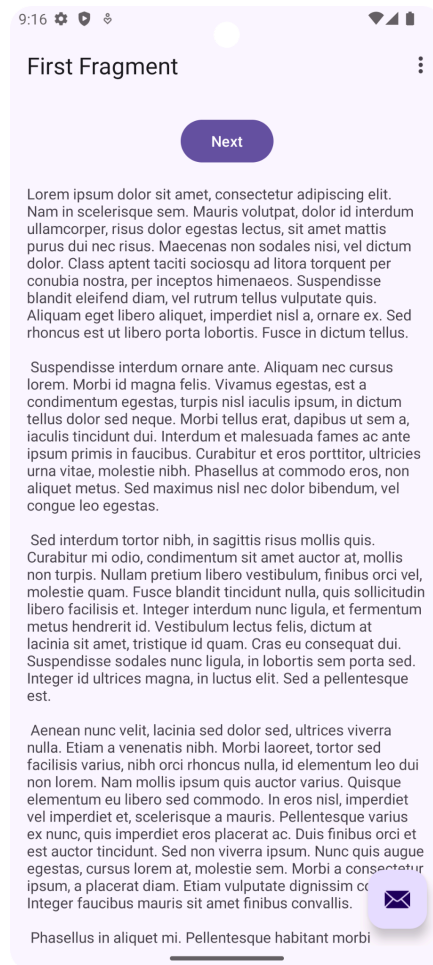


The emulator starts and boots just like a physical device. Depending on the speed of your computer, this may take a while. You can look in the small horizontal status bar at the very bottom of Android Studio for messages to see the progress.

Messages that might appear briefly in the status bar

<i>Gradle build running</i>	
<i>Waiting for target device to come on line</i>	
<i>Installing APK</i>	
<i>Launching activity</i>	

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.



Note: It is a good practice to start the emulator at the beginning of your session. Don't close the emulator until you are done testing your app, so that you don't have to wait for the emulator to boot again. Also, don't have more than one emulator running at once, to reduce memory usage.

Step 6: Run your app on a device (if you have one)

What you need:

- An Android device such as a phone or tablet.
- A data cable to connect your Android device to your computer via the USB port.
- If you are using a Linux or Windows OS, you may need to perform additional steps to run your app on a hardware device. Check the [Run Apps on a Hardware Device](#) documentation. On Windows, you may need to install the appropriate USB driver for your device. See [OEM USB Drivers](#).

Run your app on a device

To let Android Studio communicate with your device, you must turn on USB Debugging on your Android device.

On Android 4.2 and higher, the Developer options screen is hidden by default. To show Developer options and enable USB Debugging:

1. On your device, open **Settings > About phone** and tap **Build number** seven times.
2. Return to the previous screen (**Settings**). **Developer options** appears at the bottom of the list. Tap **Developer options**.
3. Enable **USB Debugging**.

Now you can connect your device and run the app from Android Studio.

1. Connect your device to your development machine with a USB cable. On the device, you might need to agree to allow USB debugging from your development device.

2. In Android Studio, click **Run (Green arrow icon)** in the toolbar at the top of the window. (You might need to select **View > Toolbar** to see this option.) The **Select Deployment Target** dialog opens with the list of available emulators and connected devices.
3. Select your device, and click **OK**. Android Studio installs the app on your device and runs it.

Note: If your device is running an Android platform that isn't installed in Android Studio, you might see a message asking if you want to install the needed platform. Click **Install and Continue**, then click **Finish** when the process is complete.

Troubleshooting

If you're stuck, quit Android Studio and restart it.

If Android Studio does not recognize your device, try the following:

1. Disconnect your device from your development machine and reconnect it.
2. Restart Android Studio.

If your computer still does not find the device or declares it "unauthorized":

1. Disconnect the device.
2. On the device, open **Settings->Developer Options**.
3. Tap **Revoke USB Debugging authorizations**.
4. Reconnect the device to your computer.
5. When prompted, grant authorizations.

If you are still having trouble, check that you installed the appropriate USB driver for your device. See the [Using Hardware Devices documentation](#).

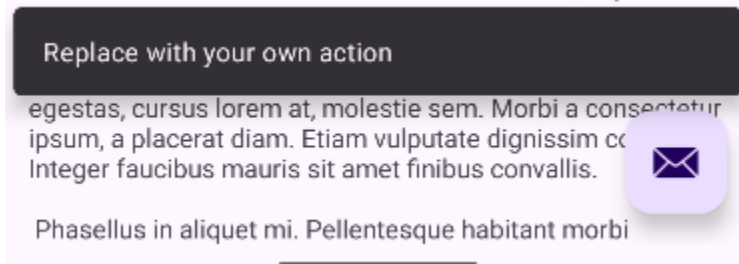
Check the [troubleshooting section in the Android Studio documentation](#).


Step 7: Explore the app template

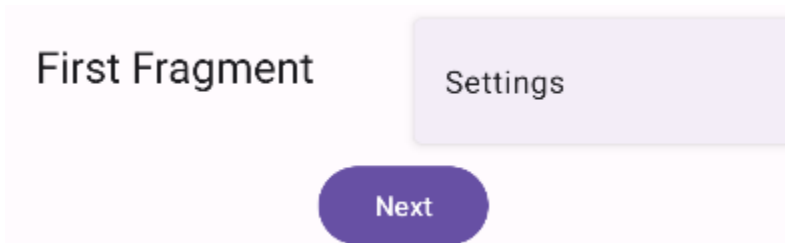
When you created the project and selected **Basic Views Activity**, Android Studio set up a number of files, folders, and also user interface elements

for you, so you can start out with a working app and major components in place. This makes it easier to build your application.

Looking at your app on the emulator or your device, in addition to the **Next** button, notice the *floating action button* with an email/mail icon. If you tap that button, you'll see it has been set up to briefly show a message at the bottom of the screen. This message space is called a *snackbar*, and it's one of several ways to notify users of your app with brief information.



At the top right of the screen, there's a menu with 3 vertical dots . If you tap on that, you'll see that Android Studio has also created an options menu with a **Settings** item. Choosing **Settings** doesn't do anything yet, but having it set up for you makes it easier to add user-configurable settings to your app.



Later in this codelab, you'll look at the **Next** button and modify the way it looks and what it does.

4. Explore the layout editor

Generally, each screen in your Android app is associated with one or more [fragments](#). The single screen displaying "Hello first fragment" is created by one fragment, called `FirstFragment`. This was generated for you when you created your new project. Each visible fragment in an Android app has a layout that defines the user interface for the fragment. Android Studio has a layout editor where you can create and define layouts.

Layouts are defined in [XML](#). The layout editor lets you define and modify your layout either by coding XML or by using the interactive visual editor.

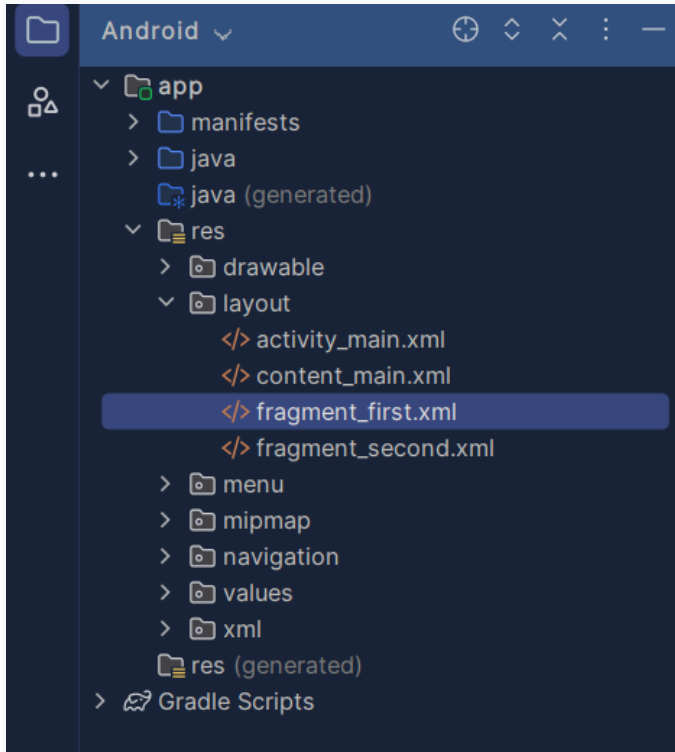
Every element in a layout is a view. In this task, you will explore some of the panels in the layout editor, and you will learn how to change property of views.

What you'll learn

- How to use the layout editor.
- How to set property values.
- How to add string resources.
- How to add color resources.

Step 1: Open the layout editor

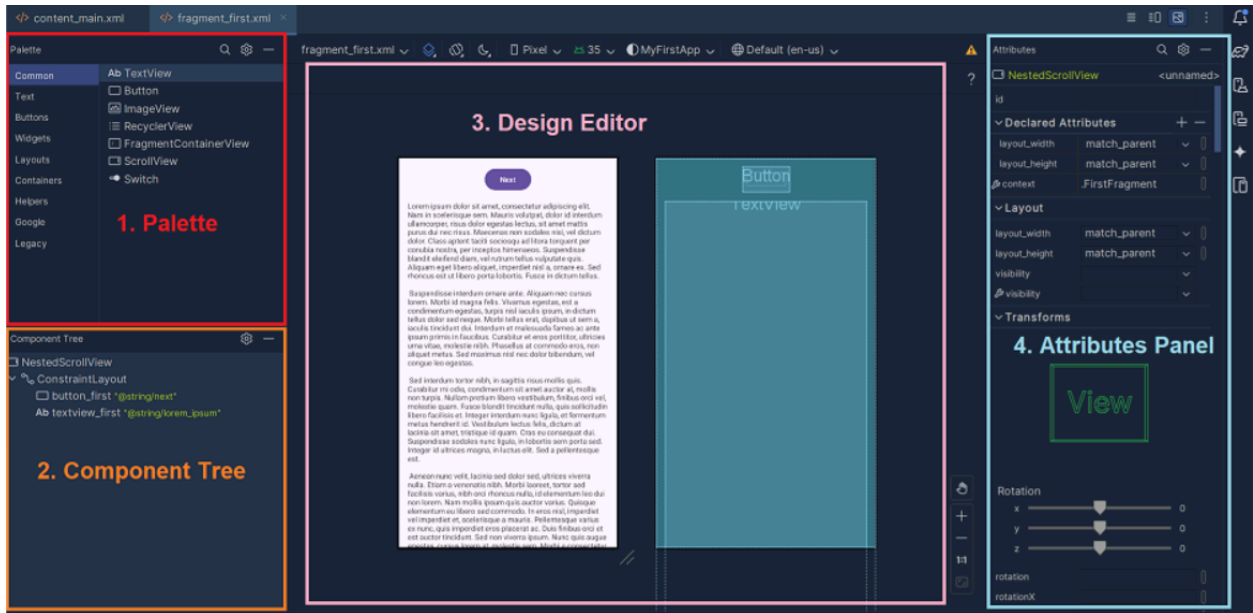
1. Find and open the layout folder (app > res > layout) on the left side in the Project



2. Double-click `fragment_first.xml`.

Troubleshooting: If you don't see the file `fragment_first.xml`, confirm you are running Android Studio 3.6 or later, which is required for this codelab.

The panels to the right of the Project view comprise the *Layout Editor*. They may be arranged differently in your version of Android Studio, but the function is the same.

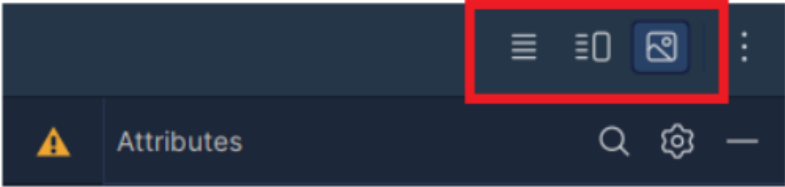


On the left is a **Palette (1)** of views you can add to your app.

Below that is a **Component Tree (2)** showing the views currently in this file, and how they are arranged in relation to each other.

In the center is the *Design editor (3)*, which shows a visual representation of what the contents of the file will look like when compiled into an Android app. You can view the visual representation, the XML code, or both.

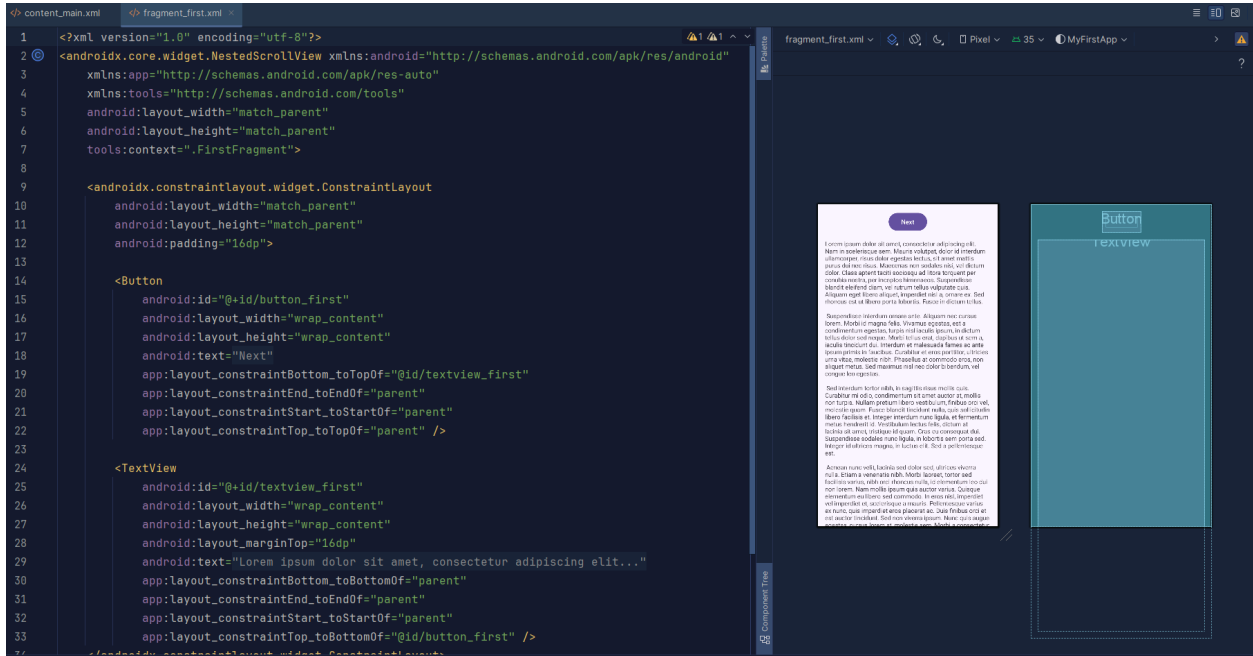
3. In the upper right corner of the Design editor, above **Attributes (4)**, find the three icons that look like this:



These represent **Code** (code only), **Split** (code + design), and **Design** (design only) views.

4. Try selecting the different modes. Depending on your screen size and work style, you may prefer switching between **Code** and **Design**, or staying in **Split** view. If your **Component Tree** disappears, hide and show the **Palette**.

Split view:

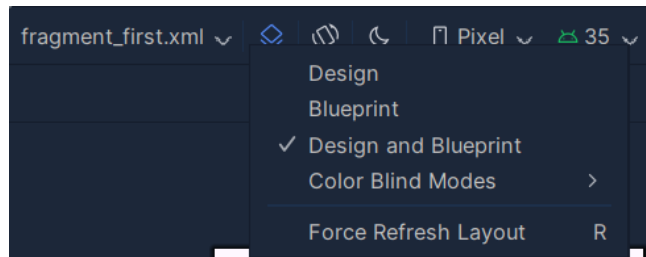


5. At the lower right of the Design editor you see + and - buttons for zooming in and out. Use these buttons to adjust the size of what you see, or click the zoom-to-fit button so that both panels fit on your screen.



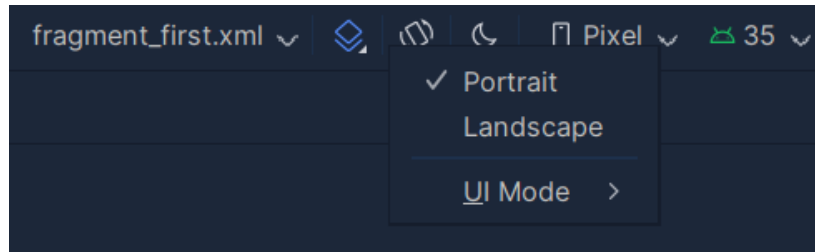
The *Design layout* on the left shows how your app appears on the device. The *Blueprint layout*, shown on the right, is a schematic view of the layout.

6. Practice using the layout menu in the top left of the design toolbar to

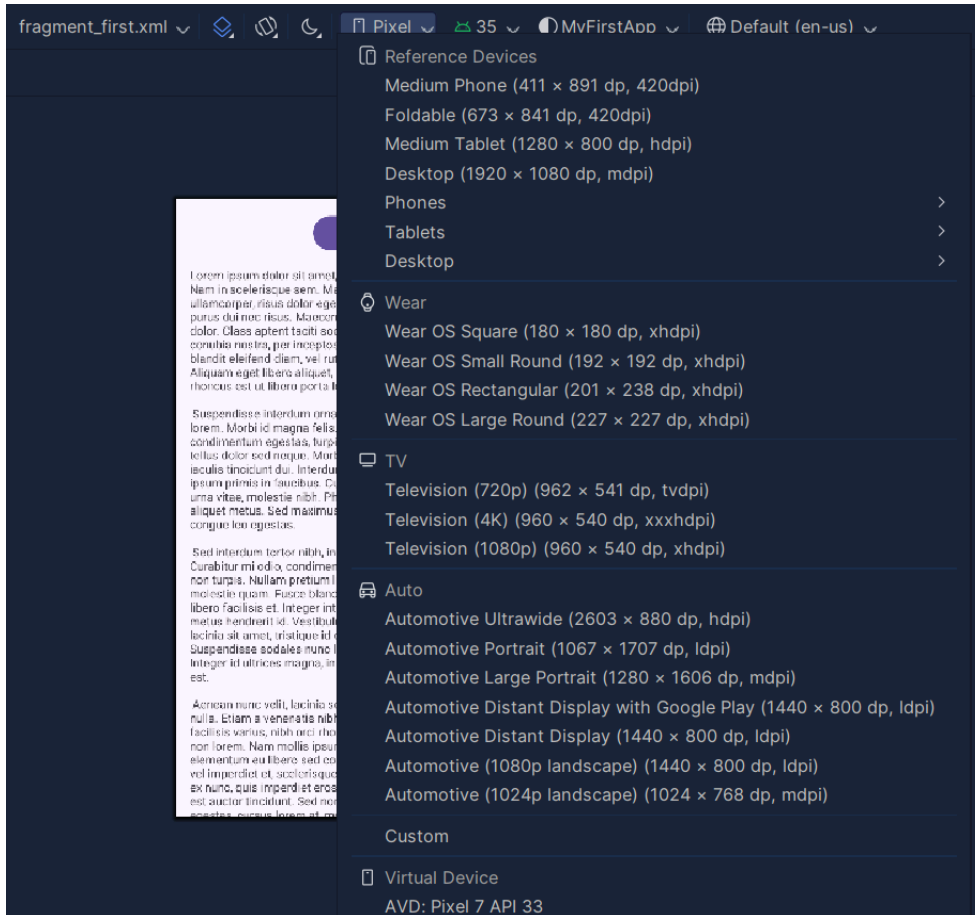


display the design view, the blueprint view, and both views side by side. Depending on the size of your screen and your preference, you may wish to only show the **Design** view or the **Blueprint** view, instead of both.

