CP3406_CP5307 Codelab 2.3: Interactive Dice Roller App

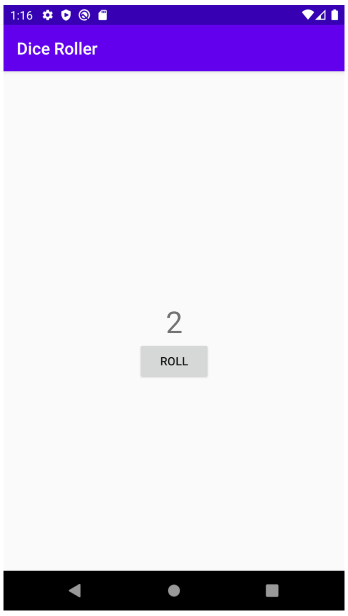
Contents

1. Overview

In this codelab, you'll create a Dice Roller app for Android where users can click on a `Button` from the app and thus launch a dice. The launch result will be shown in an element `TextView` on the screen.

You will use the design **editor** of Android Studio to create the design of your app and then write Kotlin code to set what should happen when you click on the `Button`.

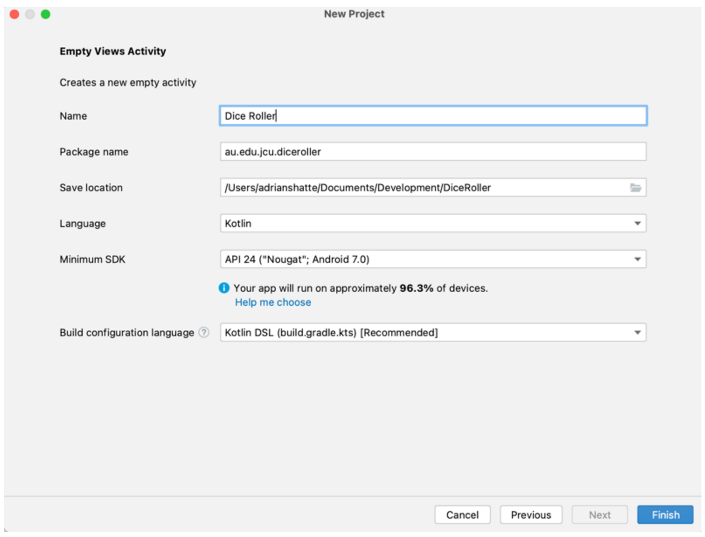
Here's how the app will look like when you've completed this codelab:



2. Configure the app

How to create an empty activity project

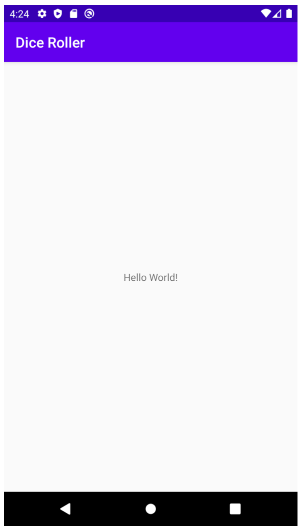
1. If you already have a project open in Android Studio, go to **File > New > New Project...** to open the **Create New Project** screen.
2. On the **Create New Project**, create a new Kotlin project with the **Empty Views Activity** template.
3. The app should be called "Dice Roller" with API level 24 and Kotlin language selected.



Empty Views Activity

Creates a new empty activity

Name	Dice Roller
Package name	au.edu.jcu.diceroller
Save location	/Users/adrianshatte/Documents/Development/DiceRoller
Language	Kotlin
Minimum SDK	API 24 ("Nougat"; Android 7.0)
	Your app will run on approximately **96.3%** of devices. Help me choose
Build configuration language	Kotlin DSL (build.gradle.kts) [Recommended]

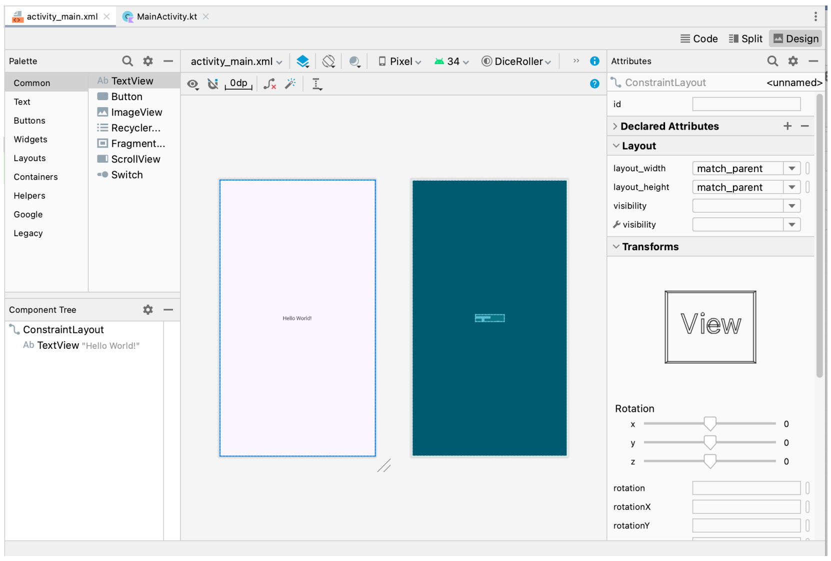


4. Run the new app, which should look like the second image above.

3. How to create the app design

How to open the design editor

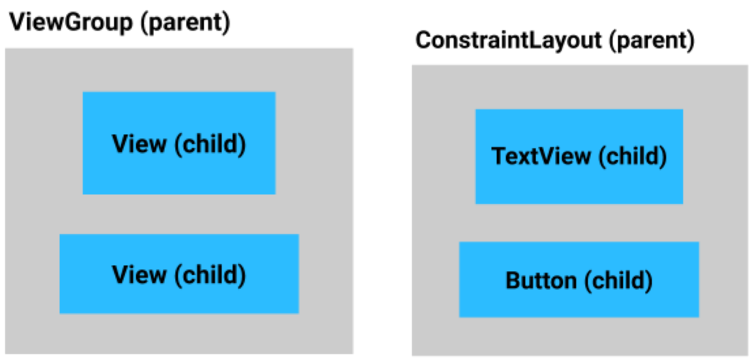
1. In the **Android/Project** window, find the file `activity_main.xml` in the res/layout folder and double-click on it. You should see the **Design Editor** and the `TextView` "Hello World" in the center of the app. Note that the design view may already be open in one of the tabs after creating the new project.



