Name: Nguyễn Văn Cường – Student ID: 20215006

Report – Week 5

4.3. Classes and object instances in Kotlin

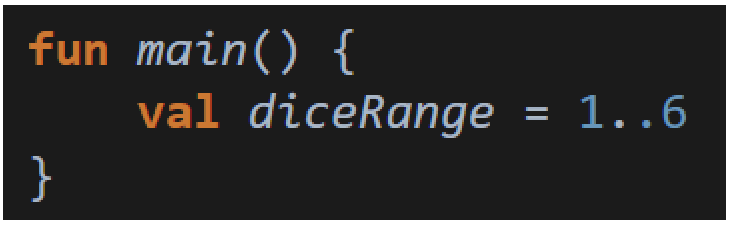
Roll random numbers

Set up your starter code

1. In your browser, open the website https://developer.android.com/training/kotlinplayground.
2. Delete all the existing code in the code editor and replace it with the code below. This is the main() function you worked with in earlier codelabs. ***fun main() { }***

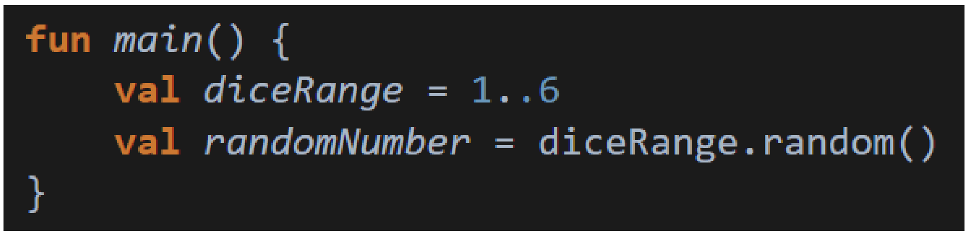
Use the random function

1. Inside your main() function, define a variable as a val called diceRange. Assign it to an IntRange from 1 to 6, representing the range of integer numbers that a 6-sided dice can roll.



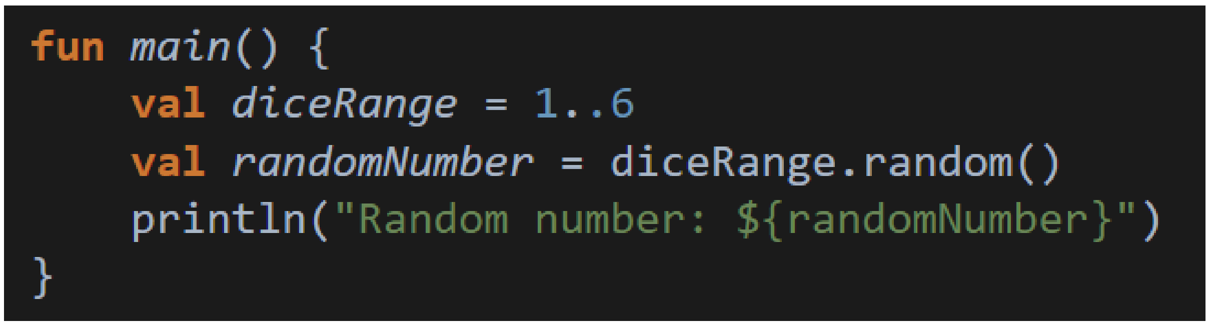
```kotlin
fun main() {
    val diceRange = 1..6
}
```

2. Inside main(), define a variable as a val called randomNumber.
3. Make randomNumber have the value of the result of calling random() on the diceRange range, as shown below.



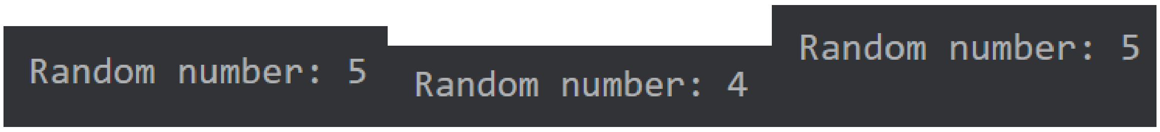
```kotlin
fun main() {
    val diceRange = 1..6
    val randomNumber = diceRange.random()
}
```

4. To see your randomly generated number, use the string formatting notation (also called a "string template") ${randomNumber} to print it, as shown below.



```kotlin
fun main() {
    val diceRange = 1..6
    val randomNumber = diceRange.random()
    println("Random number: ${randomNumber}")
}
```

5. Run your code several times. Each time, you should see output as below, with different random numbers.



```
Random number: 5
```

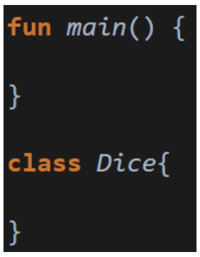
```
Random number: 4
```

```
Random number: 5
```

Create a Dice class

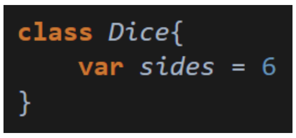
Define a Dice class

1. To start afresh, clear out the code in the main() function so that you end up with the code as shown below. ***fun main() { }***
2. Below this main() function, add a blank line, and then add code to create the Dice class. As shown below, start with the keyword class, followed by the name of the class, followed by an opening and closing curly brace. Leave space in between the curly braces to put your code for the class.



```kotlin
fun main() {

}

class Dice{

}
```