7. Use the orientation icon to change the orientation of the layout. This allows you to test how your layout will fit portrait and landscape modes.



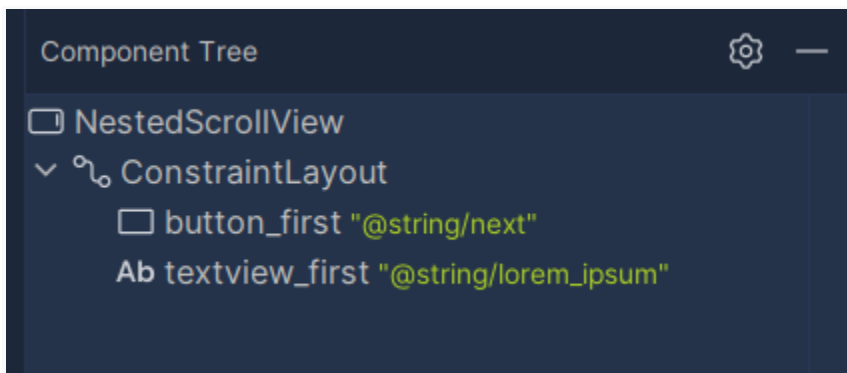
8. Use the device menu to view the layout on different devices. (This is extremely useful for testing!)



On the right is the **Attributes** panel. You'll learn about that later.

Step 2: Explore and resize the Component Tree

1. In `fragment_first.xml`, look at the **Component Tree**. If it's not showing, switch the mode to **Design** instead of **Split** or **Code**.



This panel shows the *view hierarchy* in your layout, that is, how the views are arranged in relation to each other.

2. If necessary, resize the **Component Tree** so you can read at least part of the strings.

3. Click the **Hide** icon at the top right of the **Component Tree**.

The **Component Tree** closes.

4. Bring back the **Component Tree** by clicking the vertical label **Component Tree** on the left.

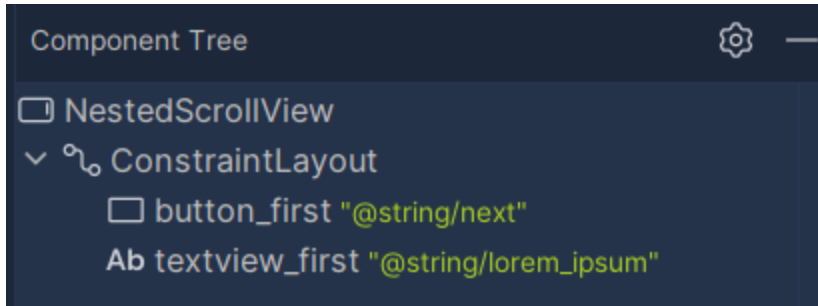


Step 3: Explore view hierarchies

1. In the **Component Tree**, notice that the root of the view hierarchy is a **ConstraintLayout** view.

Every layout must have a root view that contains all the other views. The root view is always a *view group*, which is a view that contains other views. A **ConstraintLayout** is one example of a view group.

2. Notice that the **ConstraintLayout** contains a **TextView**, called **textview_first** and a **Button**, called **button_first**.



3. If the code isn't showing, switch to **Code** or **Split** view using the icons in the upper right corner.
4. In the XML code, notice that the root element is `<androidx.constraintlayout.widget.ConstraintLayout>`. The root element contains a `<TextView>` element and a `<Button>` element.

```
<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"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/button_first" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step 4: Change property values

1. In the code editor, examine the properties in 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 elit..."
```

2. Click on the string in the text property, and you'll notice it refers to a string resource, `lorem_ipsum`.

```
android:text="@string/lorem_ipsum"
```

3. Right-click on the property and click **Go To > Declaration or Usages**, `values/strings.xml` opens with the string highlighted.

```

<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
    est.\n\n
    Aenean nunc velit, lacinia sed dolor sed, ultrices viverra nulla. Etiam a venenatis nibh.
    Morbi laoreet, tortor sed facilisis varius, nibh orci rhoncus nulla, id elementum leo dui
    non lorem. Nam mollis ipsum quis auctor varius. Quisque elementum eu libero sed commodo. In
    eros nisl, imperdiet vel imperdiet et, scelerisque a mauris. Pellentesque varius ex nunc,
    quis imperdiet eros placerat ac. Duis finibus orci et est auctor tincidunt. Sed non viverra
    ipsum. Nunc quis augue egestas, cursus lorem at, molestie sem. Morbi a consectetur ipsum, a
    placerat diam. Etiam vulputate dignissim convallis. Integer faucibus mauris sit amet finibus
    convallis.\n\n
    Phasellus in aliquet mi. Pellentesque habitant morbi tristique senectus et netus et
    malesuada fames ac turpis egestas. In volutpat arcu ut felis sagittis, in finibus massa
    gravida. Pellentesque id tellus orci. Integer dictum, lorem sed efficitur ullamcorper,
    libero justo consectetur ipsum, in mollis nisl ex sed nisl. Donec maximus ullamcorper
    sodales. Praesent bibendum rhoncus tellus nec feugiat. In a ornare nulla. Donec rhoncus
    libero vel nunc consequat, quis tincidunt nisl eleifend. Cras bibendum enim a justo luctus
    vestibulum. Fusce dictum libero quis erat maximus, vitae volutpat diam dignissim.
</string>

```

4. Change the value of the string property to Hello World!

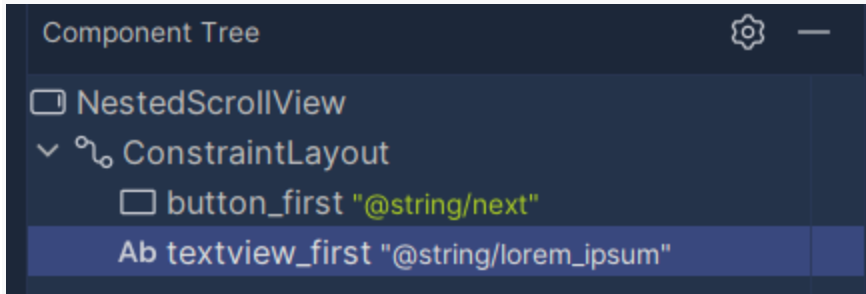
```

<string name="lorem_ipsum">
    Hello World!
</string>

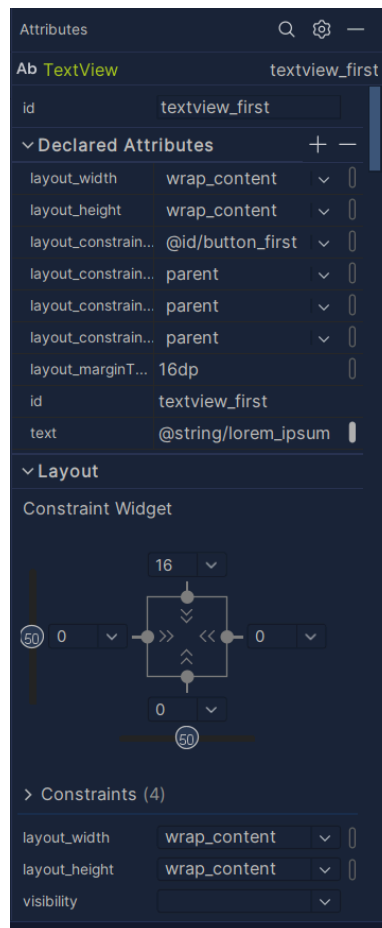
```

5. Switch back to fragment_first.xml.

6. Select textView_first in the Component Tree.

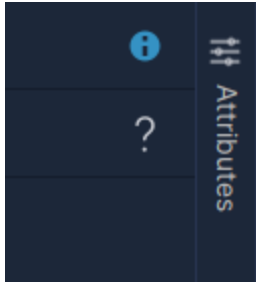


7. Look at the **Attributes** panel on the right, and open the **Declared Attributes** section if needed.

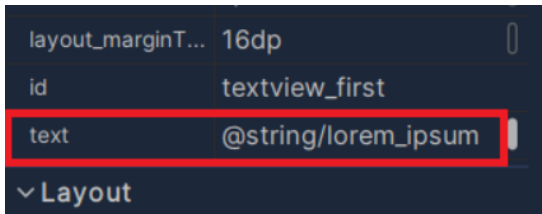


Troubleshooting this step:

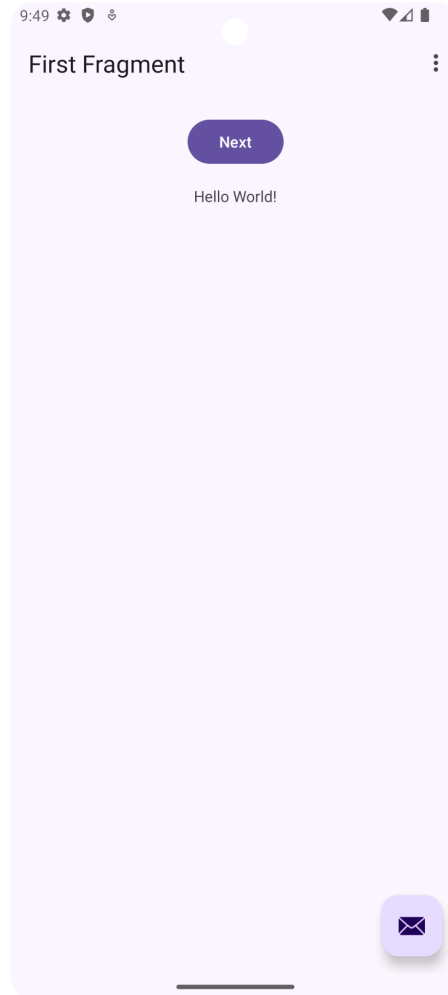
- If the Attributes panel is not visible, click the vertical Attributes label at the top right.



8. In the **text** field of the **TextView** in **Attributes**, notice it still refers to the string resource **@string/lorem_ipsum**. Having the strings in a resource file has several advantages. You can change the value of string without having to change any other code. This simplifies translating your app to other languages, because your translators don't have to know anything about the app code.

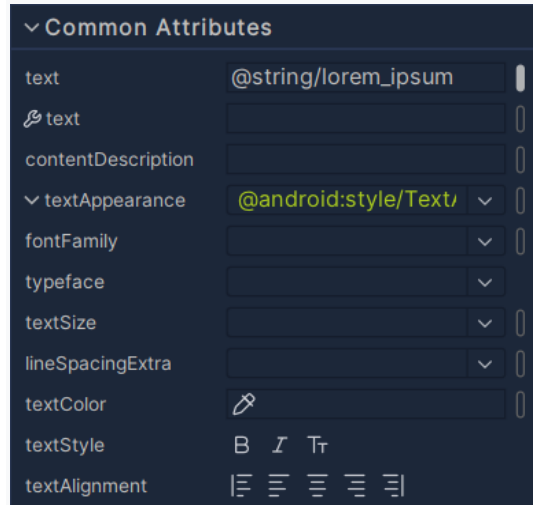


9. Run the app to see the change you made in **strings.xml**. Your app now shows "Hello World!".



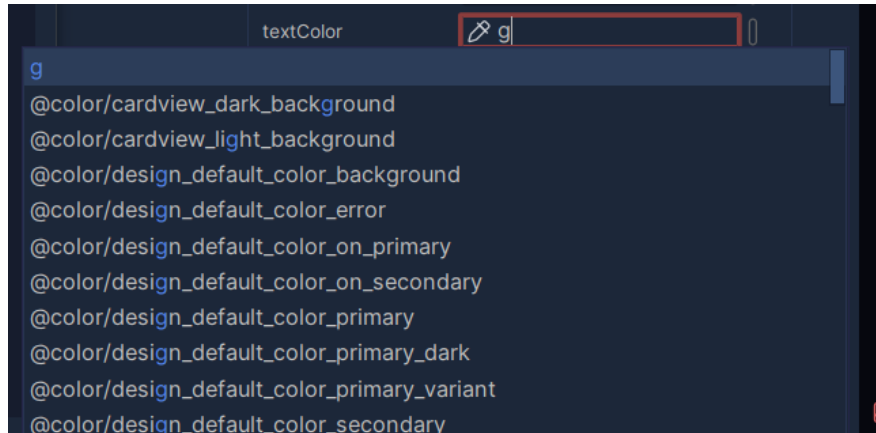
Step 5: Change text display properties

1. With `textview_first` still selected in the **Component Tree**, in the layout editor, in the list of attributes, under **Common Attributes**, expand the **textAppearance** field. (You may need to scroll down to find it.)



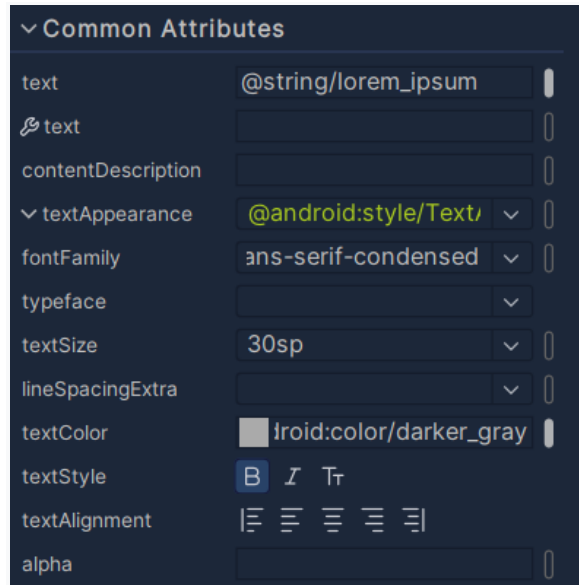
2. Change some of the text appearance properties. For example, change the font family, increase the text size, and select bold style. (You might need to scroll the panel to see all the fields.)
3. Change the text color. Click in the **textColor** field, and enter **g**.

A menu pops up with possible completion values containing the letter g. This list includes predefined colors.



4. Select **@android:color/darker_gray** and press **Enter**.

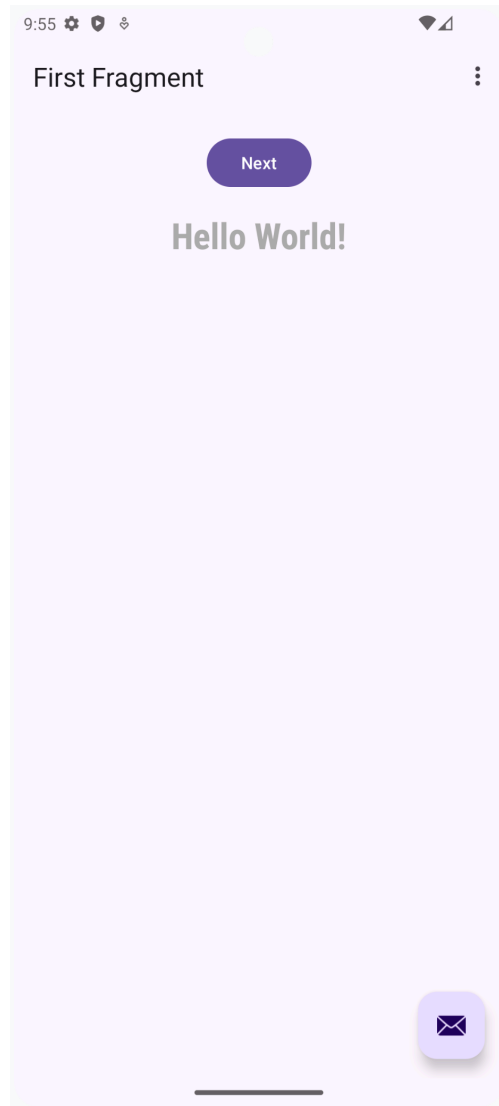
Below is an example of the **textAppearance** attributes after making some changes.



5. Look at the XML for the `TextView`. You see that the new properties have been added.

```
<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:fontFamily="sans-serif-condensed"
    android:text="@string/lorem_ipsum"
    android:textColor="@android:color/darker_gray"
    android:textSize="30sp"
    android:textStyle="bold"
```

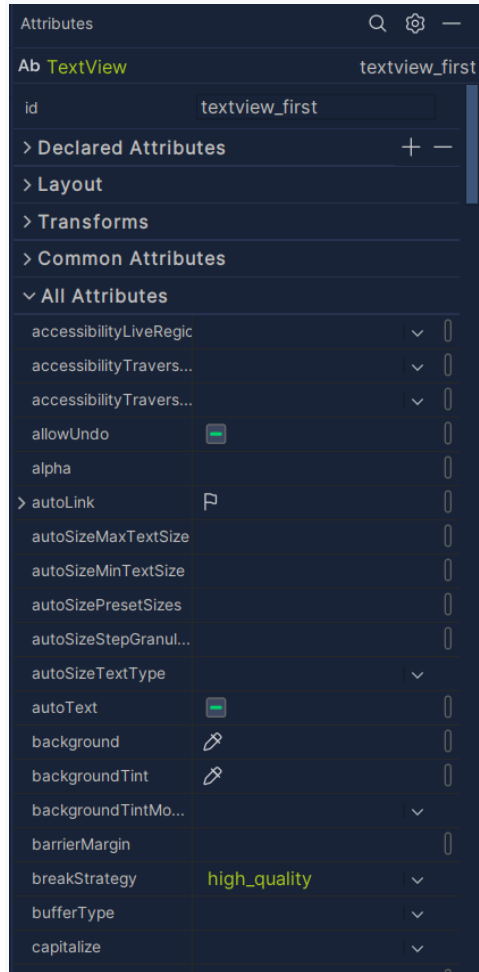
6. Run your app again and see the changes applied to your Hello World! String



Step 6: Display all attributes

1. In the Attributes panel, scroll down until you find All Attributes.

If you don't see any attributes in the **Attributes** panel, make sure `textView_first` is still selected in the **Component Tree**.