Now you'll add a `Button` to your app. A `Button` is an element of the user interface (UI) on Android that the user can press to perform an action.

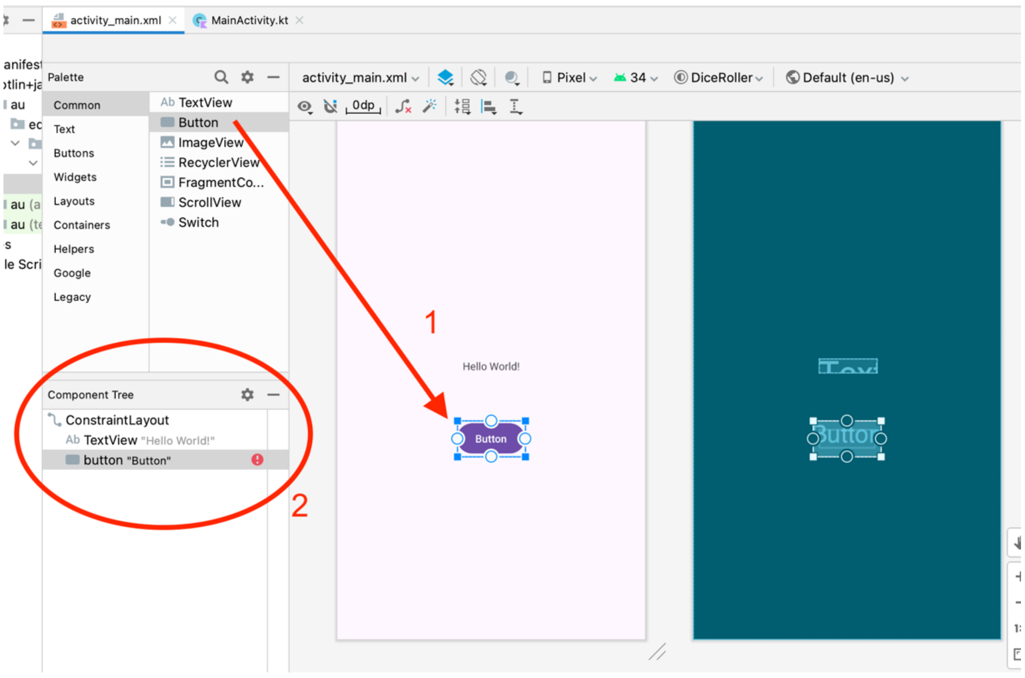
In this task, you'll add a `Button` underneath the `TextView` "Hello World". The elements `TextView` and `Button` will be contained within a `ConstraintLayout`, which is a type of `ViewGroup`.

The Views will be nested inside the ViewGroup, which forms a hierarchical parent-child relationship. You can nest ViewGroups and Views to create complicated layouts, however be careful of nesting too deeply to avoid performance issues. The ConstraintLayout can help us to keep a flat structure and avoid too much nesting.

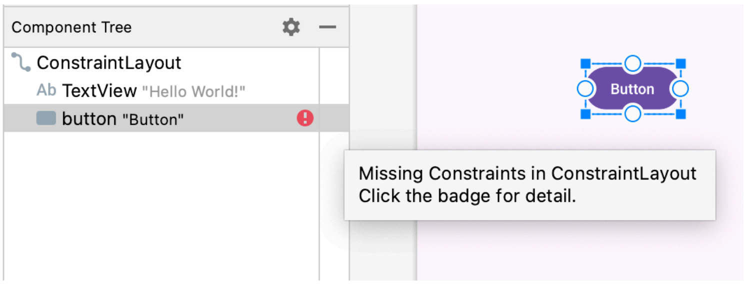


How to add a Button element to the design

1. Drag a `Button` from the **Palette** to the Design view and place it under the element `TextView` from "Hello World."



2. To the left, you will see the Component Tree palette, which shows you an overview of all the components in your design. Check that the `Button` and `TextView` appear nested under `ConstraintLayout` (as secondary elements of `ConstraintLayout`).
3. Note the error indicating that the `Button` has no restrictions. Since the element `Button` is in a `ConstraintLayout`, you must establish vertical and horizontal constraints so that it is positioned properly in relation to the screen/other elements.

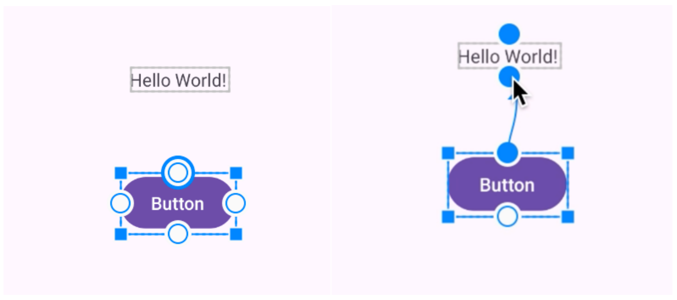


Missing Constraints in ConstraintLayout
Click the badge for detail.