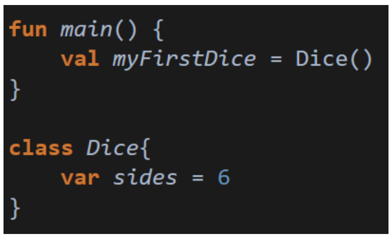
3. Inside the Dice class, add a var called sides for the number of sides your dice will have. Set sides to 6



```kotlin
class Dice{
    var sides = 6
}
```

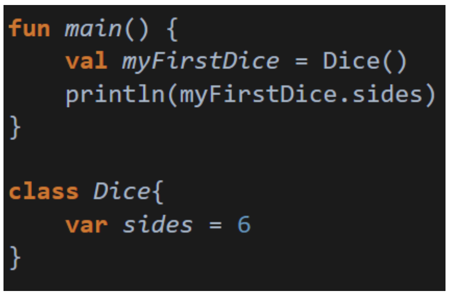
Create an instance of the Dice class

1. To create an object instance of Dice, in the main() function, create a val called myFirstDice and initialize it as an instance of the Dice class. Notice the parentheses after the class name, which denote that you are creating a new object instance from the class.



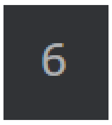
```kotlin
fun main() {
    val myFirstDice = Dice()
}

class Dice{
    var sides = 6
}
```

2. Below the declaration of myFirstDice, add a println() statement to output the number of sides of myFirstDice.



```kotlin
fun main() {
    val myFirstDice = Dice()
    println(myFirstDice.sides)
}

class Dice{
    var sides = 6
}
```

3. Run your program and it should output the number of sides defined in the Dice class.



```
6
```

Make the Dice Roll

1. In the Dice class, below the sides variable, insert a blank line and then create a new function for rolling the dice. Start with the Kotlin keyword fun, followed by the name of the method, followed by parentheses (), followed by opening and closing curly braces {}. You can leave a blank line in between the curly braces to make room for more code, as shown below. Your class should look like this.

```kotlin
class Dice {
    var sides = 6

    fun roll() {

    }
}
```

2. Inside the roll() method, create a val randomNumber. Assign it a random number in the 1..6 range. Use the dot notation to call random() on the range.

```kotlin
fun roll() {
    val randomNumber = (1..6).random()
}
```

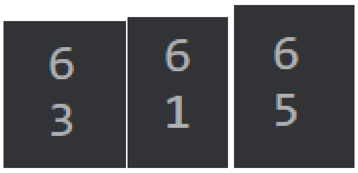
3. After generating the random number, print it to the console. Your finished roll() method should look like the code below.

```kotlin
fun roll() {
    val randomNumber = (1..6).random()
    println(randomNumber)
}
```

4. To actually roll myFirstDice, in main(), call the roll() method on myFirstDice. You call a method using the "dot notation". So, to call the roll() method of myFirstDice, you type myFirstDice.roll() which is pronounced "myFirstDice dot roll()".

```kotlin
fun main() {
    val myFirstDice = Dice()
    println(myFirstDice.sides)
    myFirstDice.roll()
}

class Dice {
    var sides = 6

    fun roll() {
        val randomNumber = (1..6).random()
        println(randomNumber)
    }
}
```