2. Scroll through the list to get an idea of the attributes you could set for a `TextView`.

5. Add color resources

So far you have learned how to change property values. Next, you will learn how to create more resources like the string resources you worked with earlier. Using resources enables you to use the same values in multiple places, or to define values and have the UI update automatically whenever the value is changed.

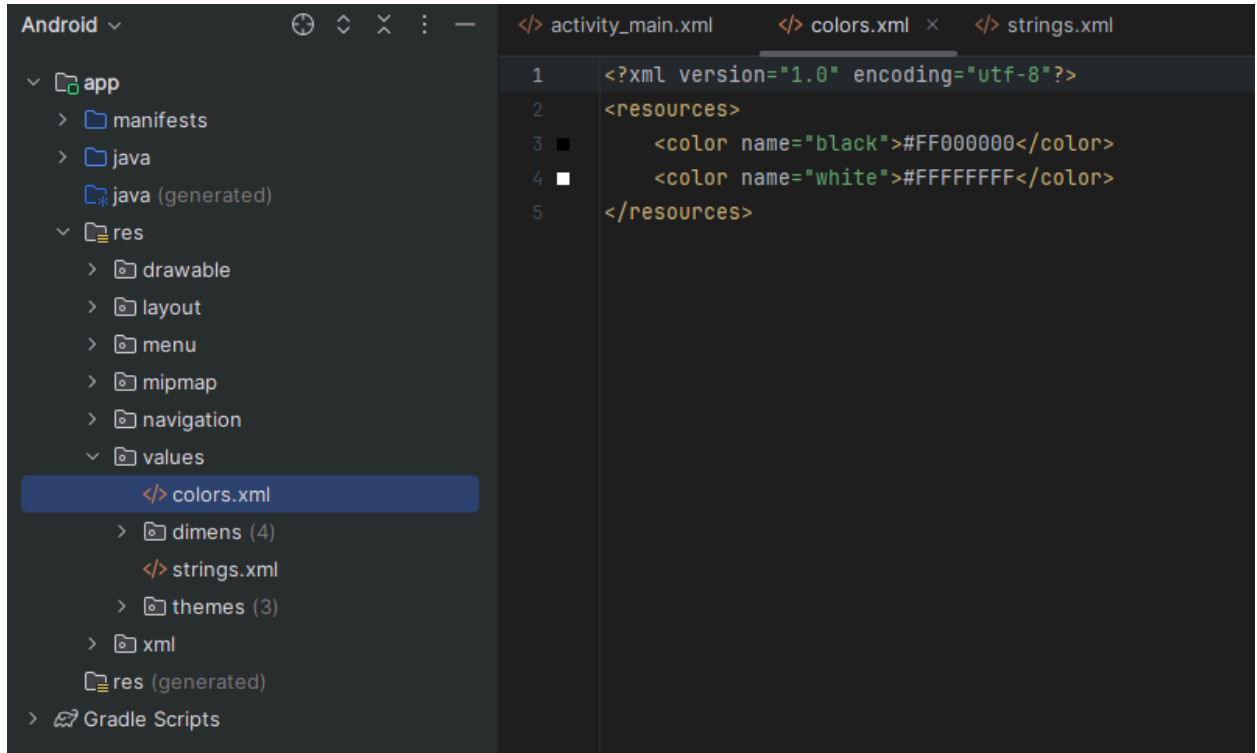
What you'll learn

- How resources are defined.
- Adding and using color resources.
- The results of changing layout height and width.

Step 1: Add color resources

First, you'll learn how to add new color resources.

1. In the Project panel on the left, double-click on res > values > colors.xml to open the color resource file.



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

The `colors.xml` file opens in the editor. So far, two colors have been defined. These are the colors you can use in your app layout.

Note: Different versions of Android Studio use different values for these colors, so you may see other colors here.

2. Go back to `fragment_first.xml` so you can see the XML code for the layout.
3. Add a new property to the `TextView` called `android:background`, and start typing to set its value to `@color`. You can add this property anywhere inside the `TextView` code.

A menu pops up offering the predefined color resources:

```
<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:fontFamily="sans-serif-condensed"
    android:text="@string/lorem_ipsum"
    android:background="@c"
    android:textColor="@color/black"
    android:textSize="30"
    android:textStyle="italic"
    app:layout_constraint...
    app:layout_constraint...
```

4. Choose `@color/black`.
5. Change the property `android:textColor` and give it a value of `@android:color/white`.

The Android framework defines a range of colors, including white, so you don't have to define white yourself.

6. In the layout editor, you can see that the `TextView` now has a black background, and the text is displayed in white.



Step 2: Add a new color to use as the screen background color

1. Back in `colors.xml`, create a new color resource called `screenBackground`:

```
JavaScript  
<color name="screenBackground">#FFEE58</color>
```

A color can be defined as 3 hexadecimal numbers (#00-#FF, or 0-255) representing the red, blue, and green (RGB) components. The color you

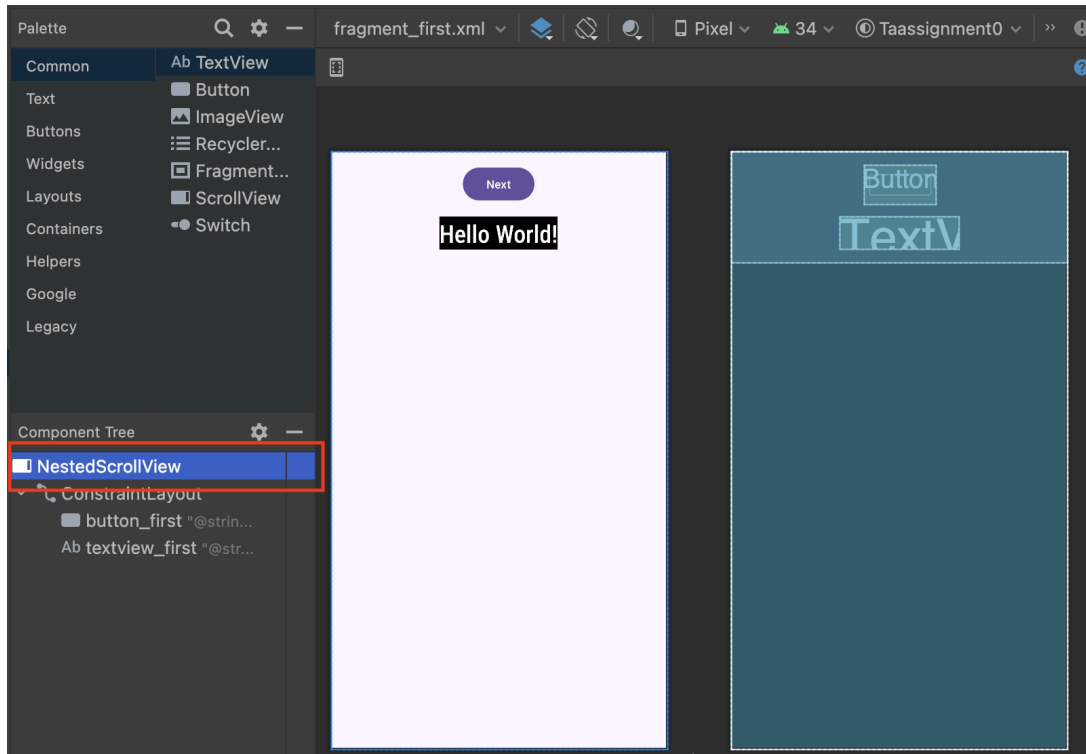
just added is yellow. Notice that the colors corresponding to the code are displayed in the left margin of the editor.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3   <color name="black">#FF000000</color>
4   <color name="white">#FFFFFFFF</color>
5   <color name="screenBackground">#FFEE58</color>
6 </resources>
```

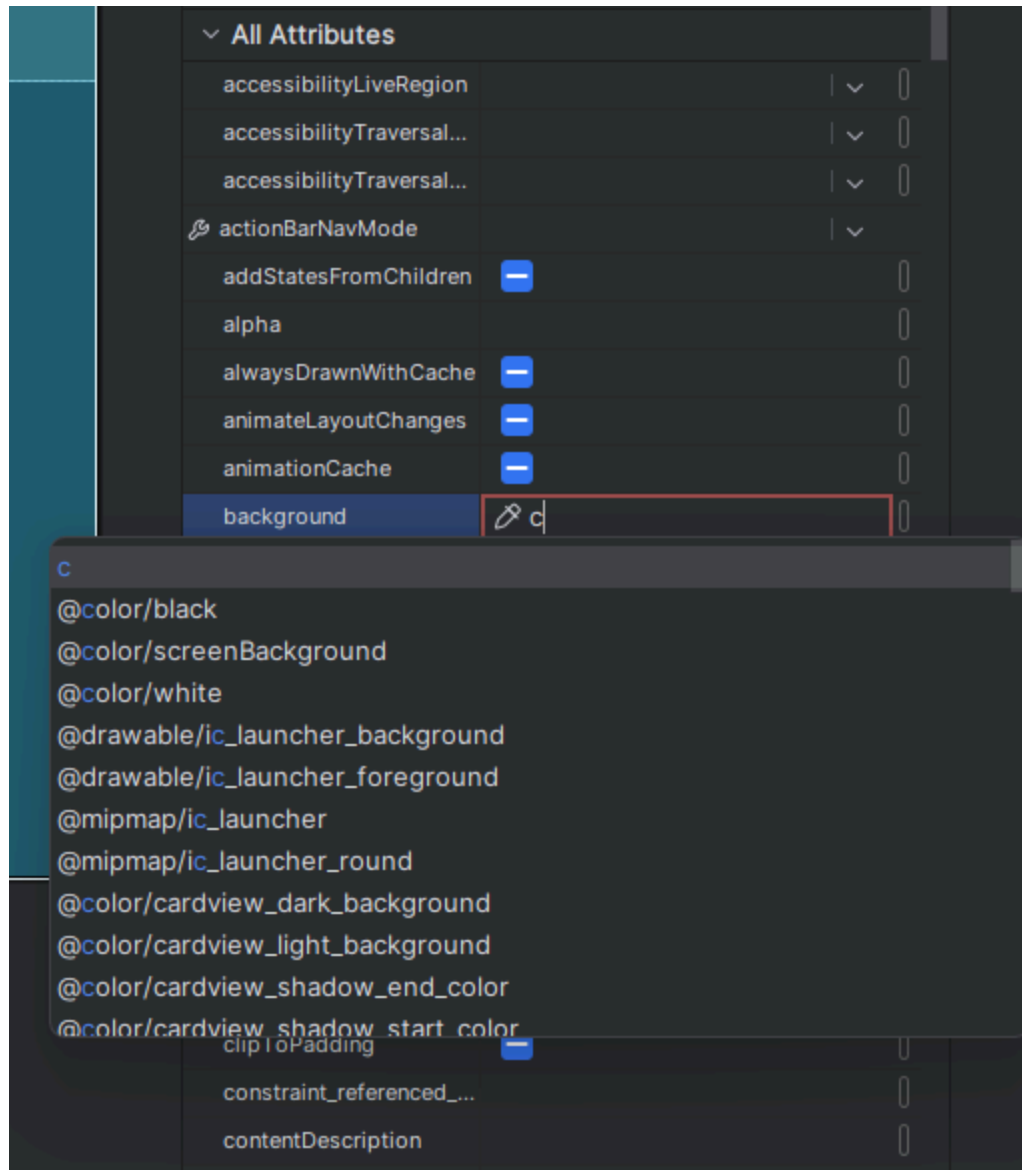
Note that a color can also be defined including an alpha value (#00-#FF) which represents the transparency (#00 = 0% = fully transparent, #FF = 100% = fully opaque). When included, the alpha value is the first of 4 hexadecimal numbers (ARGB).

The alpha value is a measure of transparency. For example, #88FFEE58 makes the color semi-transparent, and if you use #00FFEE58, it's fully transparent and disappears from the left-hand bar.

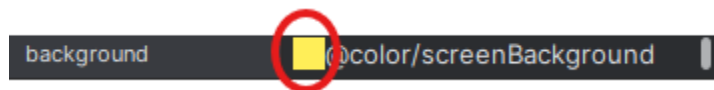
2. Go back to `fragment_first.xml`.
3. In the Component Tree, select the `NestedScrollView`.



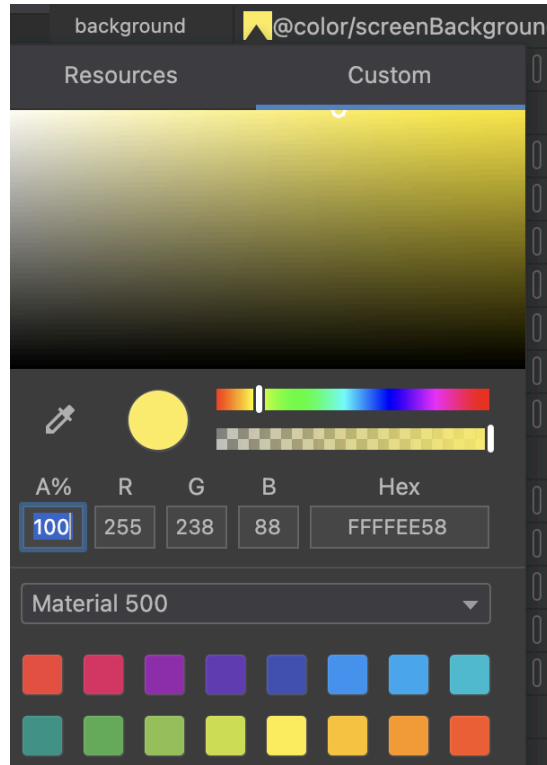
4. In the Attributes panel, select the background property and press Enter. Type "c" in the field that appears.
5. In the menu of colors that appears, select `@color/screenBackground`. Press Enter to complete the selection.



6. Click on the yellow patch to the left of the color value in the background field.



It shows a list of colors defined in `colors.xml`. Click the Custom tab to choose a custom color with an interactive color chooser.

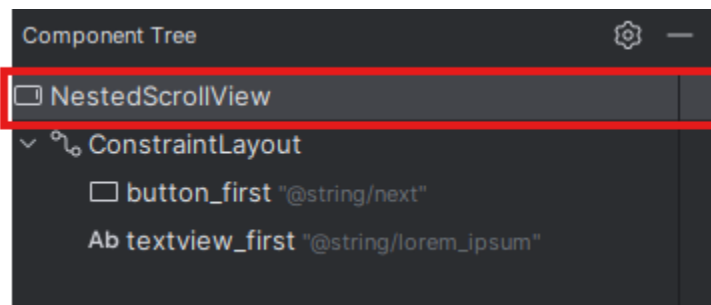


7. Feel free to change the value of the `screenBackground` color, but make sure that the final color is noticeably different from the `black` and `white` colors.

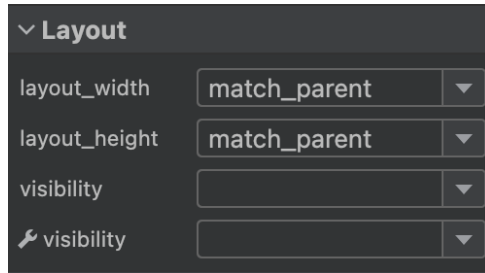
Step 3: Explore width and height properties

Now that you have a new screen background color, you will use it to explore the effects of changing the width and height properties of views.

1. In `fragment_first.xml`, in the Component Tree, select the `NestedScrollView`.



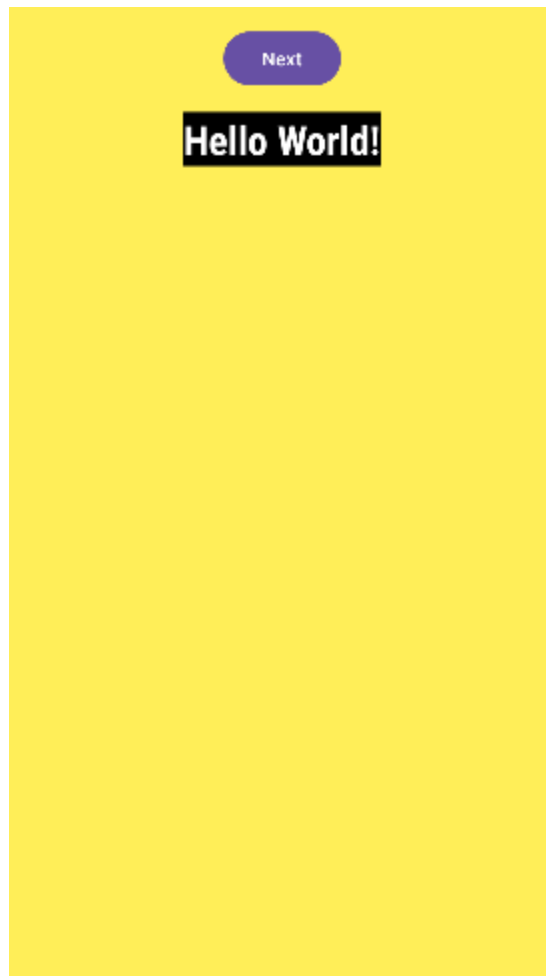
2. In the Attributes panel, find and expand the Layout section.



The `layout_width` and `layout_height` properties are both set to `match_parent`. The `NestedScrollView` is the root view of this `Fragment`, so the "parent" layout size is effectively the size of your screen.

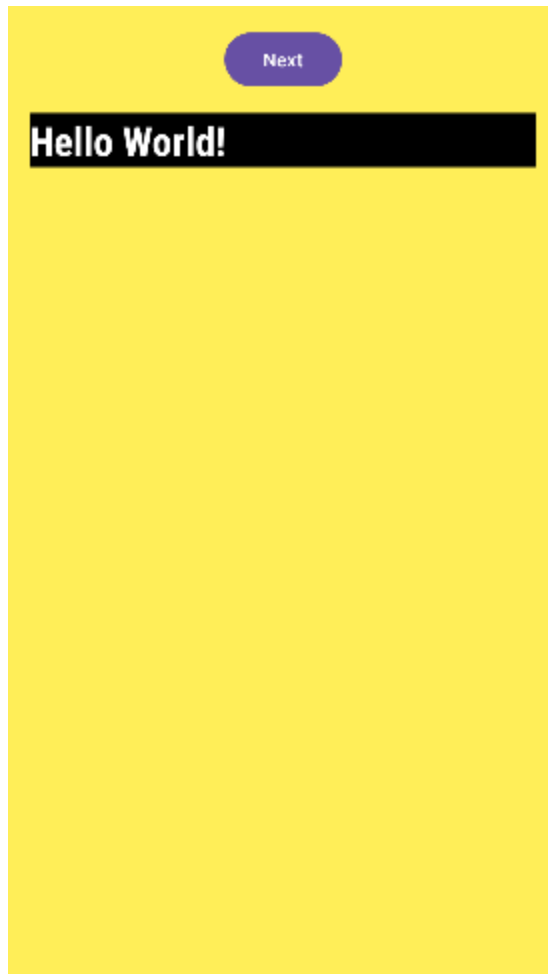
Tip: All views must have `layout_width` and `layout_height` properties.

3. Notice that the entire background of the screen uses the `screenBackground` color.



4. Select `textView_first`. Currently the layout width and height are `wrap_content`, which tells the view to be just big enough to enclose its content (plus padding)
5. Change the layout width to `match_constraint`, which tells the view to be as big as whatever it's constrained to.

The width shows `0dp`, and the text moves to the left, while the `TextView` expands to match the `ConstraintLayout` except for the button. The button and the text view are at the same level in the view hierarchy inside the constraint layout, so they share space.



6. Explore what happens if the height is `match_constraint` and the width is `wrap_content` and vice versa. You can also change the width and height of the `button_first`.

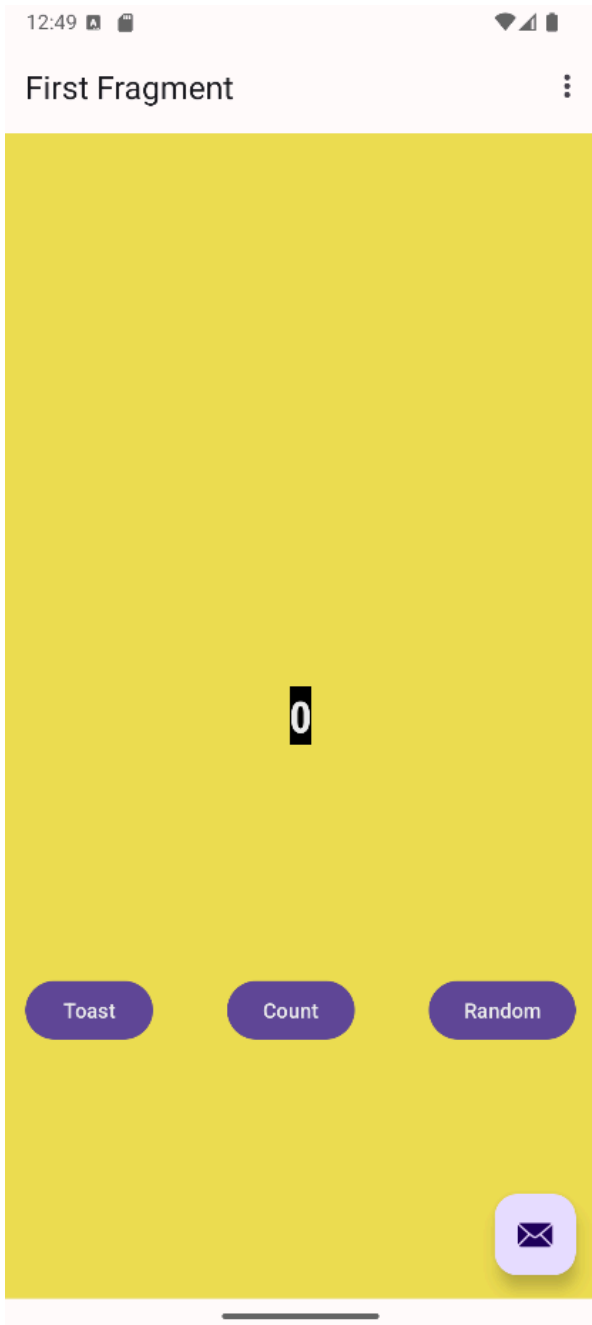
Note: When setting `layout_height` to `match_constraint` for a `TextView`, the view may disappear. In `ConstraintLayout`, `match_constraint` (**select from attribute drop-down menu**) (or “0dp”) means the view's size will be determined by its constraints.

If there isn't enough space between the top and bottom constraints, the `TextView` might collapse to zero height, making it seem like it has disappeared.

7. Set both the width and height of the `TextView` and the `Button` back to `wrap_content`.

6. Add views and constraints

In this task, you will add two more buttons to your user interface, and



update the existing button, as shown below.

What you'll learn

- How to add new views to your layout.
- How to constrain the position of a view to another view.

Step 1: View constraint properties

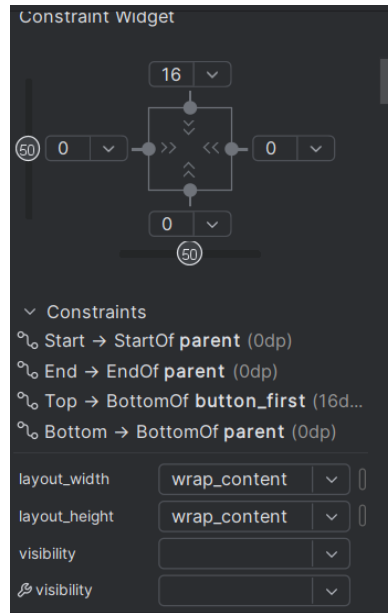
1. In `fragment_first.xml`, look at the constraint properties for the `TextView`.