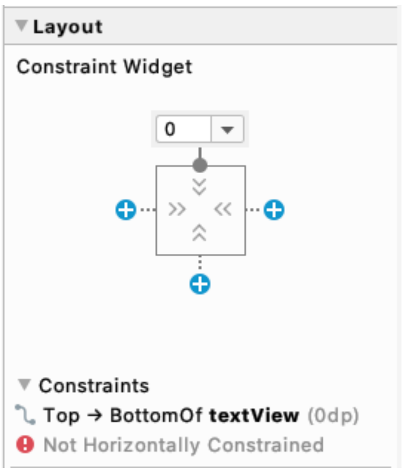
How to position the Button element

In this step, you will add a vertical restriction from the top of the element `Button` up to the bottom of the element `TextView`. This will position the element `Button` underneath `TextView`.

1. In the **Design** view, click on the Button so that it is selected. On the upper edge of the `Button`, click and drag the blue-rimmed white circle, and an arrow will follow. Drag it up to the lower edge of the `TextView` element. "Hello World." This will establish a constraint that causes the `Button` to be positioned just below the `TextView`.

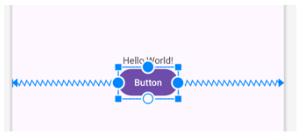


2. Look at the **Attributes** to the right of the **design editor** (with the Button element still selected).
3. In **Constraint Widget,** look at the new design restriction that is set at the bottom of `TextView`, **Top → BottomOf textView (0dp)**. **(0dp)** means there's a margin of 0. You'll also see an error regarding missing horizontal restrictions.

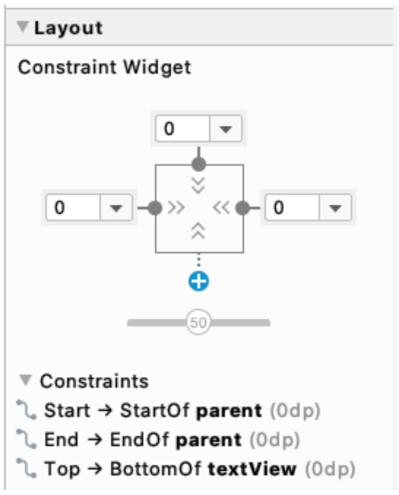


Constraints
Top → BottomOf **textView** (0dp)
Not Horizontally Constrained

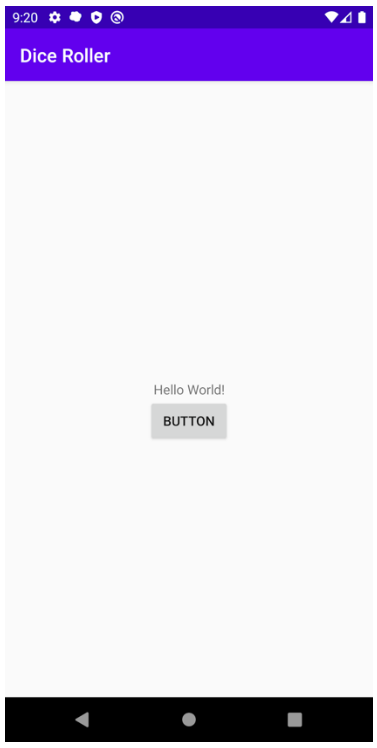
4. Drag a horizontal restriction from the left side of `Button` to the left edge of the screen (the `ConstraintLayout`).
5. Repeat the action on the right side by connecting the right edge of `Button` with the right side of the `ConstraintLayout`. The result should be seen as the following:



6. With `Button` still selected, look at the **Constraint Widget** again. Observe the two additional restrictions that were added: **Start → StartOf parent (0dp)** and **End → EndOf parent (0dp)**. This means the `Button` is horizontally centred within its parent element, `ConstraintLayout`.



Constraints
Start → StartOf **parent** (0dp)
End → EndOf **parent** (0dp)
Top → BottomOf **textView** (0dp)

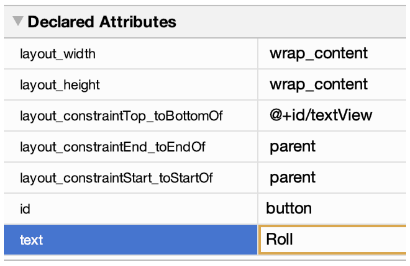


7. Run the app. It should look similar to the screenshot shown to the right. You can click on the `Button` but it won't execute any action yet.

How to change the text of the Button element

You will now make some additional changes to the UI in the **design editor**.

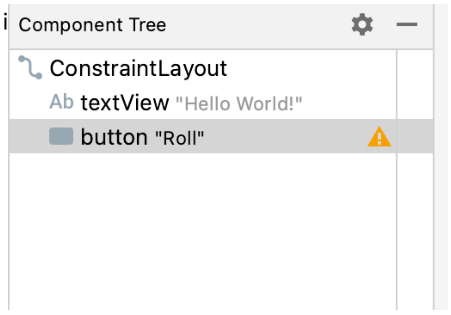
1. In the **design editor**, with `Button` selected, go to **Attributes**, change the **text** field to “**Roll”** and press the key `Enter` (`Return` on a Mac).



Declared Attributes	
layout_width	wrap_content
layout_height	wrap_content
layout_constraintTop_toBottomOf	@+id/textView
layout_constraintEnd_toEndOf	parent
layout_constraintStart_toStartOf	parent
id	button
text	Roll

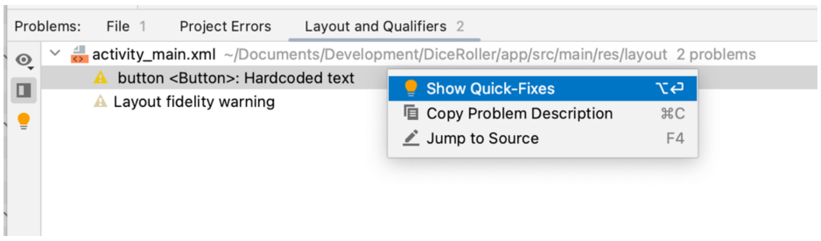
2. In **Component Tree**, you'll see an orange warning triangle next to `Button`. If you place the cursor on the triangle, a message will appear. Android Studio detected a *hardcoded string* ("Roll") in the. It is suggested that you use a *strings resource* instead.