5. Run your code! You should see the result of a random dice roll below the number of sides.



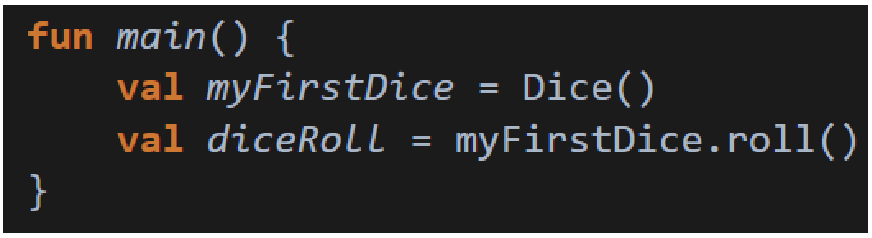
```
6
3
```

```
6
1
```

```
6
5
```

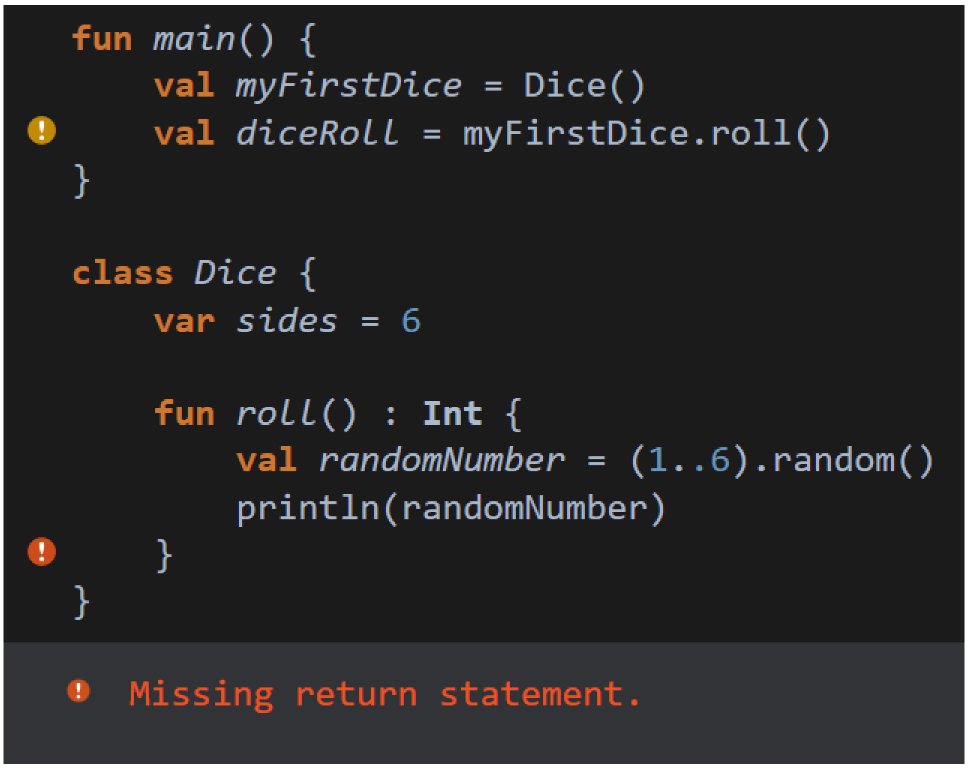
Return your dice roll's value

1. In main() modify the line that says myFirstDice.roll(). Create a val called diceRoll. Set it equal to the value returned by the roll() method.



```kotlin
fun main() {
    val myFirstDice = Dice()
    val diceRoll = myFirstDice.roll()
}
```

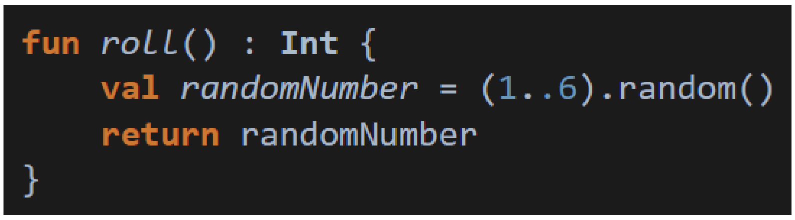
2. Change the roll() function to specify what type of data will be returned. In this case, the random number is an Int, so the return type is Int. The syntax for specifying the return type is: After the name of the function, after the parentheses, add a colon, space, and then the Int keyword for the return type of the function. The function definition should look like the code below.
3. Run this code. You will see an error in the Problems View. It says:



```kotlin
fun main() {
    val myFirstDice = Dice()
    val diceRoll = myFirstDice.roll()
}

class Dice {
    var sides = 6

    fun roll() : Int {
        val randomNumber = (1..6).random()
        println(randomNumber)
    }
}
```

```
Missing return statement.
```

4. In roll(), remove the println() statement and replace it with a return statement for randomNumber. Your roll() function should look like the code below.



```kotlin
fun roll() : Int {
    val randomNumber = (1..6).random()
    return randomNumber
}
```

5. In main() remove the print statement for the sides of the dice.
6. Add a statement to print out the value of sides and diceRoll in an informative sentence. Your finished main() function should look similar to the code below.

```
fun main() {
    val myFirstDice = Dice()
    val diceRoll = myFirstDice.roll()
    println("Your ${myFirstDice.sides} sided dice rolled ${diceRoll}!")
}
```

7. Run your code and your output should be like this.

```
Your 6 sided dice rolled 6!
```

Change the number of sides on your dice

1. In your Dice class, in your roll() method, change the hard-coded 1..6 to use sides instead, so that the range, and thus the random number rolled, will always be right for the number of sides.

```
fun roll() : Int {
    val randomNumber = (1..sides).random()
    return randomNumber
}
```

2. In the main() function, below and after printing the dice roll, change sides of myFirstDice to be set to 20.
3. Copy and paste the existing print statement below after where you changed the number of sides.
4. Replace the printing of diceRoll with printing the result of calling the roll() method on myFirstDice.