```
app:layout_constraintBottom_toBottomOf="parent"  
app:layout_constraintEnd_toEndOf="parent"  
app:layout_constraintStart_toStartOf="parent"  
app:layout_constraintTop_toBottomOf="@id/button_first" />
```

These properties define the position of the `TextView`. Read them carefully.

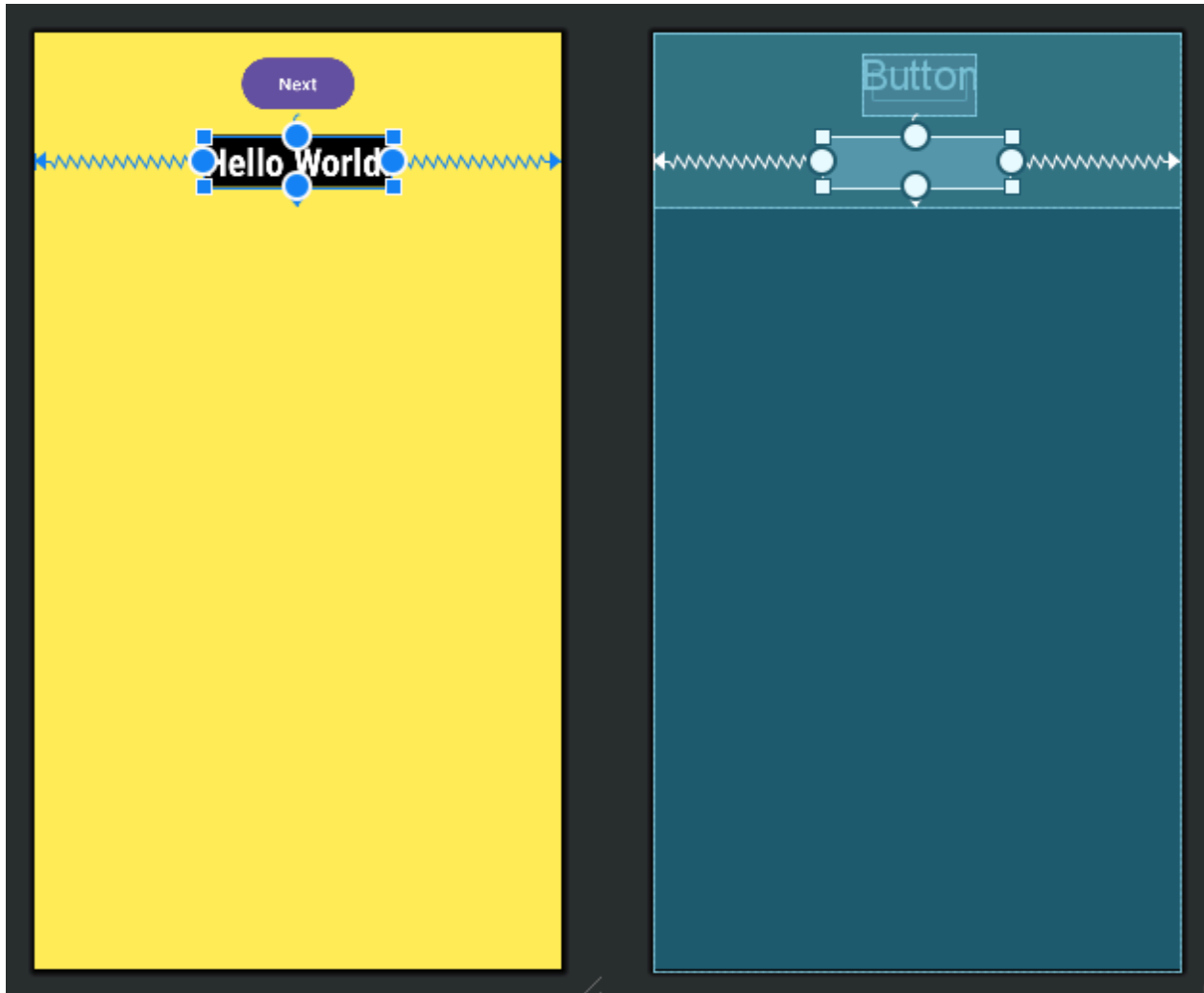
You can constrain the top, bottom, left, and right of a view to the top, bottom, left, and right of other views.

2. Select `textview_first` in the **Component Tree** and look at the **Constraint Widget** in the **Attributes** panel.



The square represents the selected view. Each of the grey dots represents a constraint, to the top, bottom, left, and right; for this example, from the `TextView` to its parent, the `ConstraintLayout`, or to the `Next` button for the bottom constraint.

3. Notice that the blueprint and design views also show the constraints when a particular view is selected. Some of the constraints are jagged lines, but the one to the `Next` button is a squiggle, because it's a little different. You'll learn more about that in a bit.

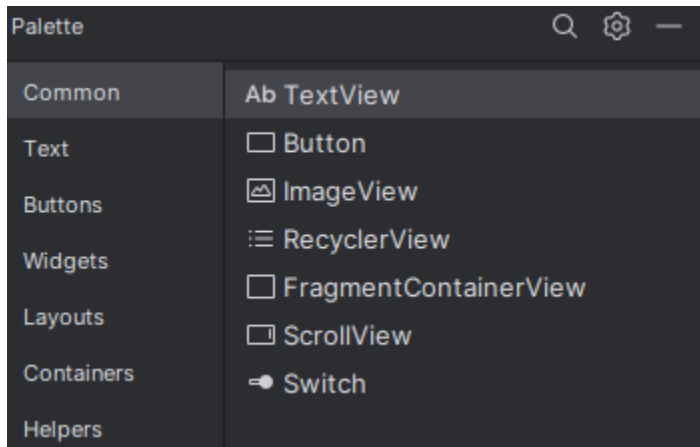


3. Add `android:fillViewport="true"` to `NestedScrollView` properties.

Step 2: Add buttons and constrain their positions

To learn how to use constraints to connect the positions of views to each other, you will add buttons to the layout. Your first goal is to add a button and some constraints, and change the constraints on the **Next** button.

1. Notice the **Palette** at the top left of the layout editor. Move the sides if you need to, so that you can see many of the items in the palette.

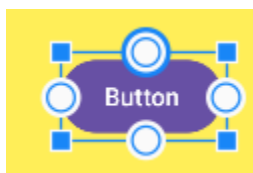


2. Click on some of the categories, and scroll the listed items if needed to get an idea of what's available.
3. Select **Button**, which is near the top, and drag and drop it onto the design view, placing it underneath the TextView near the other button. Notice that a Button has been added to the **Component Tree** under ConstraintLayout.

Step 3: Add a constraint to the new button

You will now constrain the top of the button to the bottom of the TextView.

1. Move the cursor over the circle at the top of the Button.



2. Click and drag the circle at the top of the Button onto the circle at the bottom of the TextView. The **Button** moves up to sit just below



the `TextView` because the top of the button is now *constrained* to the bottom of the `TextView`.

3. Take a look at the **Constraint Widget** in the **Layout** pane of the **Attributes** panel. It shows some constraints for the Button, including **Top -> BottomOf textView**.
4. Take a look at the XML code for the button. It now includes the attribute that constrains the top of the button to the bottom of the `TextView`.

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

5. You may see a warning, "**Not Horizontally Constrained**". To fix this, add a constraint from the left side of the button to the left side of the screen.
6. Also add a constraint to constrain the bottom of the button to the bottom of the screen.



Before adding another button, relabel this button so things are a little clearer about which button is which.

1. Click on the button you just added in the design layout.
2. Look at the **Attributes** panel on the right, and notice the **id** field.
3. Change the **id** from `button` to `toast_button`.

* you can also modify it via code from the xml file

Step 4: Adjust the Next button

You will adjust the button labeled **Next**, which Android Studio created for you when you created the project. The constraint between it and the

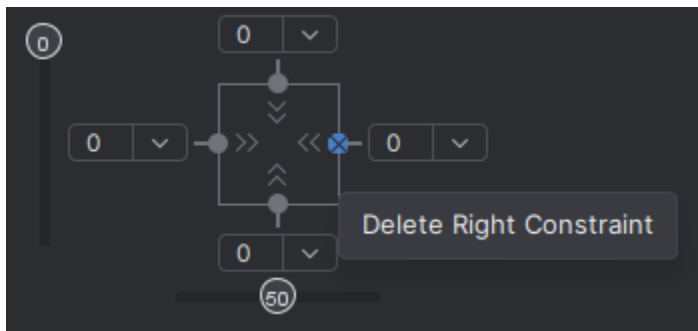
TextView looks a little different, a wavy line instead of a jagged one, with no arrow. This indicates a [chain](#), where the constraints link two or more objects to each other, instead of just one to another. For now, you'll delete the chained constraints and replace them with regular constraints.

To delete a constraint:

- In the design view or blueprint view, hold the `Ctrl` key (Command on a Mac) and move the cursor over the circle for the constraint until the circle highlights, then click the circle.



- Or click on one of the constrained views, then right-click on the constraint and select **Delete** from the menu.
- Or in the **Attributes** panel, move the cursor over the circle for the constraint until it shows an x, then click it.



* You can also delete it by deleting the line in the xml file.

If you delete a constraint and want it back, either undo the action, or create a new constraint.

Step 5: Delete the chain constraints

1. Click on the **Next** button, and then delete the constraint from the bottom of the button to the TextView.

2. Click on the TextView, and then delete the constraint from the top of the text to the **Next** button.

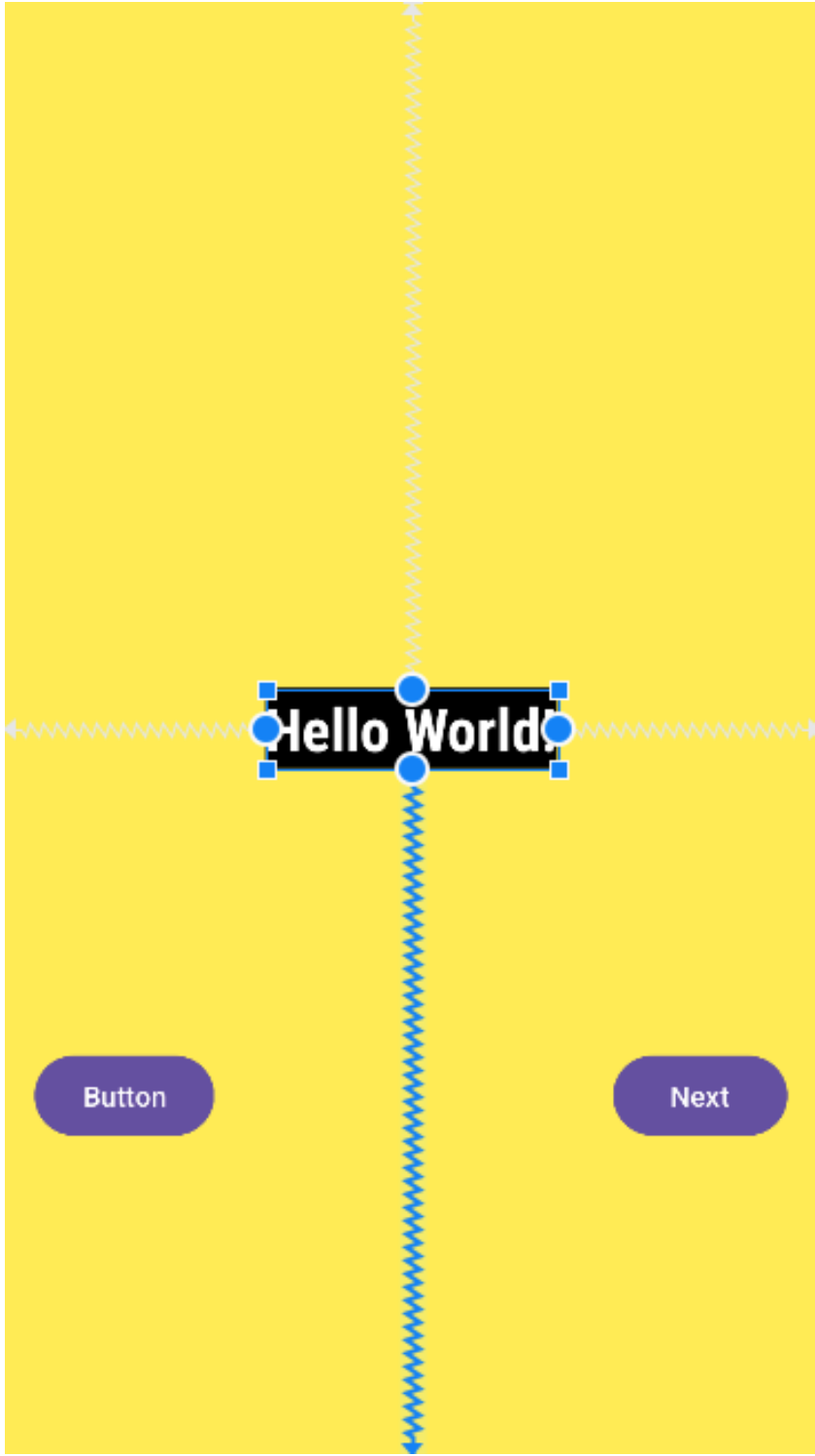
Step 6: Add new constraints

1. Constrain the right side of the **Next** button to the right of the screen if it isn't already.
2. Delete the constraint on the left side of the **Next** button.
3. Now constrain the top and bottom of the **Next** button so that the top of the button is constrained to the bottom of the TextView and the bottom is constrained to the bottom of the screen. The right side of the button is constrained to the right side of the screen.
4. Also constrain the bottom of the TextView to the bottom of the screen and the top of the TextView to the top of the screen.
5. Add `android:fillViewport="true"` to NestedScrollView properties to expand the ConstraintLayout (if you did not add previously).