Why not have hardcoded strings? It can be difficult to maintain a large app if you’ve got all your strings spread throughout many files and need to make changes. Also, what if you want to translate your app into different languages? Fortunately, Android Studio has an automatic solution for extracting hardcoded strings and putting them into a single file.



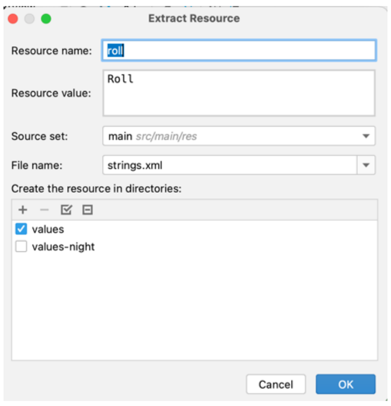
3. In **Component Tree**, double-click on the orange triangle.

The list of problems with your project will be opened.


Problems: File 1 Project Errors Layout and Qualifiers 2
Show Quick-Fixes ⌥↩
Copy Problem Description ⌘C
Jump to Source F4

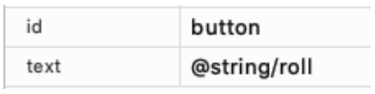
activity_main.xml ~/Documents/Development/DiceRoller/app/src/main/res/layout 2 problems
button <Button>: Hardcoded text
Layout fidelity warning

4. Right-click on the Hardcoded text issue with our Button, and select “Show Quick-Fixes”, then select “Extract string resource”.
5. The **Extract Resource** dialog will be opened. Extracting a string consists of taking the text "Roll" and creating a string resource called `roll` in `strings.xml` (in the folder res/values). The default values are correct, so click **OK**.



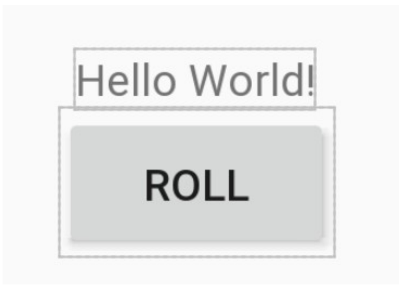
Resource name: roll
Resource value: Roll
Source set: main src/main/res
File name: strings.xml
Create the resource in directories:
[✓ values]
values-night

6. Note that in **Attributes**, the **text** attribute for `Button` Now it says `@string/roll` which refers to the resource you have just created.



id	button
text	@string/roll

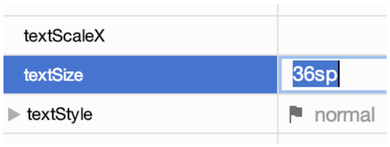
In the **Design** view, the `Button` will still show the text as **Roll**.



How to adjust the TextView style

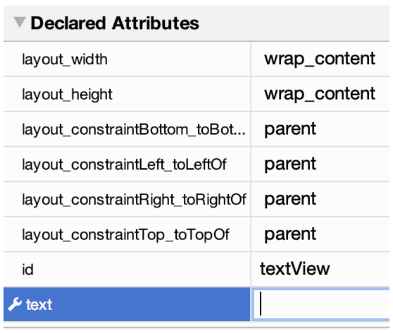
The text "Hello World" is quite small, and the message is not relevant to your app. In this step, you will replace the text "Hello, World" with a number to show the dice roll result. You will also enlarge the font size, so that it is easier to read.

1. In **Design Editor**, select `TextView` so that your attributes appear in the **Attributes** window.
2. Change the **textSize** parameter of `TextView` to **36sp** so that it is big and easy to read. You may need to scroll to find **textSize** (it will be in Common Attributes, within textAppearance).



textScaleX	
textSize	36sp
▶ textStyle	normal

3. Erase the **text** attribute of `TextView`. You don't need to show anything in `TextView` until the user rolls the dice.

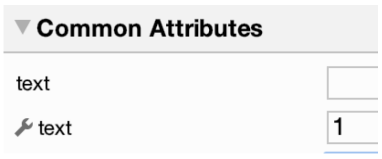


▼ Declared Attributes	
layout_width	wrap_content
layout_height	wrap_content
layout_constraintBottom_toBot...	parent
layout_constraintLeft_toLeftOf	parent
layout_constraintRight_toRightOf	parent
layout_constraintTop_toTopOf	parent
id	textView
text	

However, it is very useful to see text in `TextView` while editing the design and code of your app. For this purpose, you can add text to `TextView` that is only visible in the design preview, but not when the app runs.

4. Select `TextView` in **Component Tree**.
5. In **Common Attributes**, look for the **text** attribute and, underneath it, another **text** with a tool icon. The **text** attribute is what will be shown to the user when the app runs. The **text** attribute with the tool icon is the "tools text" attribute that only you as a developer can see.
6. Configure the text of this attribute as "1" in `TextView` (to pretend that the launch result was 1). This

"1" number will only appear in the Design **Editor** of Android Studio, but it will not when you run the app on a real device or emulator.