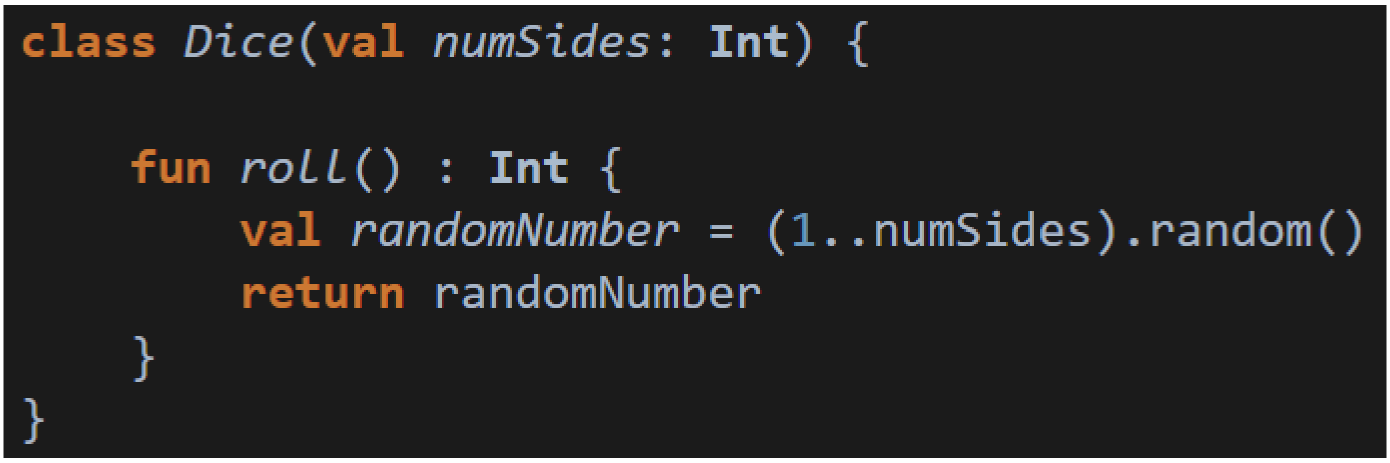
```
fun main() {
    val myFirstDice = Dice()
    val diceRoll = myFirstDice.roll()
    println("Your ${myFirstDice.sides} sided dice rolled ${diceRoll}!")

    myFirstDice.sides = 20
    println("Your ${myFirstDice.sides} sided dice rolled ${myFirstDice.roll()}!")
}
```

5. Run your program and you should see a message for the 6-sided dice, and a second message for the 20-sided dice.

```
Your 6 sided dice rolled 2!
Your 20 sided dice rolled 1!
```

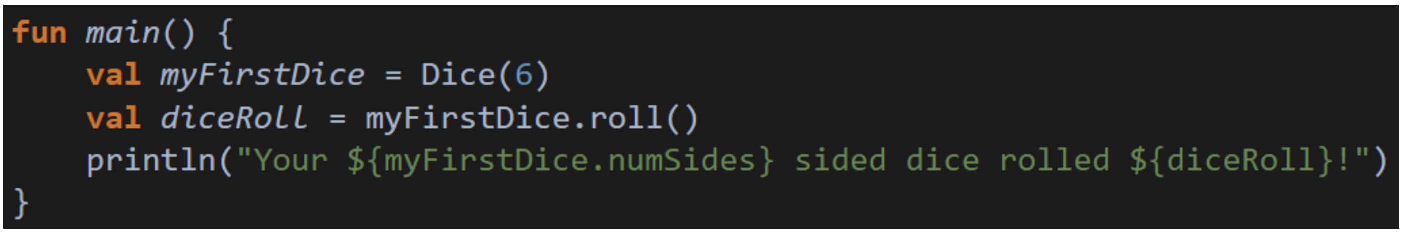
Customize your dice

1. Modify the Dice class definition to accept an integer called numSides. The code inside your class does not change.
2. Inside the Dice class, delete the sides variable, as you can now use numSides.
3. Also, fix the range to use numSides.



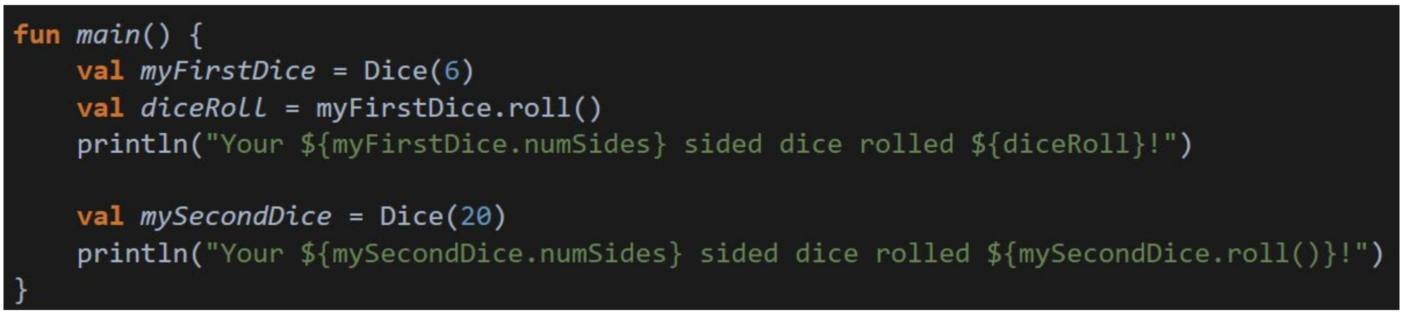
```
class Dice(val numSides: Int) {

    fun roll() : Int {
        val randomNumber = (1..numSides).random()
        return randomNumber
    }
}
```

4. In main(), to create myFirstDice with 6 sides, you must now supply in the number of sides as an argument to the Dice class, as shown below.
5. In the print statement, change sides to numSides.
6. Below that, delete the code that changes sides to 20, because that variable does not exist anymore.
7. Delete the println statement underneath it as well.