```
<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"
    android:background="@color/screenBackground"
    tools:context=".FirstFragment"
    android:fillViewport="true">
```

It may seem like the views are jumping around a lot, but that's normal as you add and remove constraints.

Your layout should now look something like this.



```

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

<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@color/black"
    android:fontFamily="sans-serif-condensed"
    android:text="Hello World!"
    android:textColor="@android:color/white"
    android:textSize="30sp"
    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:text="Button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textview_first" />

```

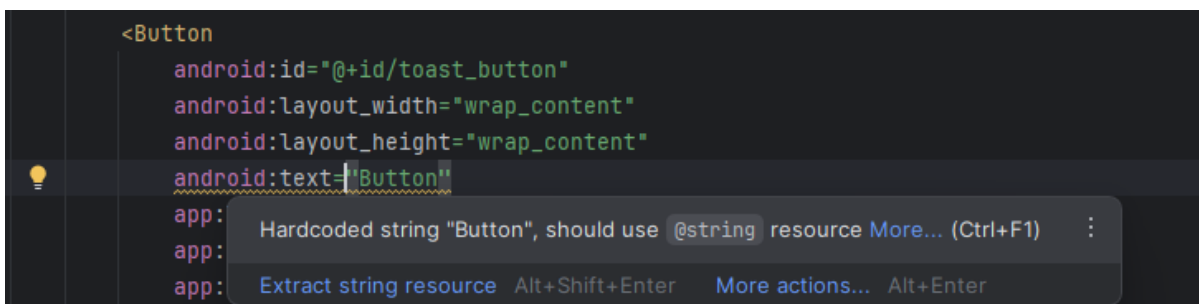
* Notice the buttons move when you move textview, since they are constrained to it.

Step 7: Extract string resources

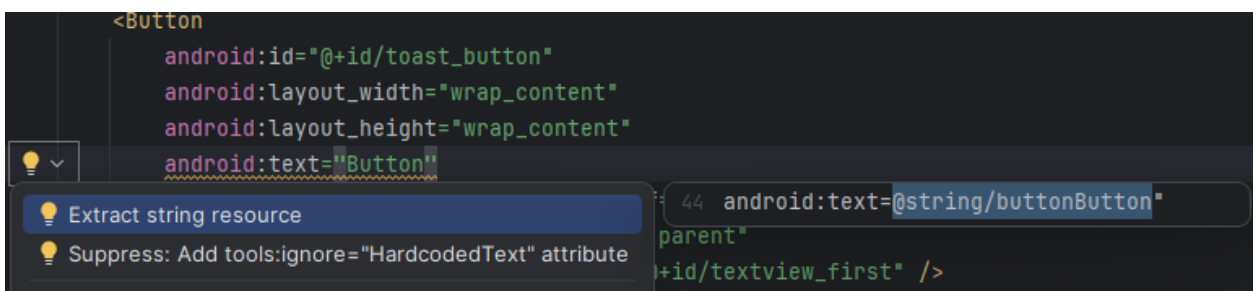
1. In the `fragment_first.xml` layout file, find the text property for the `toast_button` button.

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

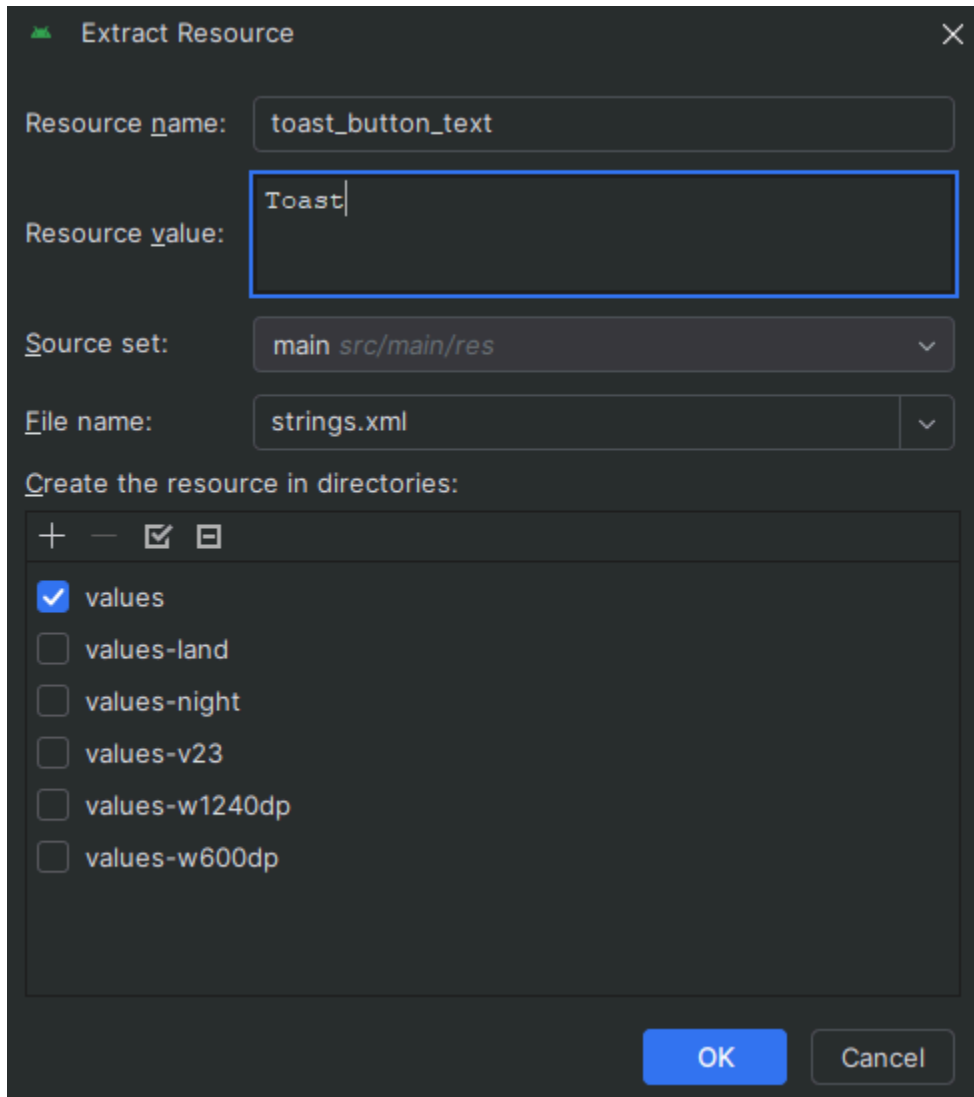
2. Notice that the text "Button" is directly in the layout field, instead of referencing a string resource as the `TextView` does. This will make it harder to translate your app to other languages.
3. To fix this, click the highlighted code. A light bulb appears on the left.



4. Click the lightbulb. In the menu that pops up, select **Extract string resource**.



5. In the dialog box that appears, change the resource name to `toast_button_text` and the resource value to `Toast` and click **OK**.



6. Notice that the value of the `android:text` property has changed to `@string/toast_button_text`.

```
<Button
    android:id="@+id/toast_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/toast_button_text"
```

7. Go to the `res > values > strings.xml` file. Notice that a new string

```
<string name="toast_button_text">Toast</string>
```



resource has been added, named `toast_button_text`. Run the

app to make sure it displays as you expect it to.

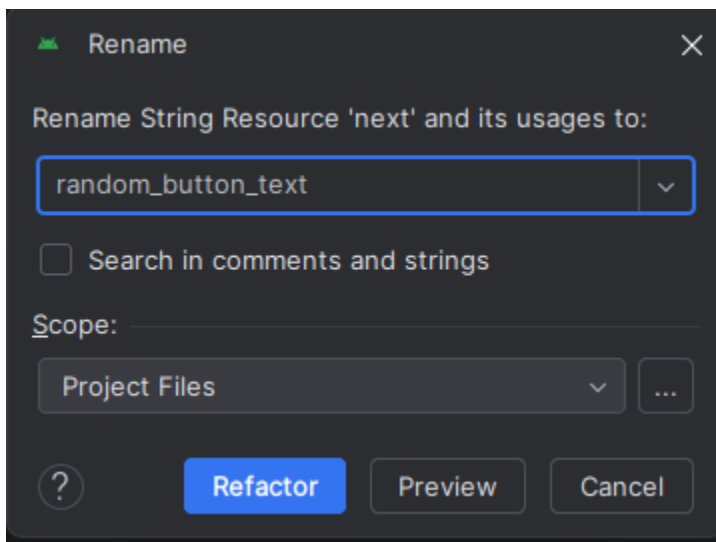
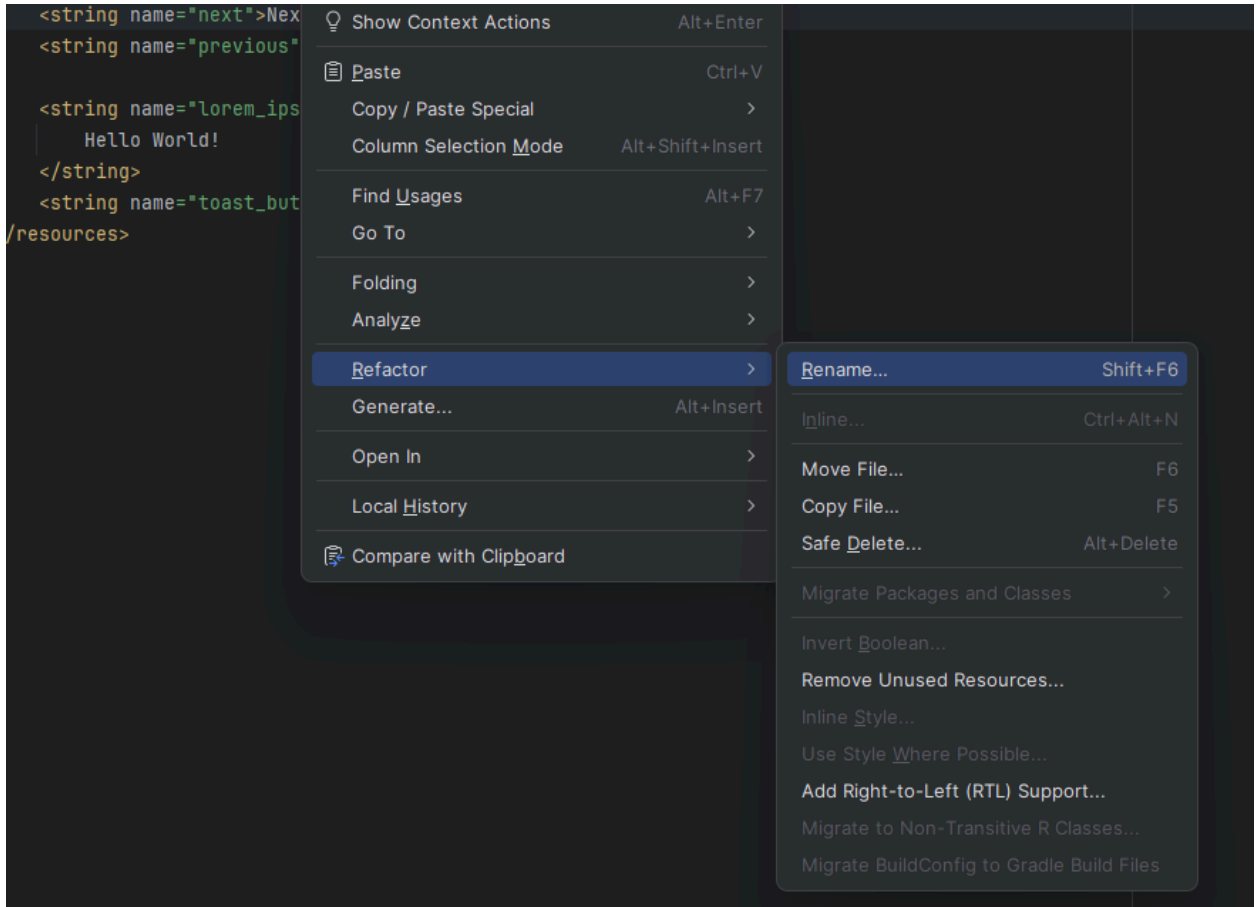
You now know how to create new string resources by extracting them from existing field values. (You can also add new resources to the `strings.xml` file manually.) And you know how to change the id of a view.

Note: The `id` for a view helps you identify that view distinctly from other views. You'll use this later to find particular views using the `findViewById()` method in your Java code.

Step 8: Update the Next button

The Next button already has its text in a string resource, but you'll make some changes to the button to match its new role, which will be to generate and display a random number.

1. As you did for the **Toast** button, change the id of the **Next** button from `button_first` to `random_button` in the **Attributes** panel.
2. If you get a dialog box asking to update all usages of the button, click **Yes**. This will fix any other references to the button in the project code.
3. In `strings.xml`, right-click on the `next` string resource.
4. Select **Refactor** > **Rename...** and change the name to `random_button_text`.



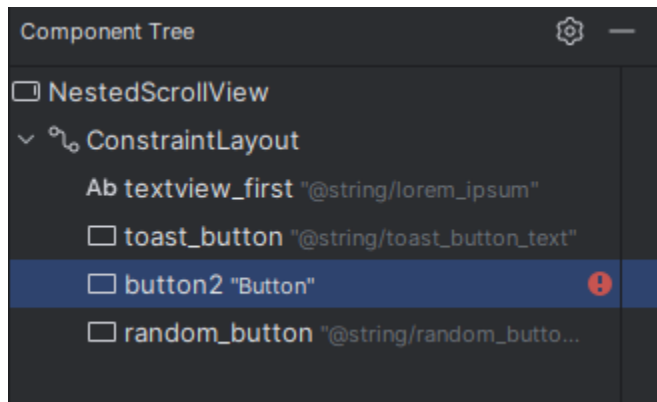
5. Click **Refactor** to rename your string and close the dialog.
6. Change the value of the string from Next to Random.

7. If you want, move `random_button_text` to below `toast_button_text`.

Step 9: Add a third button

Your final layout will have three buttons, vertically constrained the same, and evenly spaced from each other.

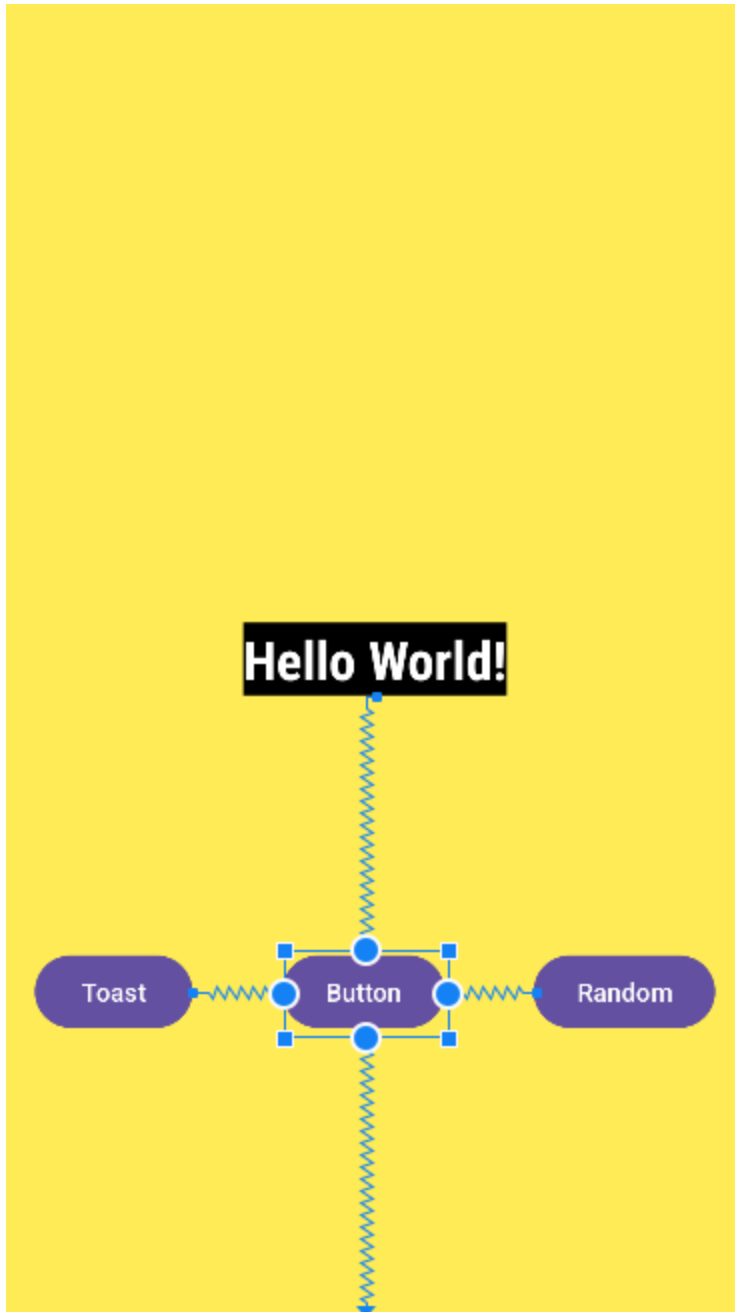
1. In `fragment_first.xml`. Drag **Random** button below the **Toast** button. Add another button to the layout, and drop it somewhere between the **Toast** button and the **Random** button, below the



TextView.

2. Add vertical constraints the same as the other two buttons. Constrain the top of the third button to the bottom of TextView; constrain the bottom of the third button to the bottom of the screen.
3. Add horizontal constraints from the third button to the other buttons. Constrain the left side of the third button to the right side of the **Toast** button; constrain the right side of the third button to the left side of the **Random** button.

Your layout should look something like this:



4. Examine the XML code for `fragment_first.xml`. Do any of the buttons have the attribute `app:layout_constraintVertical_bias`? It's OK if you do not see that constraint.

The "bias" constraints allows you to tweak the position of a view to be more on one side than the other when both sides are [constrained in opposite](#)

[directions](#). For example, if both the top and bottom sides of a view are constrained to the top and bottom of the screen, you can use a vertical bias to place the view more towards the top than the bottom.

Here is the XML code for the finished layout. Your layout might have different margins and perhaps some different vertical or horizontal bias constraints. The exact values of the attributes for the appearance of the TextView might be different for your app.

JavaScript