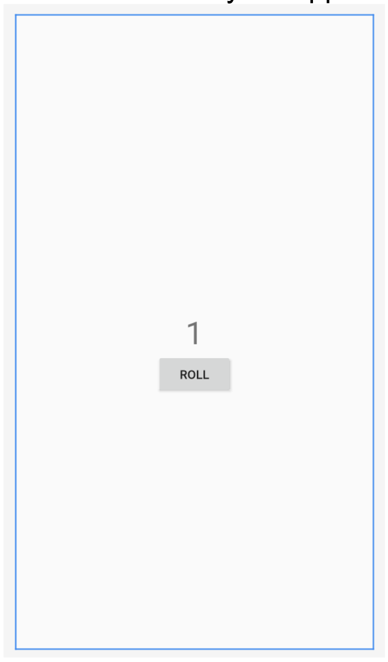

Common Attributes

text

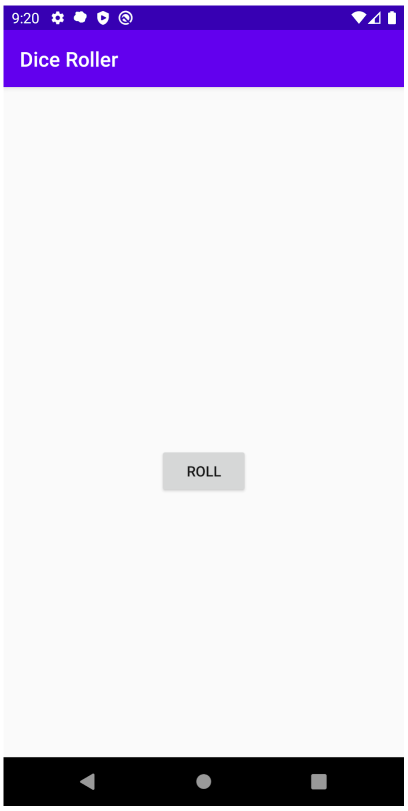
text 1

Keep in mind that despite this being a hardcoded string, only app developers will see this text, so you don't need to create a strings.xml resource for it.

7. Check out your app in the preview. It should now show "1."



8. Run the app in the emulator. Note that the value "1" is not shown. This is the expected behaviour.



Excellent. You're done with the design changes.

You have an app with a button, but if you press it, nothing happens. To change this, you must write Kotlin code that launches the dice and updates the screen when the button is pressed.

To make this change, you need to understand a little more the structure of the Android apps.

4. Introduction to activities

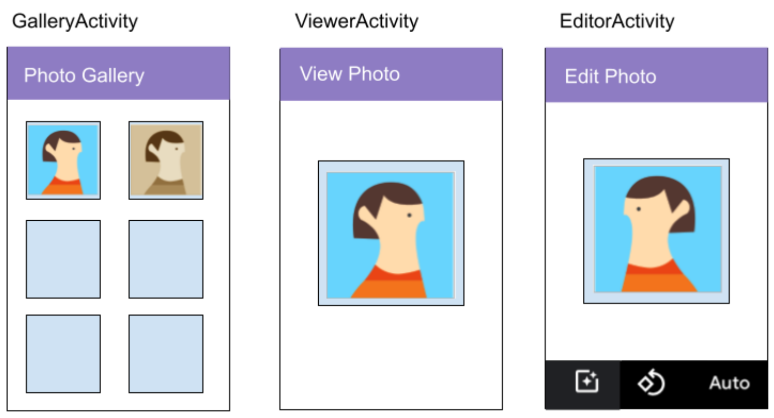
An `Activity` provides the window in which your app draws your UI. Usually, an `Activity` occupies the entire screen of your running app. Each app has one or more activities. Often, the first activity is called `MainActivity`, and this is usually created for you as part of the project template. For example, when the user travels through the list of apps on their device and presses the "Dice Roller" app icon, the Android system will start the `MainActivity` from the app to launch it.

In the code of `MainActivity`, you must provide details about the design of `Activity` and the way the user should interact with it.

- In the birthday card app, there's an element `Activity` showing the image and the birthday message.
- In the Dice Roller app, there's an element `Activity` showing the design of `TextView` and `Button` you just compiled.

In the case of more complicated apps, there may be several screens and more than one `Activity`. Every `Activity` has a specific purpose.

For example, in a photo gallery app, you can have a `Activity` to show one grid of photos, another `Activity` to see a single photo and a third `Activity` to edit a single photo.



How to open MainActivity.kt file

We need to add code to respond when the user presses the button on `MainActivity`. First, let’s explore the code of `MainActivity` that was created in our app.

5. Navigate to the file `MainActivity.kt` and open it. Below is what you should see:

```
package com.example.diceroller


import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)


   }
}
```

You don't need to understand every single word of the code above yet, but you do have to have a general idea of what it does. The more you work with the Android code, the more familiar you will become and the more you'll understand.

6. Note that this file contains a Kotlin class named `MainActivity`, identified by the keyword `class`.

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

7. Note that there is no function `main()` on your `MainActivity`.

Previously, you learned that every Kotlin program should have a function `main()`. Android apps work differently. Instead of calling for a function `main()` Android system will call the method `onCreate()` from `MainActivity` when the app first opens.

8. Look for the method `onCreate()`, which is similar to the following code:

```
   override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
   }
```

You'll learn about `override` in another codelab (you should have a general idea what it means from prior programming experience). The rest of the method `onCreate()` configures the element `MainActivity` by means of the import code and the initial design configuration with `setContentView()`.

9. Observe the lines that begin with `import`.

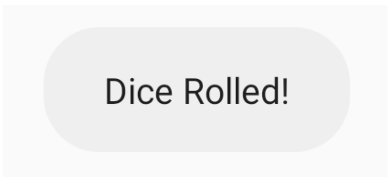
Android provides a multi-class *framework* to facilitate the writing of Android apps, but needs to know specifically which classes are required. You can determine the kind of framework that will be used in your code by using `import`. For example, the class `Button` is defined in `android.widget.Button`.

5. How to make the Button element interactive

Now that you know a little more about the element `MainActivity`, you will modify the app so that something happens on the screen when clicking on the `Button`.

How to show a message when you click on the Button element

In this step, you will specify that a short message will appear at the bottom of the screen when you click the button.