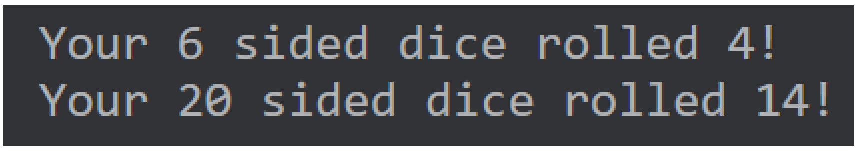
```
fun main() {
    val myFirstDice = Dice(6)
    val diceRoll = myFirstDice.roll()
    println("Your ${myFirstDice.numSides} sided dice rolled ${diceRoll}!")
}
```

8. After printing the first dice roll, add code to create and print a second Dice object called mySecondDice with 20 sides.
9. Add a print statement that rolls and prints the returned value.
10. Your finished main() function should look like this.



```
fun main() {
    val myFirstDice = Dice(6)
    val diceRoll = myFirstDice.roll()
    println("Your ${myFirstDice.numSides} sided dice rolled ${diceRoll}!")

    val mySecondDice = Dice(20)
    println("Your ${mySecondDice.numSides} sided dice rolled ${mySecondDice.roll()}!")
}
```

11. Run your finished program, and your output should look like this.



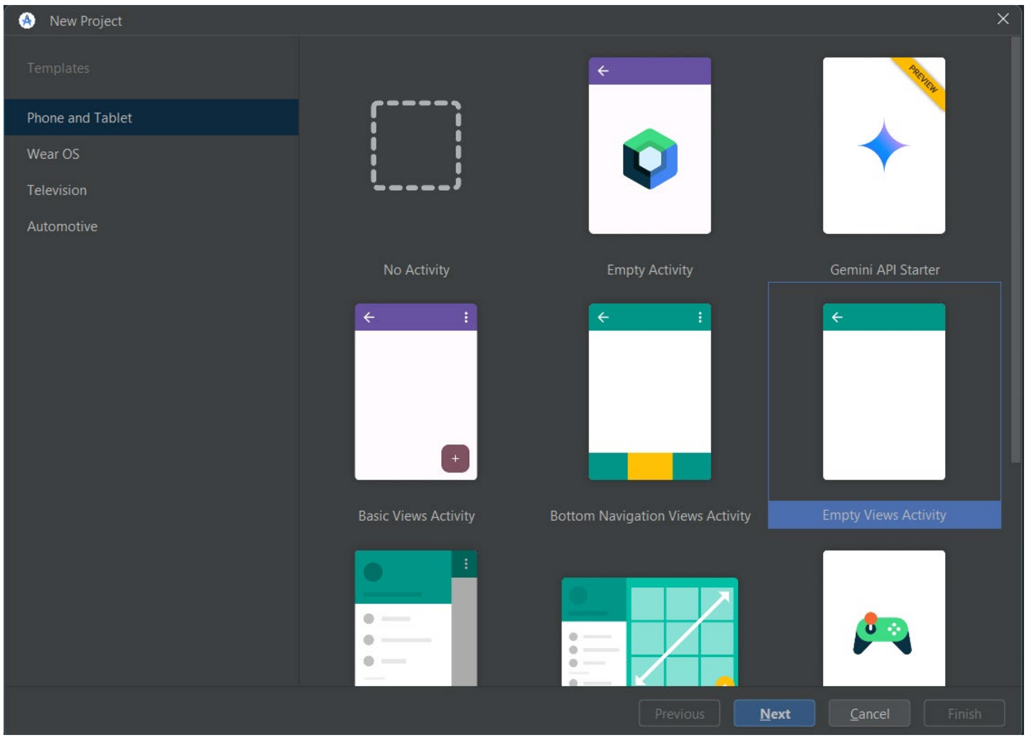
```
Your 6 sided dice rolled 4!
Your 20 sided dice rolled 14!
```