```
<?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"

    android:background="@color/screenBackground"

    tools:context=".FirstFragment"

    android:fillViewport="true">

    <androidx.constraintlayout.widget.ConstraintLayout
```



```
android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
```

```
android:padding="16dp">
```

```
<TextView
```

```
    android:id="@+id/textview_first"
```

```
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
```

```
    android:background="@color/black"
```

```
    android:fontFamily="sans-serif-condensed"
```

```
    android:text="@string/lorem_ipsum"
```

```
    android:textColor="@android:color/white"
```

```
    android:textSize="30sp"
```

```
    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: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/button2"
```

```
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
```

```
    android:text="Button"
```

```
    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" />
```

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

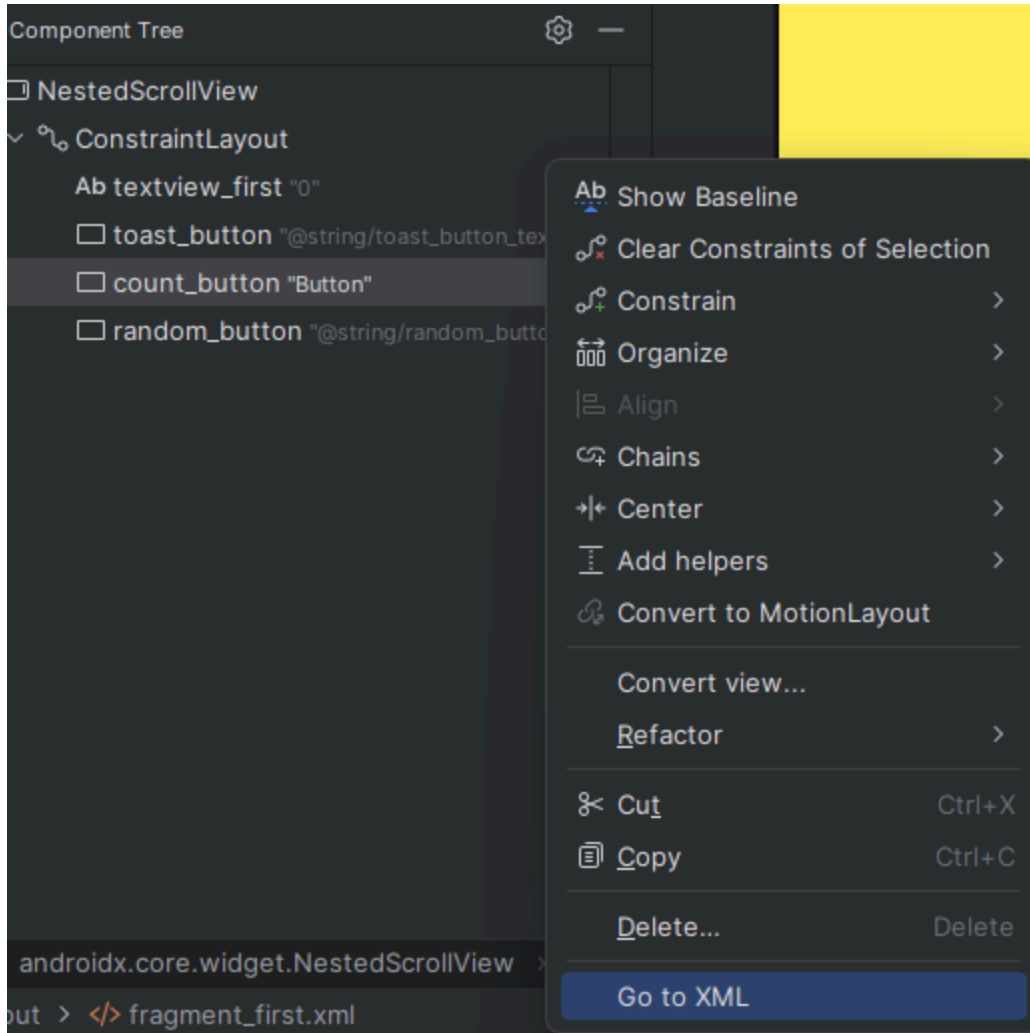
</androidx.constraintlayout.widget.ConstraintLayout>

</androidx.core.widget.NestedScrollView>
```

Step 10: Get your UI ready for the next task

The next task is to make the buttons do something when they are pressed. First, you need to get the UI ready.

1. Change the text of the TextView to show **0** (the number zero).
2. Change the id of the last button you added, button2, to count_button in the **Attributes** panel in the design editor.
3. In the XML, extract the string resource to count_button_text and set the value to Count .



The buttons should now have the following text and ids:

Button	text	id
Left button	Toast	@+id/toast_button
Middle button	Count	@+id/count_button
Right button	Random	@+id/random_button

5. Run the app.

Step 11: Fix errors if necessary

If you edited the XML for the layout directly, you might see some errors

```
app:layout_constraintEnd_toStartOf="@+id/button"  
app:layout_constraintStart_toEndOf="@+id/button2"
```

The errors occur because the buttons have changed their id and now these constraints are referencing non-existent views.

If you have these errors, fix them by updating the id of the buttons in the constraints that are underlined in red.

```
.....\app\src\main\java\com\example\myfirstapp\FirstFragment.java:32: error: cannot find symbol  
binding.buttonFirst.setOnClickListener(v ->  
    ^  
symbol:   variable buttonFirst  
location: variable binding of type FragmentFirstBinding
```

```
binding.buttonFirst.setOnClickListener(v ->  
    NavHostF  
);  
}
```

Cannot resolve symbol 'buttonFirst'

Rename reference Alt+Shift+Enter More actions... Alt+Enter

```
binding.textviewFirst.setOnClickListener(v ->  
    ( fragment: FirstFragment.this)  
    stFragment_to_SecondFragment)  
);
```

textviewFirst
toastButton
countButton
randomButton

Press Enter or Tab to replace

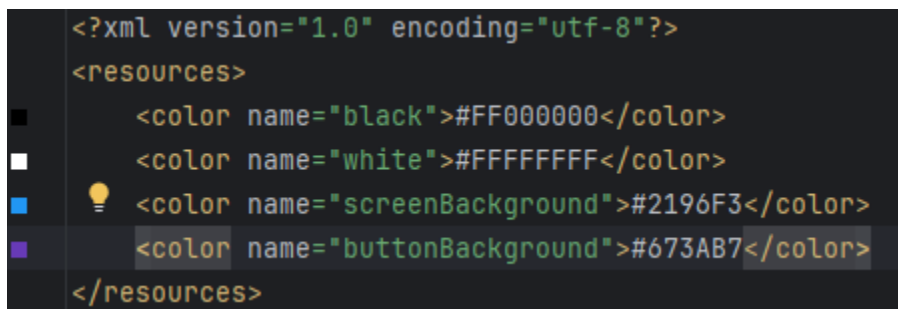
7. Update the appearance of the buttons and the TextView

Your app's layout is now basically complete, but its appearance can be improved with a few small changes.

Step 1: Add new color resources

1. In `colors.xml`, change the value of `screenBackground` to `#2196F3`, which is a blue shade in the [Material Design palette](#).
2. Add a new color named `buttonBackground`. Use the value `#673AB7`, which is a darker shade in the purple palette.

```
<color name="buttonBackground">#673AB7</color>
```



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="black">#FF000000</color>
  <color name="white">#FFFFFFFF</color>
  <color name="screenBackground">#2196F3</color>
  <color name="buttonBackground">#673AB7</color>
</resources>
```

Step 2: Add a background color for the buttons

1. In the layout, add a background color to each of the buttons. (You can either edit the XML in `fragment_first.xml` or use the **Attributes** panel, whichever you prefer.)

```
android:background="@color/buttonBackground"
```

```

<Button
    android:id="@+id/toast_button"
    android:background="@color/buttonBackground"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Toast"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textview_first" />

<Button
    android:id="@+id/count_button"
    android:background="@color/buttonBackground"
    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" />

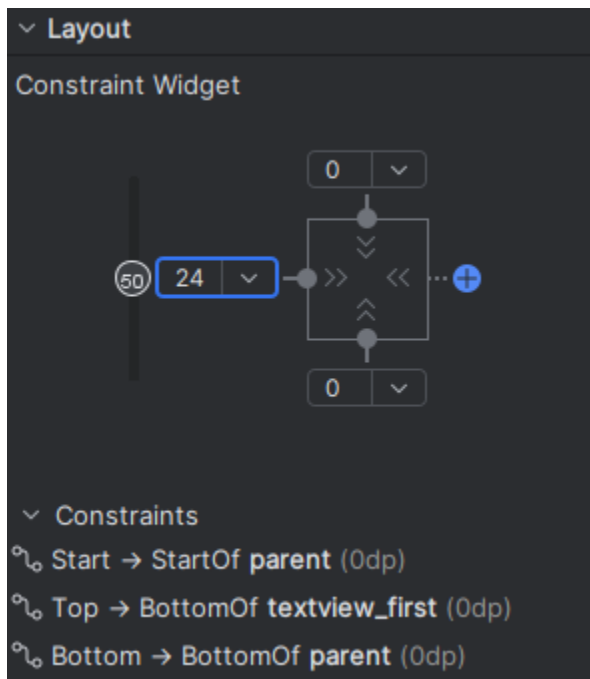
<Button
    android:id="@+id/random_button"
    android:background="@color/buttonBackground"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Random"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textview_first" />

```

Step 3: Change the margins of the left and right buttons

1. Give the **Toast** button a left (start) margin of 24dp and give the **Random** button a right (end) margin of 24dp. (Using start and end instead of left and right makes these margins work for all language directions.)

One way to do this is to use the **Constraint Widget** in the **Attributes** panel. The number on each side is the margin on that side of the selected view. Type **24** in the field and press **Enter**.



Step 4: Update the appearance of the TextView

1. Remove the background color of the TextView, either by clearing the value in the **Attributes** panel or by removing the `android:background` attribute from the XML code. When you remove the background, the view background becomes transparent.
2. Increase the text size of the **TextView** to 72sp.

```
android:textSize="72sp"
```



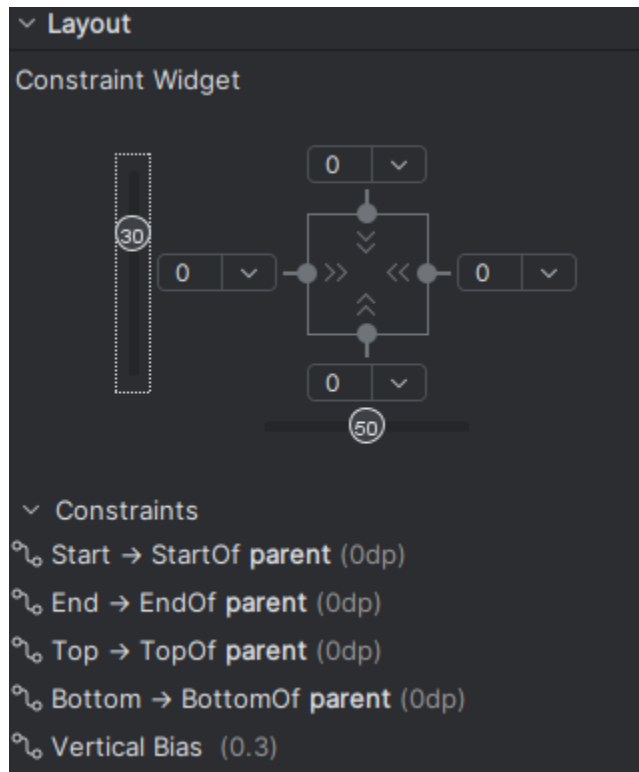
```
<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:fontFamily="sans-serif"
    android:text="0"
    android:textColor="@android:color/white"
    android:textSize="72sp"
    android:textStyle="bold"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

3. Change the font-family of the TextView to sans-serif (if it's not already).
4. Add an app:layout_constraintVertical_bias property to the TextView, to bias the position of the view upwards a little so that it is more evenly spaced vertically in the screen. Feel free to adjust the value of this constraint as you like. (Check in the design view to see how the layout looks.)

app:layout_constraintVertical_bias="0.3"

```
<TextView
    android:id="@+id/textview_first"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:fontFamily="sans-serif"
    android:text="0"
    android:textColor="@android:color/white"
    android:textSize="72sp"
    android:textStyle="bold"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.3" />
```

5. You can also set the vertical bias using the **Constraint Widget**. Click and drag the number **50** that appears on the left side, and slide it upwards until it says **30**.



6. Make sure the **layout_width** is **wrap_content**, and the horizontal bias is 50
(`app:layout_constraintHorizontal_bias="0.5"` in XML).

Step 5: Run your app

If you implemented all the updates, your app will look like the following figure. If you use different colors and fonts, then your app will look a bit different.

0

Toast

Count

Random



8. Make your app interactive

You have added buttons to your app's main screen, but currently the buttons do nothing. In this task, you will make your buttons respond when the user presses them.

First you will make the **Toast** button show a pop-up message called a [toast](#). Next you will make the **Count** button update the number that is displayed in the `TextView`.

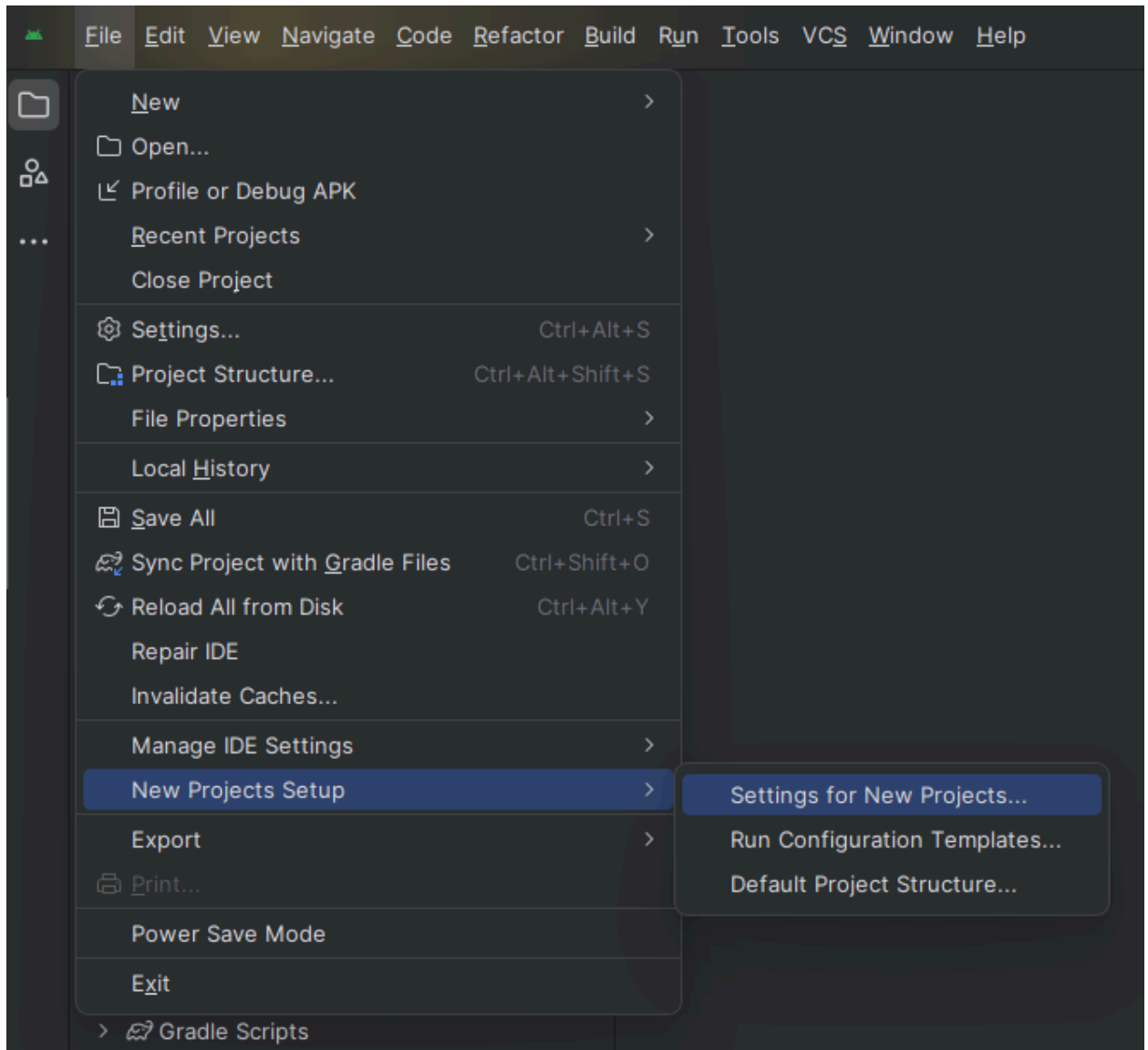
What you'll learn

- How to find a view by its ID.
- How to add click listeners for a view.
- How to set and get property values of a view from your code.