1. Add the following code to the method `onCreate()` after the call `setContentView()`. The method `findViewById()` looks for the element `Button` in the design. `R.id.button` is the resource ID of the `Button`, which is a unique identifier for this element.

```
val rollButton: Button = findViewById(R.id.button)
```

Note: Android automatically assigns ID numbers to your app's resources. For example, the **Roll** button has a resource ID, and the text string of the button also has one. Resource IDs have the format `R.<type>.<name>`. For example, `R.string.roll`.

The code will store a reference to the object `Button` in a variable called `rollButton`.

The method `onCreate()` should now be:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)
   val rollButton: Button = findViewById(R.id.button)
}
```

2. Verify that Android Studio has automatically added an `import` for the class `Button`.

```
import android.os.Bundle
import android.widget.Button
import androidx.appcompat.app.AppCompatActivity
```

Note: If the automatic import does not work, `Button` will stand out in red with an error. You can add the correct import manually if you place the text cursor within the word `Button` and then you press `Alt+Enter` (`Option+Enter` if you use Mac).

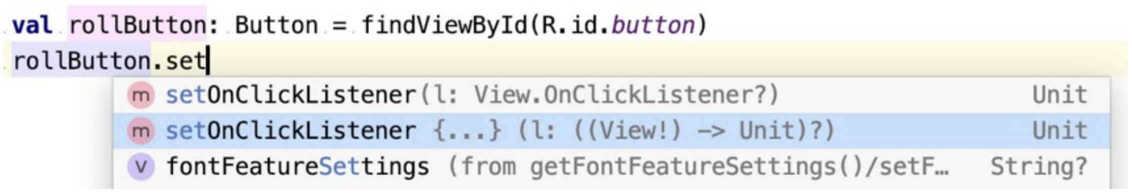
Now you must associate some code with the element `Button` so that something happens when it is pressed. An *onClickListener object* serves to determine what should happen when something is pressed or clicked.

3. Use the object `rollButton` and configure a listener by calling `setOnClickListener()`. Instead of the parentheses that follow the name of the method, you will use curly brackets. This is a special syntax to declare a Lambda in Kotlin.

Whatever you put inside the braces will comprise the instructions of what will happen when the button is pressed. In the next step, your app will display a Toast, which is a short message.

```
rollButton.setOnClickListener {
}
```

As you write, Android Studio can show several suggestions. For this case, choose the **setOnClickListener** option**.**



```
val rollButton: Button = findViewById(R.id.button)
rollButton.set
m setOnClickListener(l: View.OnClickListener?)                Unit
m setOnClickListener {...} (l: ((View!) -> Unit)?)            Unit
v fontFeatureSettings (from getFontFeatureSettings()/setF...  String?
```

Inside the braces, add the instructions of what should happen when the button is pressed. For now, your app will show a `Toast`, which is a short message shown to the user.

4. Call `Toast.makeText()` to create a `Toast` with the text `"Dice Rolled!"`.

```
val toast = Toast.makeText(this, "Dice Rolled!", Toast.LENGTH_SHORT)
```

5. Then call the method `show()` so that the `Toast` appears on the screen.

```
toast.show()
```

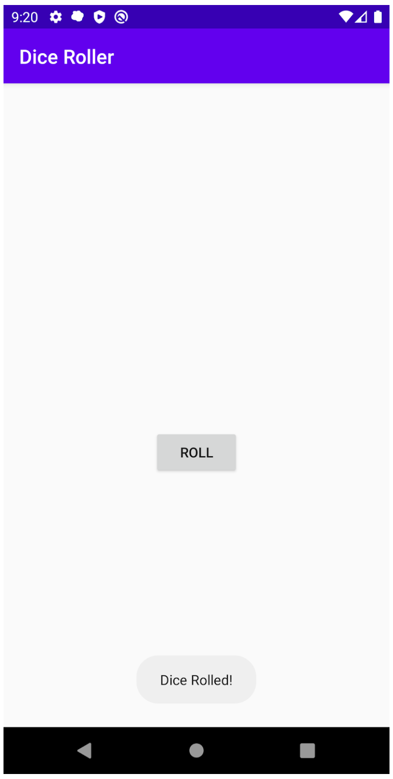
That's what your updated `MainActivity` looks like. Note that the `package` and `import` lines are not shown below, but should still be in the file:

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)
       val rollButton: Button = findViewById(R.id.button)
       rollButton.setOnClickListener {
           val toast = Toast.makeText(this, "Dice Rolled!", Toast.LENGTH_SHORT)
           toast.show()
       }
   }
}
```

You can combine the two lines of the Toast call without a variable. This is a common pattern to simplify your code.

```
Toast.makeText(this, "Dice Rolled!", Toast.LENGTH_SHORT).show()
```

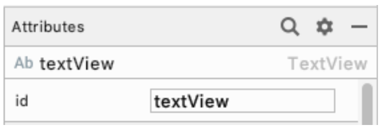
6. Run the app and click the **Roll** button. At the bottom of the screen, the Toast message should appear in a pop-up and disappear after a short period.



How to update TextView when you click on the Button element

Instead of showing a `Toast`, we want the app to update the `TextView` on the screen when you click on the **Roll** button.