4.4. Create an interactive Dice Roller app

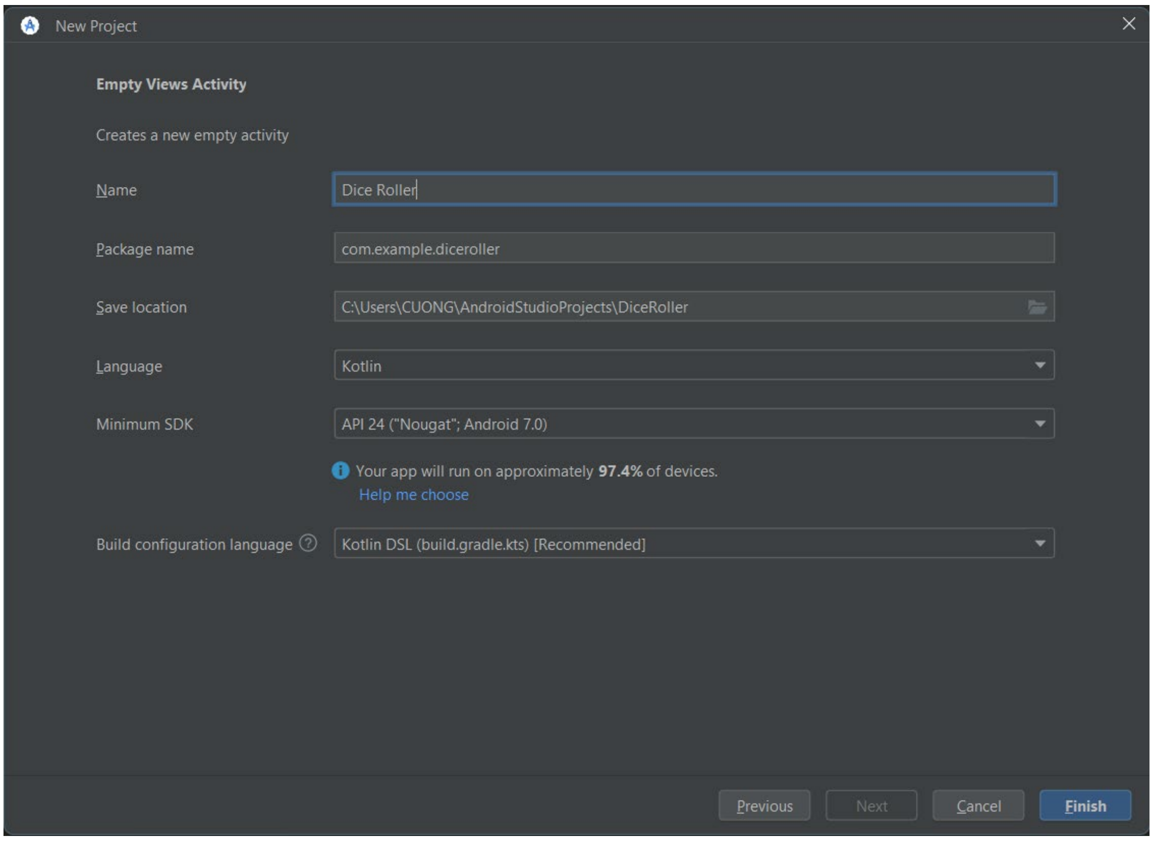
Set up your app

Create an Empty Views Activity project

1. If you already have an existing project open in Android Studio, go to File > New > New Project... to open the Create New Project screen.
2. In Create New Project, create a new Kotlin project using the Empty Views Activity template.



3. Call the app "Dice Roller", with a minimum API level of 24 (Nougat).

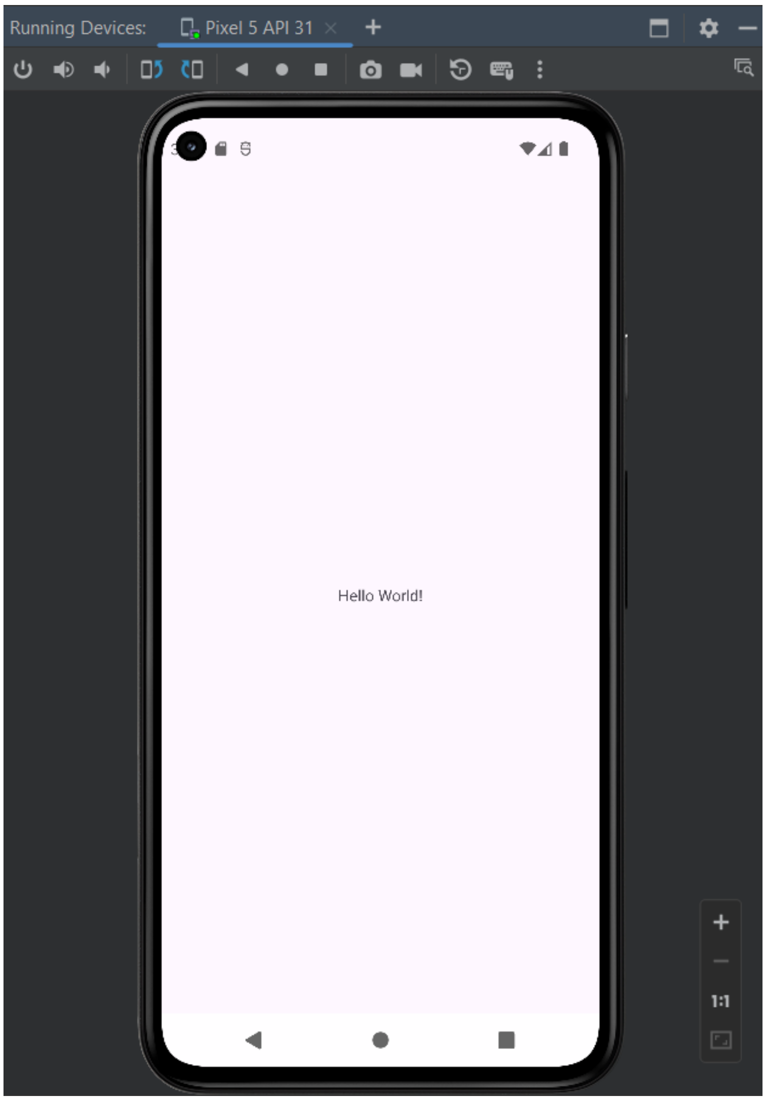

New Project

Empty Views Activity

Creates a new empty activity

Name	Dice Roller
Package name	com.example.diceroller
Save location	C:\Users\CUONG\AndroidStudioProjects\DiceRoller
Language	Kotlin
Minimum SDK	API 24 ("Nougat"; Android 7.0)
	Your app will run on approximately **97.4%** of devices. Help me choose
Build configuration language	Kotlin DSL (build.gradle.kts) [Recommended]