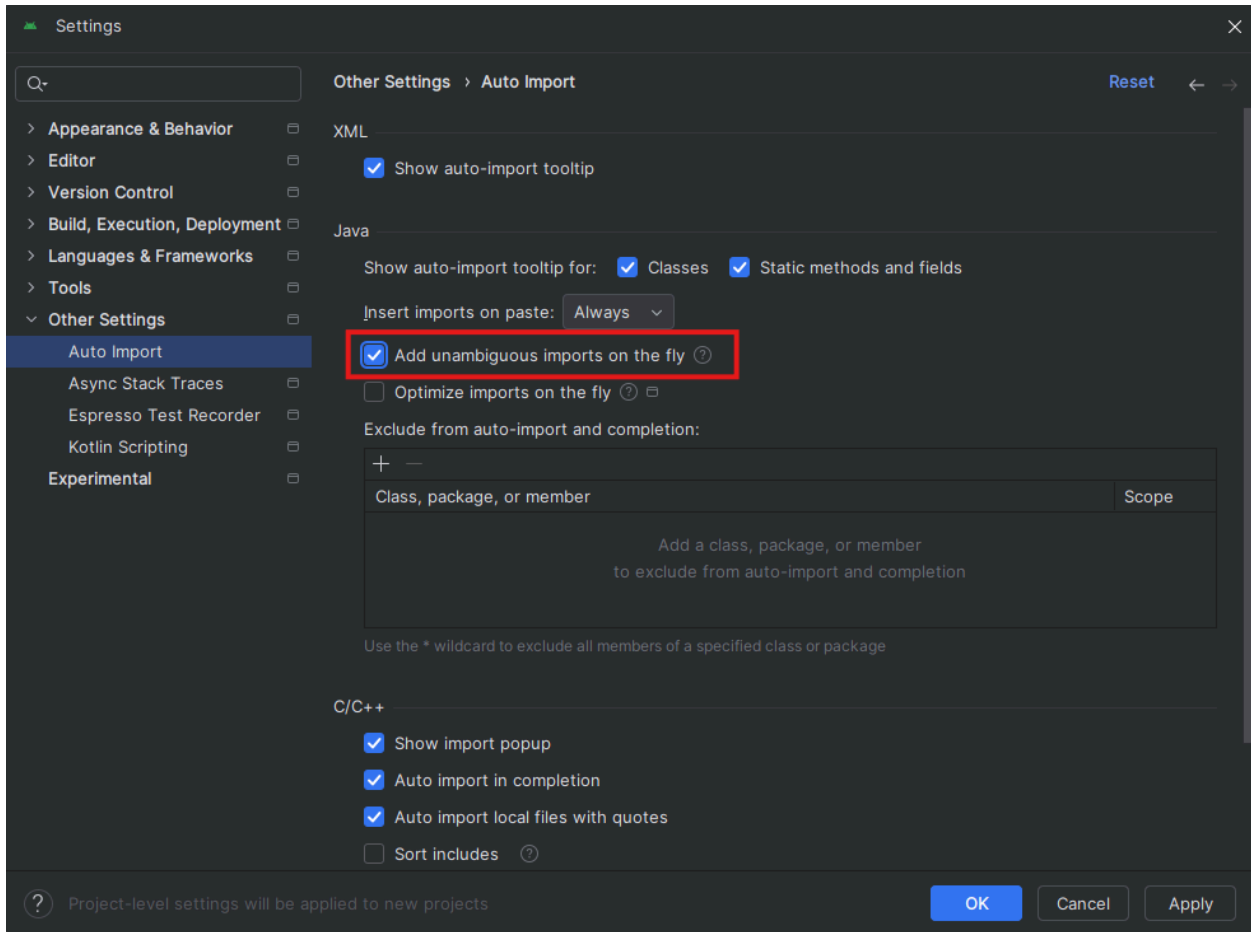
Step 1: Enable auto imports

To make your life easier, you can enable auto-imports so that Android Studio automatically imports any classes that are needed by the Java code.

1. In Android Studio, open the settings editor by going to **File > New Projects Setup > Settings for New Projects... > Other Settings**



2. Select **Auto Imports**. In the **Java** section, make sure **Add Unambiguous Imports on the fly** is checked.



3. Close the settings editor by pressing **OK**.

Step 2: Show a toast

In this step, you will attach a Java method to the **Toast** button to show a toast when the user presses the button. A toast is a short message that appears briefly at the bottom of the screen.

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

This class has only two methods, `onCreateView()` and `onViewCreated()`. These methods execute when the fragment starts.

As mentioned earlier, the [id](#) for a view helps you identify that view distinctly from other views. Using the `findViewById()` method, your code can find the `random_button` using its id, `R.id.random_button`.

2. Copy and paste the following code into `onViewCreated()`. It sets up a click listener for the `random_button`, which was originally created as the **Next** button.

Java

```
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_SecondFragment);

    }

});
```

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    view.findViewById(R.id.random_button).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            NavHostFragment.findNavController( fragment: FirstFragment.this)
                .navigate(R.id.action_FirstFragment_to_SecondFragment);
        }
    });
}
```

Here is what this code does:

- 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 in this case is for navigating to another fragment, and navigate there. (You will learn about that later.)

3. Just below that click listener, add code to set up a click listener for the `toast_button`, which creates and displays a toast. Here is the code:

Java

```
view.findViewById(R.id.toast_button).setOnClickListener(new
View.OnClickListener() {

    @Override

    public void onClick(View view) {

        Toast myToast = Toast.makeText(getActivity(), "Hello toast!",
Toast.LENGTH_SHORT);

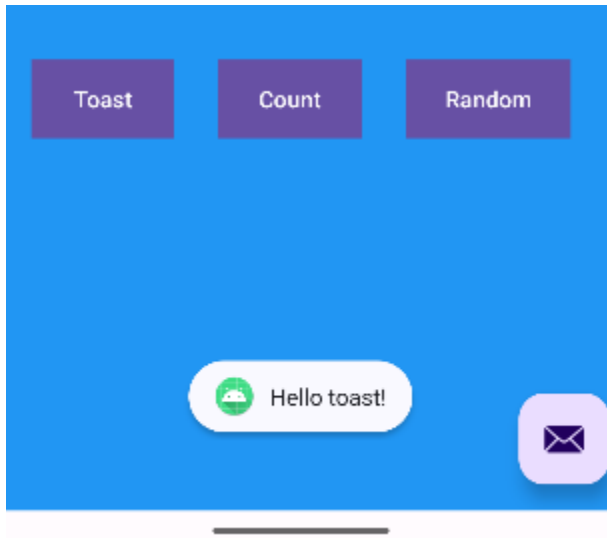
        myToast.show();

    }

});
```

```
public void onCreateView(@NonNull View view, Bundle savedInstanceState) {
    view.findViewById(R.id.toast_button).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast myToast = Toast.makeText(getActivity(), text: "Hello toast!", Toast.LENGTH_SHORT);
            myToast.show();
        }
    });
}
```


4. Run the app and press the **Toast** button. Do you see the toasty message at the bottom of the screen?



5. If you want, extract the message string into a resource as you did for the button labels.

You have learned that to make a view interactive you need to set up a click listener for the view which says what to do when the view (button) is clicked on. The click listener can either:

- Implement a small amount of code directly.
- Call a method that defines the desired click behavior in the activity.

Step 3: Make the **Count** button update the number on the screen

The method that shows the toast is very simple; it does not interact with any other views in the layout. In the next step, you add behavior to your layout to find and update other views.

Update the **Count** button so that when it is pressed, the number on the screen increases by 1.

1. In the `fragment_first.xml` layout file, notice the `id` for the `TextView`:

```
<TextView
    android:id="@+id/textview_first"
```

2. In `FirstFragment.java`, add a click listener for the `count_button` below the other click listeners in `onViewCreated()`. Because it has a little more work to do, have it call a new method, `countMe()`.

Java

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

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    view.findViewById(R.id.toast_button).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast myToast = Toast.makeText(getActivity(), text: "Hello toast!", Toast.LENGTH_SHORT);
            myToast.show();
        }
    });

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

3. In the `FirstFragment` class, add the method `countMe()` that takes a single `View` argument. This method will be invoked when the **Count** button is clicked and the click listener called.

Java

```
private void countMe(View view) {  
  
}
```

4. Get the value of the `showCountTextView`. You will define that in the next step.

...

Java

```
// Get the value of the text view  
String countString = showCountTextView.getText().toString();
```

5. Convert the value to a number, and increment it.

Java

```
...  
// Convert value to a number and increment it  
Integer count = Integer.parseInt(countString);  
count++;
```

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

Java

```
...  
// Display the new value in the text view.  
showCountTextView.setText(count.toString());
```

Here is the whole method:

Java

```
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());  
  
}
```

```

private void countMe(View view) { 1 usage
    // 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());
}

```

Step 4: Cache the TextView for repeated use

You could call `findViewById()` in `countMe()` to find `showCountTextView`. However, `countMe()` is called every time the button is clicked, and `findViewById()` is a relatively time consuming method to call. So it is better to find the view once and cache it.

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

`TextView showCountTextView;`

```

public class FirstFragment extends Fragment {

    private FragmentFirstBinding binding; 3 usages
    TextView showCountTextView; 2 usages

    @Override
    public View onCreateView(
        @NonNull LayoutInflater inflater, ViewGroup container,

```

2. In `onCreateView()`, you will 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.

```
Java
// 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`.

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

4. Return `fragmentFirstLayout` from `onCreateView()`.

```
return fragmentFirstLayout;
```

Here is the whole method and the declaration of `showCountTextView`:

```
Java

TextView showCountTextView;
```

```

@Override

public View onCreateView(

    LayoutInflater inflater, ViewGroup container,

    Bundle savedInstanceState

) {

    // Inflate the layout for this fragment

    View fragmentFirstLayout = inflater.inflate(R.layout.fragment_first,
container, false);

    // Get the count text view

    showCountTextView = fragmentFirstLayout.findViewById(R.id.textview_first);

    return fragmentFirstLayout;

}

```

```

public class FirstFragment extends Fragment {

    private FragmentFirstBinding binding; 1 usage
    TextView showCountTextView; 3 usages

    @Override
    public View onCreateView(
        @NonNull LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState
    ) {

        // Inflate the layout for this fragment
        View fragmentFirstLayout = inflater.inflate(R.layout.fragment_first, container, attachToRoot: false);
        // Get the count text view
        showCountTextView = fragmentFirstLayout.findViewById(R.id.textview_first);

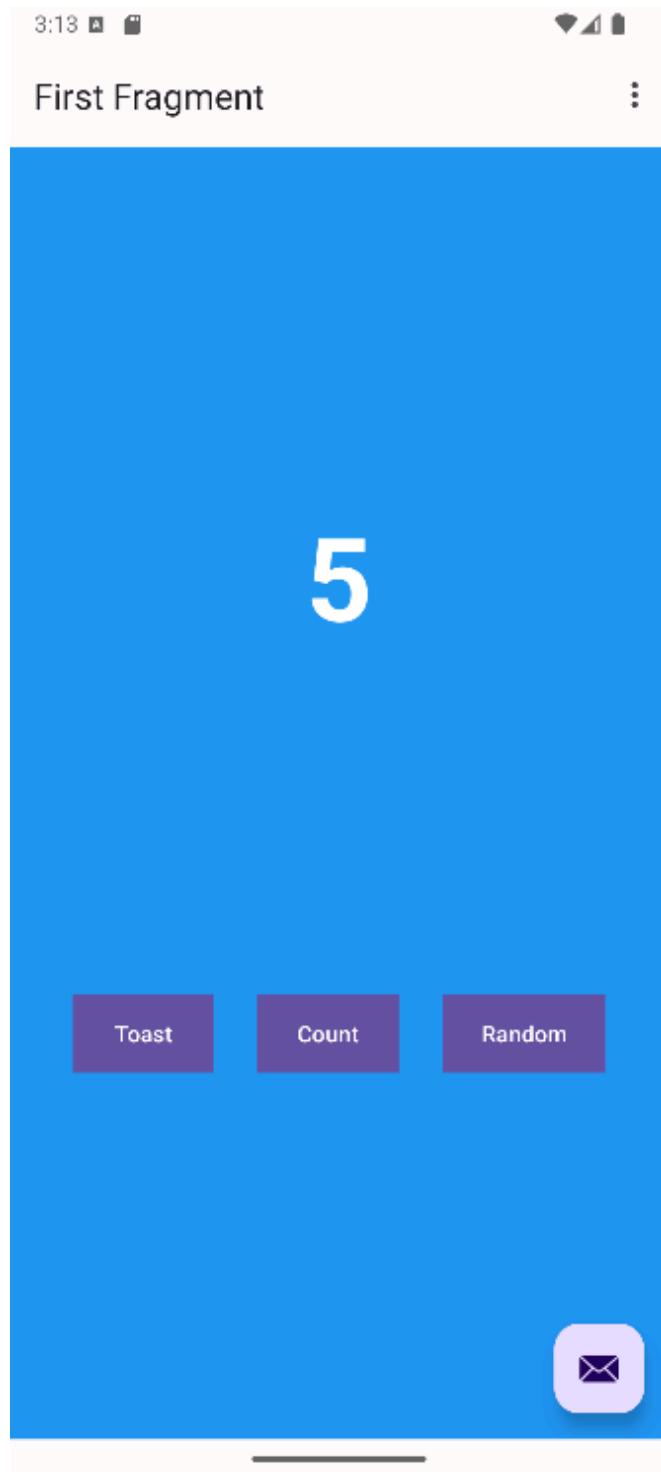
        return fragmentFirstLayout;

    }
}

```

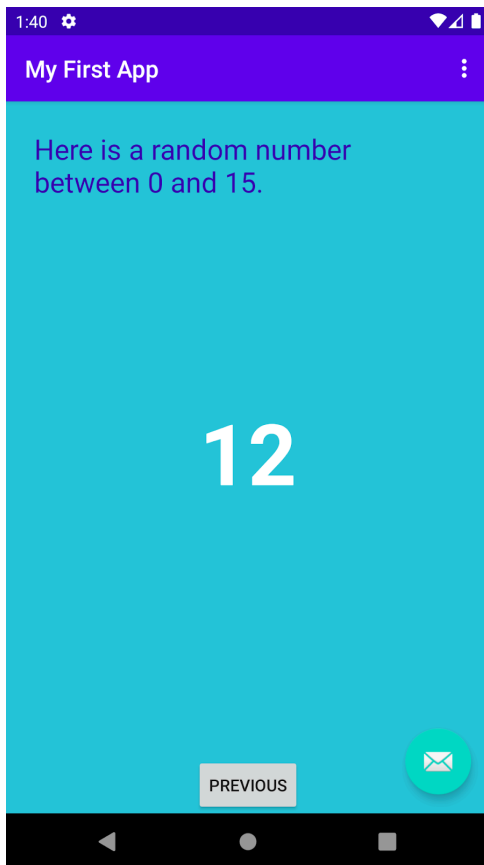
Hint: if your app is no longer able to run after this implementation, try replacing line `View fragmentFirstLayout = inflater.inflate(R.layout.fragment_first, container, false);` with `View fragmentfirstlayout = binding.getRoot;`

5. Run your app. Press the **Count** button and watch the count update.



9. Implement the second fragment

So far, you've focused on the first screen of your app. Next, you will update the Random button to display a random number between 0 and the current count on a second screen.

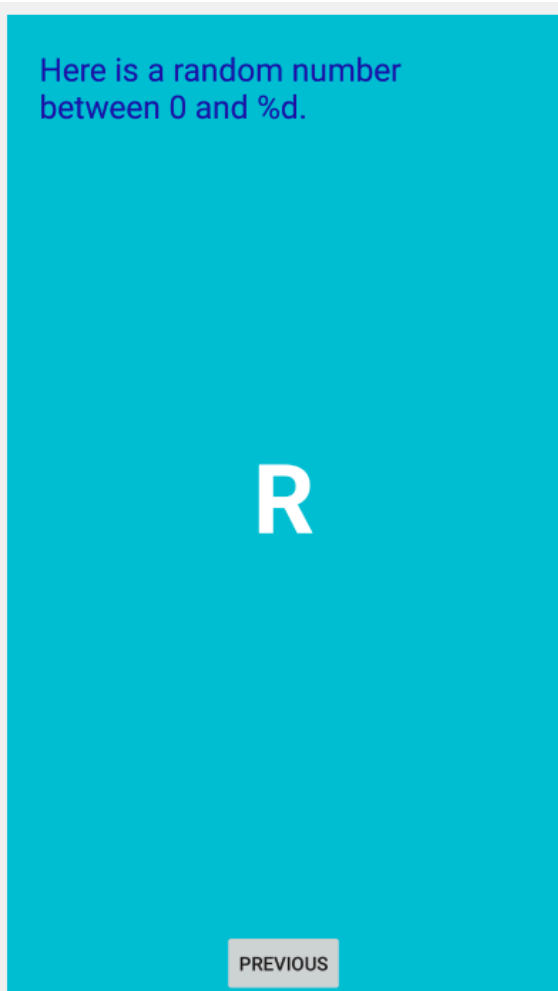


What you'll learn

- How to pass information to a second fragment.

Update the layout for the second fragment

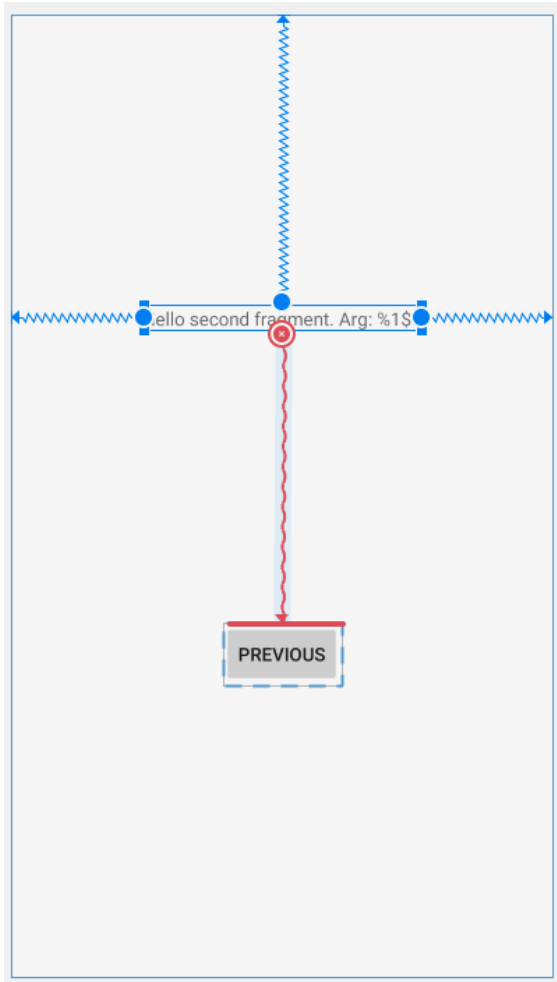
The screen for the new fragment will display a heading title and the random number. Here is what the screen will look like in the design view:



The %d indicates that part of the string will be replaced with a number. The R is just a placeholder.