1. Go back to `activity_main.xml`
2. Click on the item `TextView`.
3. Notice that the **id** is **textView**.



id: textView

4. Open up. `MainActivity.kt`
5. Delete the lines of code that create and show the `Toast`.

```
rollButton.setOnClickListener {
}
```

6. Create a new variable called `resultTextView` to store `TextView`.
7. Use `findViewById()` to store a reference to the `textView` in the variable.

```
val resultTextView: TextView = findViewById(R.id.textView)
```

8. Set the text of `resultTextView` to be "6" (it's a String, hence the quotatin marks).

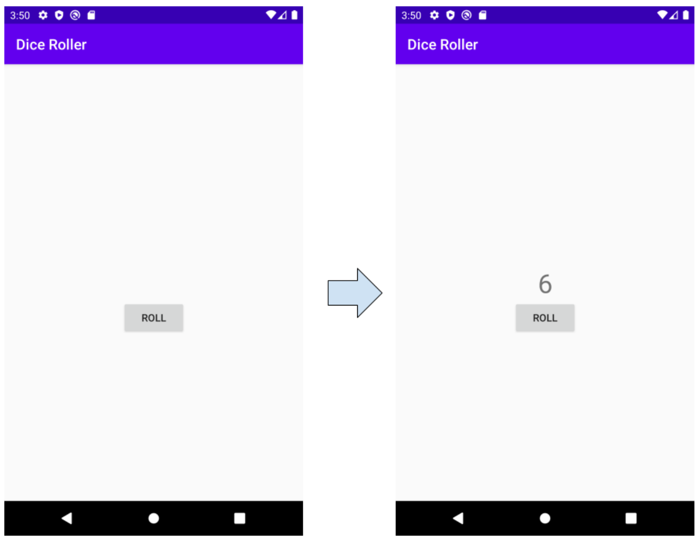
```
resultTextView.text = "6"
```

This is similar to what you did when you set the **text** element in **Attributes**, but now it's in your code, so the text should be in double quotes. Configuring it explicitly means that, for the time being, `TextView` will always show 6. In the next task, you'll add the code to launch the dice and show different values.

The class `MainActivity` should look like this:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val rollButton: Button = findViewById(R.id.button)
        rollButton.setOnClickListener {
            val resultTextView: TextView = findViewById(R.id.textView)
            resultTextView.text = "6"
        }
    }
}
```

9. Run the app and click the button. It should update the `TextView` to show "6".



6. How to add the dice launch logic

The only thing missing is for the dice to be thrown.

How to add the Dice class

1. Within the scope of the class `MainActivity`, create the inner class `Dice` with a method `roll()`.

```
class Dice(val numSides: Int) {

       fun roll(): Int {

       return (1..numSides).random()
    }

}
```

2. Notice that Android Studio will highlight `numSides` with a wavy grey line (it may take a moment to appear.)
3. Place the cursor over `numSides` and a pop-up window will appear with the message **Property 'numSides' could be private**.

If you change `numSides` to be `private`, then you can only access it from the class `Dice`. Since the only code you'll use `numSides` with is in the class `Dice`, it's okay to make this argument `private` for the class `Dice`.

4. To make the suggested correction on Android Studio, click **Make 'numSides' 'private'** in the pop up.

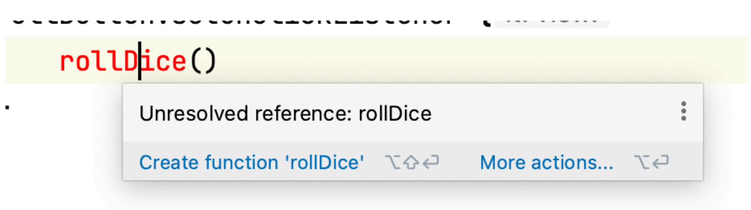
How to create a rollDice() method stub

Now you've added a class `Dice` to your app, you must update `MainActivity` to use it. In order to better organize your code, place all the logic about the dice roll in a function.

1. Replace the code of the click listening object that sets the text in "6" by a call to `rollDice()`.

```
rollButton.setOnClickListener {
   rollDice()
}
```

2. Since `rollDice()` is not yet defined, Android Studio will mark an error and show `rollDice()` in red.
3. If you put the cursor on `rollDice()`, Android Studio will show the problem and some possible solutions.

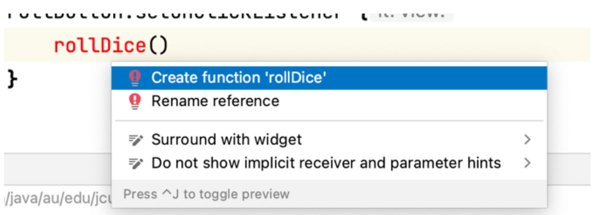


```
rollDice()
```

Unresolved reference: rollDice

4. Click **More actions** to open the menu. Android Studio offers to do more work for you.

Tip: If you find it difficult to place the cursor and then click **on More actions...**, you can click on `rollDice()` and press `Alt+Enter`(`Option+Enter`if you use Mac) to access the menu.



```
    rollDice()
}
```

Create function 'rollDice'
Rename reference
Surround with widget
Do not show implicit receiver and parameter hints
Press ^J to toggle preview

5. Select **Create function 'rollDice'**. Android Studio will create an empty definition for function within `MainActivity`.

```
private fun rollDice() {
    TODO("Not yet implemented")
}
```

Create an instance of the Dice object and roll it

In this step, you'll make the method `rollDice()` roll a dice. Then, the result will be shown in the `TextView`.

1. Inside `rollDice()` erase the call. `TODO()`.
2. Add code to create a 6-sided dice.

```
val dice = Dice(6)
```

3. Call the method `roll()` in order to launch the dice and save the result in a variable called `diceRoll`.

```
val diceRoll = dice.roll()
```

4. Call `findViewById()` to look for the item `TextView`.