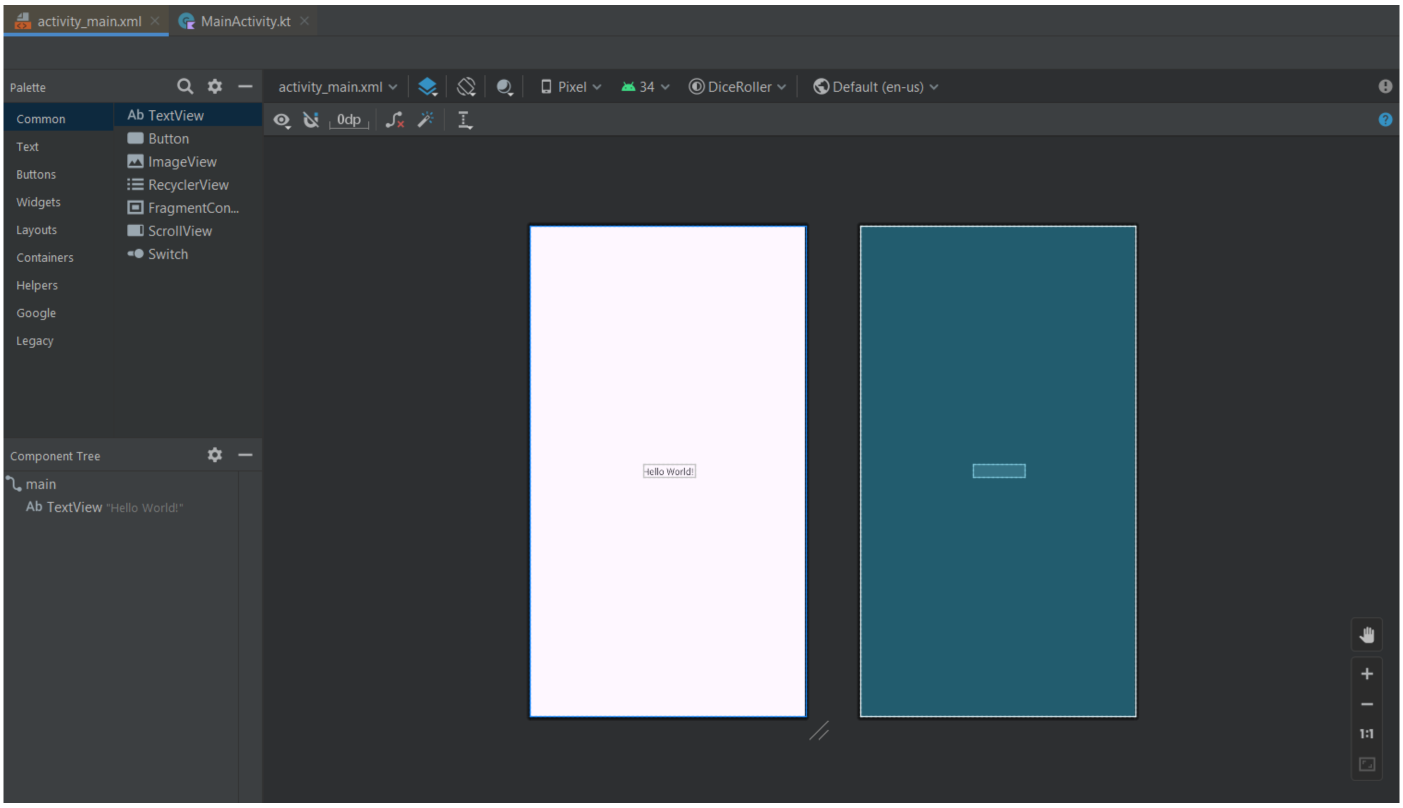
4. Run the new app and it should look like this.