Step 1: Add a TextView for the random number

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`. Remove the chain constraints between the `TextView` and the `Button` by clicking on them and using the delete key (or right-click to remove them via the context menu).



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.

Set the id to `@+id/textview_random` (textview_random in the Attributes panel.)

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.

Set both width and height to `wrap_content`.

Set the textColor to @android:color/white, set the textSize to 72sp, and the textStyle to bold.

▼ textStyle	🚩 bold
normal	<input type="checkbox"/> false
bold	<input checked="" type="checkbox"/> true
italic	<input type="checkbox"/> false

Set the text to "R". This text is just a placeholder until the random number is generated.

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. 10. If you get a warning "Not Horizontally Constrained," add a constraint from the start of the button to the left side of the screen and the end of the button to the right side of the screen.

Here is the XML code for the TextView that displays the random number:

```
Unset
<TextView
    android:id="@+id/textview_random"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="R"
```

```
android:textColor="@android:color/white"  
android:textSize="72sp"  
android:textStyle="bold"  
app:layout_constraintBottom_toTopOf="@+id/button_second"  
app:layout_constraintEnd_toEndOf="parent"  
app:layout_constraintStart_toStartOf="parent"  
app:layout_constraintTop_toBottomOf="@+id/textview_second"  
app:layout_constraintVertical_bias="0.45" />
```

Step 2: Update the TextView to Display the Header

1. In `fragment_second.xml`, select `textview_second`, which currently has the text `"Hello second fragment. Arg: %1$s"` in the `hello_second_fragment` string resource.
2. If `android:text` isn't set, set it to the `hello_second_fragment` string resource.

Unset

```
android:text="@string/hello_second_fragment"
```

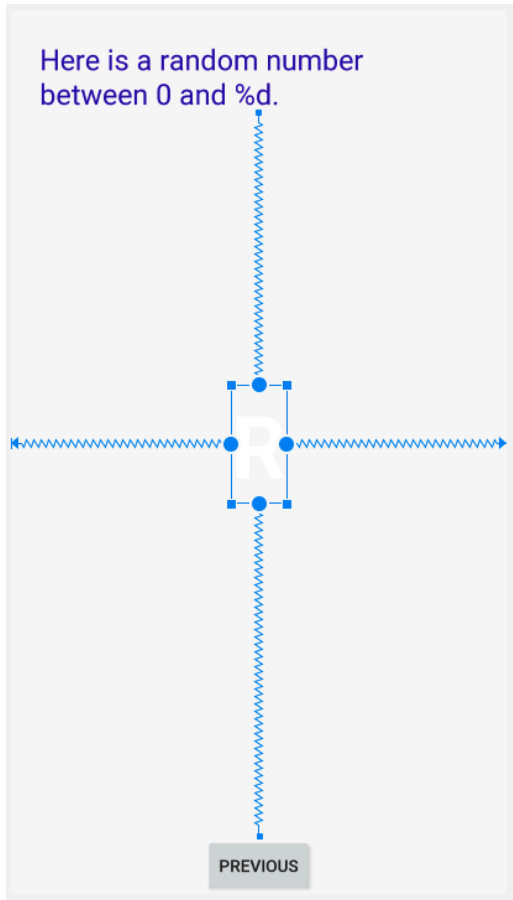
3. Change the id to `textview_header` in the Attributes panel.
4. Set the width to `match_constraint`, but set the height to `wrap_content`, so the height will change as needed to match the height of the content.
5. Set top, left and right margins to `24dp`. Left and right margins may also be referred to as "start" and "end" to support localization for right to left languages.
6. Remove any bottom constraint.
7. Set the text color to `@color/colorPrimaryDark` and the text size to `24sp`.
8. In `strings.xml`, change `hello_second_fragment` to `"Here is a random number between 0 and %d."`

9. Use Refactor > Rename... to change the name of `hello_second_fragment` to `random_heading`.

Here is the XML code for the `TextView` that displays the heading:

Unset

```
<TextView
    android:id="@+id/textview_header"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="24dp"
    android:layout_marginLeft="24dp"
    android:layout_marginTop="24dp"
    android:layout_marginEnd="24dp"
    android:layout_marginRight="24dp"
    android:text="@string/random_heading"
    android:textColor="@color/colorPrimaryDark"
    android:textSize="24sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```



Step 3: Change the background color of the layout

Give your new activity a different background color than the first activity:

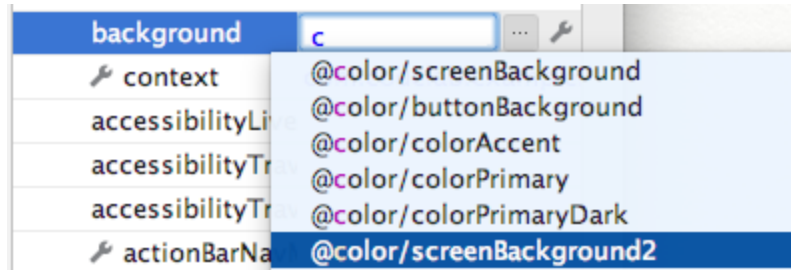
1. In `colors.xml`, add a new color resource:

Unset

```
<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.

In the Attributes panel:



Or in XML:

Unset

```
android:background="@color/screenBackground2"
```

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.

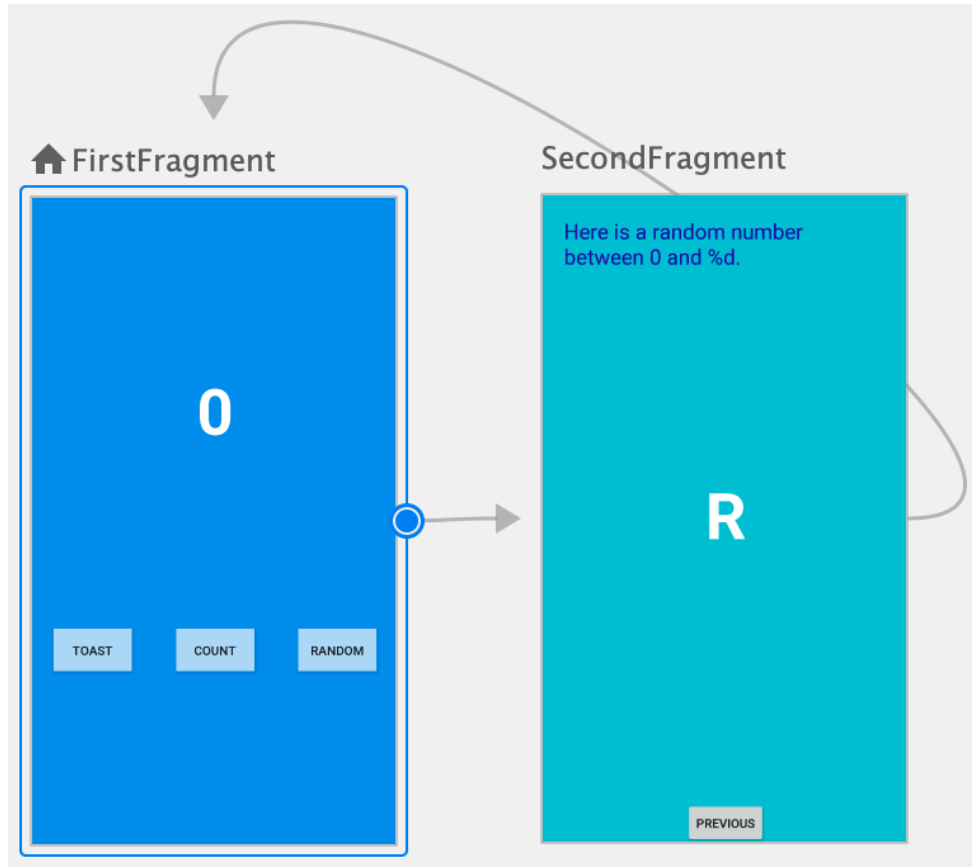
Step 4: Examine the navigation graph

When you created your project, you chose Basic Activity as the template for the new project. When Android Studio uses the Basic 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).

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.



Step 5: Enable SafeArgs

This will enable SafeArgs in Android Studio.

1. Open Gradle Scripts > build.gradle.kts (Project: My First App)
2. Find the `dependencies` section in the `buildscript` section, and add the following lines:

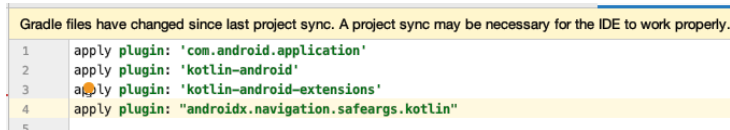
```
Unset
val navVersion = "2.3.0-alpha04"
classpath("androidx.navigation:navigation-safe-args-gradle-plugin:$navVersion")
```

3. Open Gradle Scripts > build.gradle.kts (Module: app)
4. Add the following line to enable SafeArgs under the `plugins` section:

Unset

```
id("androidx.navigation.safeargs")
```

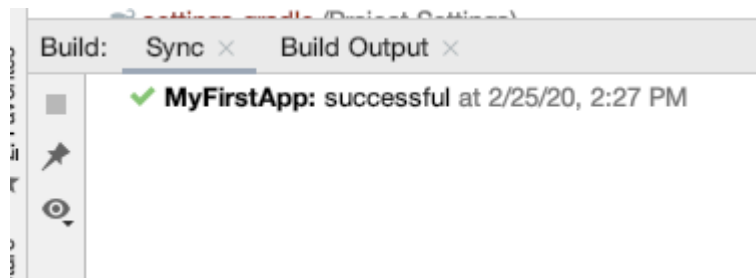
5. Android Studio should display a message about the Gradle files being changed. Click Sync Now on the right hand side.



Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

```
1 apply plugin: 'com.android.application'
2 apply plugin: 'kotlin-android'
3 apply plugin: 'kotlin-android-extensions'
4 apply plugin: "androidx.navigation.safeargs.kotlin"
5
```

After a few moments, Android Studio should display a message in the Sync



tab that it was successful:

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

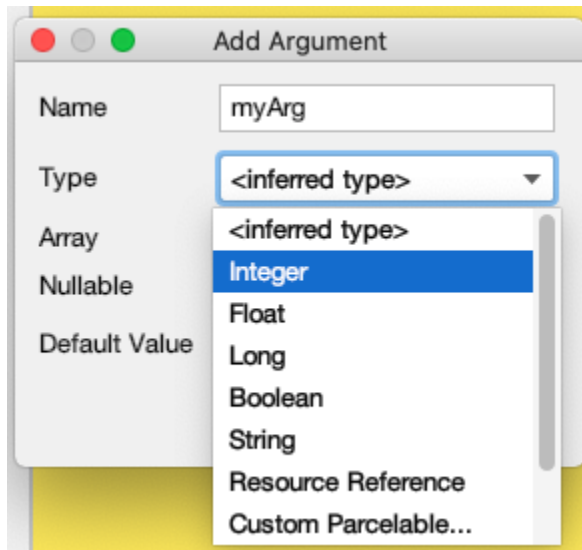
Troubleshooting: If the sync was not successful, confirm that you added the correct lines to the correct Gradle file. If there are still problems, check the developer's guide [about Safe Args](#) for an updated `nav_version` or other changes.

Step 6: 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.



Step 7: 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`.

Java

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

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

Java

```
FirstFragmentDirections.ActionFirstFragmentToSecondFragment action =  
FirstFragmentDirections.actionFirstFragmentToSecondFragment(currentC  
ount);
```

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

Java

```
NavHostFragment.findNavController(FirstFragment.this).navigate(actio  
n);
```

Here is the whole method, including the code you added earlier:

Java

```
public void onCreateView(@NonNull View view, Bundle  
savedInstanceState) {  
    super.onCreateView(view, savedInstanceState);  
  
    view.findViewById(R.id.random_button).setOnClickListener(new  
View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            int currentCount =  
Integer.parseInt(showCountTextView.getText().toString());  
  
            FirstFragmentDirections.ActionFirstFragmentToSecondFragment action =  
            FirstFragmentDirections.actionFirstFragmentToSecondFragment(currentC  
ount);
```

```

NavHostFragment.findNavController(FirstFragment.this).navigate(action);
    }
});

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

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

```

6. 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.

Step 8: Update SecondFragment to compute and display a 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 `SecondFragment.java`, add an import for `navArgs` to the list of imported libraries.

Java

```
import androidx.navigation.fragment.navArgs;
```

2. 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`.

Java

```
Integer count =  
SecondFragmentArgs.fromBundle(getArguments()).getMyArg();  
String countText = getString(R.string.random_heading, count);  
TextView headerView =  
view.getRootView().findViewById(R.id.textView_header);  
headerView.setText(countText);
```

3. Get a random number between 0 and the count.

Java

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

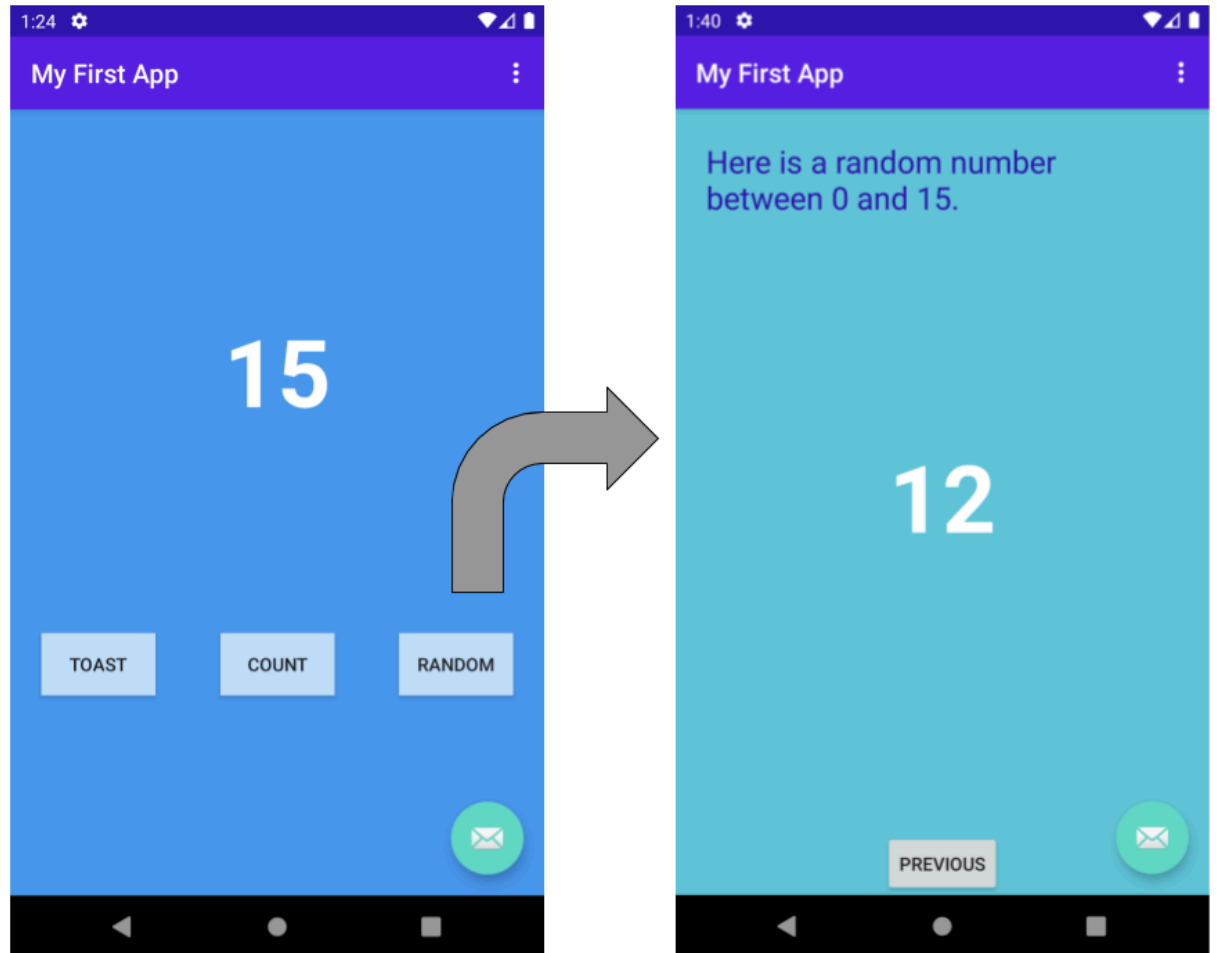
4. Add code to convert that number into a string and set it as the text for `textView_random`.

Java

```
TextView randomView =  
view.getRootView().findViewById(R.id.textView_random);
```

```
randomView.setText(randomNumber.toString());
```

5. Run the app. Press the Count button a few times, then press the Random button. Does the app display a random number in the new activity?



Congratulations, you have built your first Android app!

10. Learn more

The intention of this codelab was to get you started building Android apps. We hope you want to know a lot more though, like how do I save data? How do I run background tasks? How do I display a list of photos? How do I ...

We encourage you to keep learning. We have more Android courses built by Google to help you on your learning journey.

Written tutorials

- [Android Developer Fundamentals](#) teaches programmers to build Android apps. This course is also available in some schools.
- [Kotlin Bootcamp codelabs course](#) is an introduction to **Kotlin** for programmers. You need experience with an object oriented programming language (Java, C++, Python) to take this course..
- Find more at developer.android.com, the official Android developer documentation from Google.

These interactive, video-based courses were created by Google experts in collaboration with Udacity. Take these courses at your own pace in your own time.

- [Developing Android Apps in Kotlin](#): If you know how to program, learn how to build Android apps. This course uses **Kotlin**.
- [Kotlin Bootcamp for Programmers](#): This is an introduction to Kotlin for programmers. You need some experience with an object oriented programming language (Java, C++, Python) to take this course.