```
val resultTextView: TextView = findViewById(R.id.textView)
```

The variable `diceRoll` is a number, but `TextView` uses text. You can use the method `toString()` on `diceRoll` for the purpose of turning it into a string.

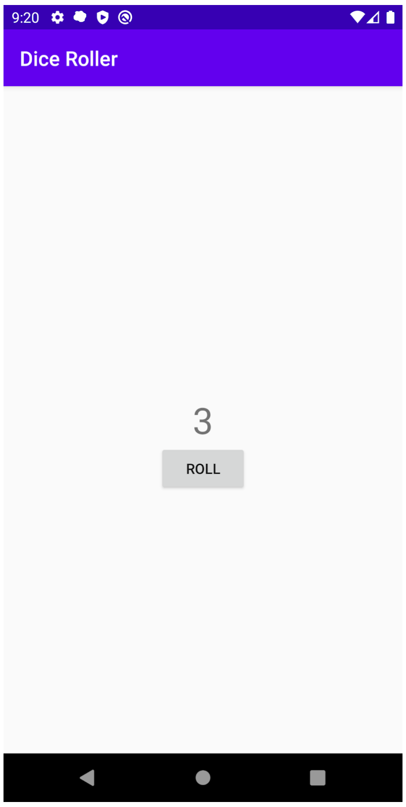
5. Convert `diceRoll` to a string and use it to update the text of `resultTextView`.

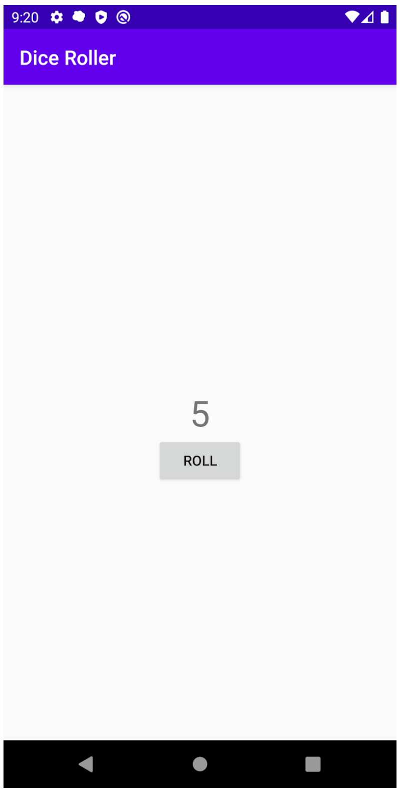
```
resultTextView.text = diceRoll.toString()
```

The complete method of `rollDice()` should look as follows:

```
private fun rollDice() {
    val dice = Dice(6)
    val diceRoll = dice.roll()
    val resultTextView: TextView = findViewById(R.id.textView)
    resultTextView.text = diceRoll.toString()
}
```

6. Run the app. The outcome of the dice launch should change to other values besides 6. As this is a random number between 1 and 6, the value 6 can sometimes appear as well.





7. How to implement coding best practices

It's normal for your code to look a little messy after you've adjusted some parts to make your app work. Before you let it be, you should do some simple cleaning tasks. In this way, the app will be in good condition and will be easier to maintain it in the future.

These habits are the ones practiced by professional Android developers when they write their code.

Android style guide

When you work as a team, the ideal is for members to write code in a similar way so that the code maintains consistency. Therefore, Android has a style guide to write Android code (with name conventions, format guidelines and other best practices). Use these guidelines when you write Android code: Kotlin Style Guide to Android developers.

Here are some ways to comply with the style guide.

Clean the code

Reduce the code

You can make your code more concise if you summarize it on a smaller number of lines. For example, this is the code that configures the click listener in the element `Button`.

```
rollButton.setOnClickListener {
    rollDice()
}
```

Because the instructions within the listener occupy only 1 line, you can place both the method call `rollDice()` and the braces on one line.

```
rollButton.setOnClickListener { rollDice() }
```

Re-format the code

Now, you'll re-format your code to make sure it complies with the recommended code format conventions for Android.

1. In class `MainActivity.kt`, select the entire text from the file with the key combination `Control+A` on Windows (or `Command+A` in Mac). You can also go to the Android Studio **Edit** menu**.**
2. Once you have all the text selected in the file, go to the Android Studio **Code** menu **and reformat Code** or use the key combination `Ctrl+Alt+L` (or `Command+Option+L` in Mac).

That updates the format of your code, including blank lines and indentations, among others. You may not see any changes, and that's good. It means your code already is in a clean format.

Comment on the code

Add some comments to the code to describe what will happen. As the code becomes more complicated, it's also important to take note of *the reason* you wrote it to work the way you did. If you later change the code, *what* the code *does* will remain clear, but you may not remember why you wrote it as you did.

It is common to add a comment for each class (`MainActivity` and `Dice` are the only ones you have in this app) and every method you write. Use the symbols `/**` and `*/`at the beginning and end of the commentary to tell the system that this is not code. When the system runs the code, these lines will be omitted.

Example of a comment in a class:

```
/**
* This activity allows the user to roll a dice and view the result
* on the screen.
*/
class MainActivity : AppCompatActivity() {
```

Example of a comment in a method:

```
/**
* Roll the dice and update the screen with the result.
```

```
*/
private fun rollDice() {
```

Within a method, you can add comments if you think that will help your code reader. Remember that you can use the symbol `//`at the beginning of your comment. Everything that follows the symbol `//` in a line will be considered a commpent.

Example of 2 comments within a method:

```
private fun rollDice() {
   // Create new Dice object with 6 sides and roll it
   val dice = Dice(6)
   val diceRoll = dice.roll()
   // Update the screen with the dice roll
   val resultTextView: TextView = findViewById(R.id.textView)
   resultTextView.text = diceRoll.toString()
}
```

1. Take a few minutes to add some comments to your code.
2. As a result of all these changes in comments and formatting, we recommend that you run the app again to make sure it continues to work as expected.

8. Summary

- Add a `Button`in an Android app using the **design editor**.
- Modify the class `MainActivity.kt`to add interactive behavior to the app.
- Show a message `Toast`emerge as a temporary solution to the effects of verifying that you are on the right track.
- Configure a click listening object for an item `Button`by `setOnClickListener()`in order to add the behavior you should have when clicking on an item `Button`.
- When the app is running, you can update the screen by calling the methods in `TextView`, the `Button`or other elements of the UI design.
- Comment on your code so that you help people who read it understand your approach.
- Change the code format and clean your code.