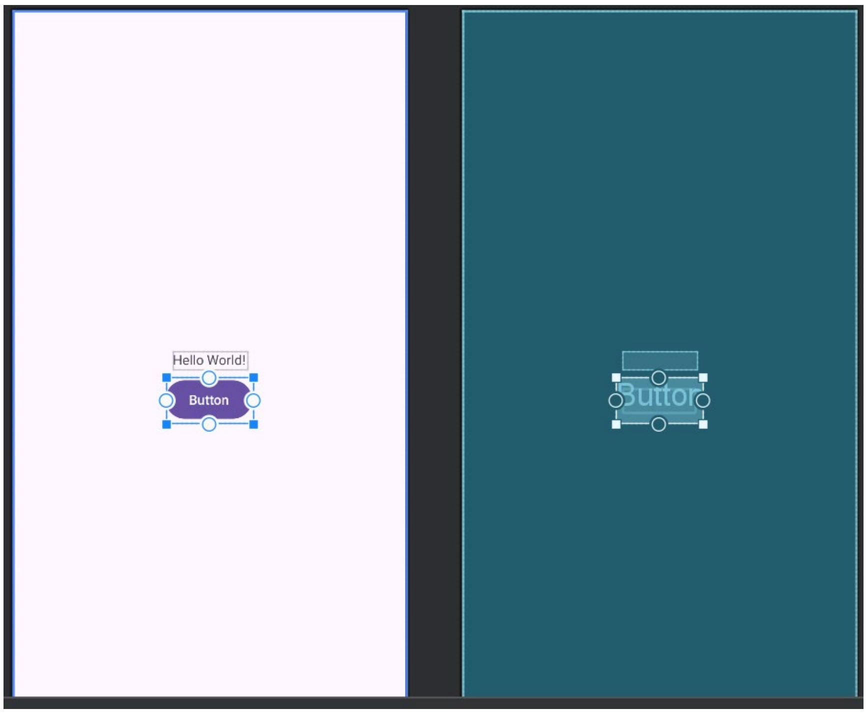
Create the layout for the app

Open the Layout Editor

1. In the Project window, double-click activity_main.xml (app > res > layout > activity_main.xml) to open it. You should see the Layout Editor, with only the "Hello World" TextView in the center of the app.

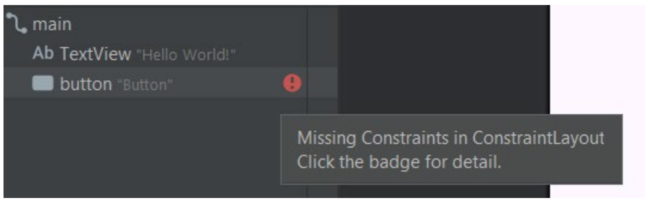


2. Next you will add a Button to your app. A Button is a user interface (UI) element in Android that the user can tap to perform an action. In this task, you add a Button below the "Hello World" TextView. The TextView and the Button will be located within a ConstraintLayout, which is a type of ViewGroup.



Add a Button to the layout

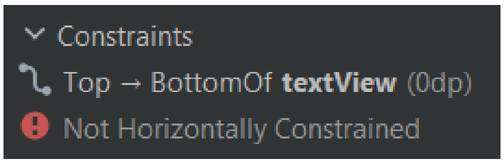
1. Drag a Button from the Palette onto the Design view, positioning it below the "Hello World" TextView.
2. Below the Palette in the Component Tree, verify that the Button and TextView are listed under the ConstraintLayout (as children of the ConstraintLayout).
3. Notice an error that the Button is not constrained. Since the Button is sitting within a ConstraintLayout, you must set vertical and horizontal constraints to position it.



Missing Constraints in ConstraintLayout
Click the badge for detail.

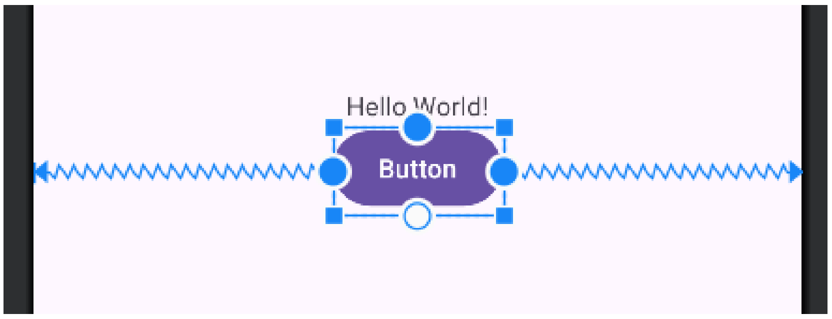
Position the Button

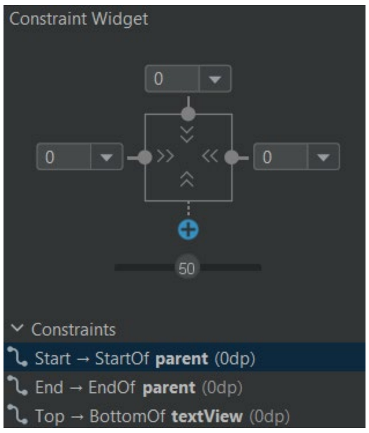
1. In the Design view, at the top edge of the Button, press and hold the white circle with a blue border. Drag the pointer, and an arrow will follow the pointer. Release when you reach the bottom edge of the "Hello World" TextView. This establishes a layout constraint, and the Button slides up to just beneath the TextView.
2. Look at the Attributes on the right hand side of the Layout Editor.
3. In the Constraint Widget, notice a new layout constraint that is set to the bottom of the TextView, Top → BottomOf textView (0dp). (0dp) means there is a margin of 0. You also have an error for missing horizontal constraints.



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

4. Add a horizontal constraint from the left side of the Button to the left side of the parent ConstraintLayout.
5. Repeat on the right side, connecting the right edge of the Button to the right edge of the ConstraintLayout. The result should look like this:

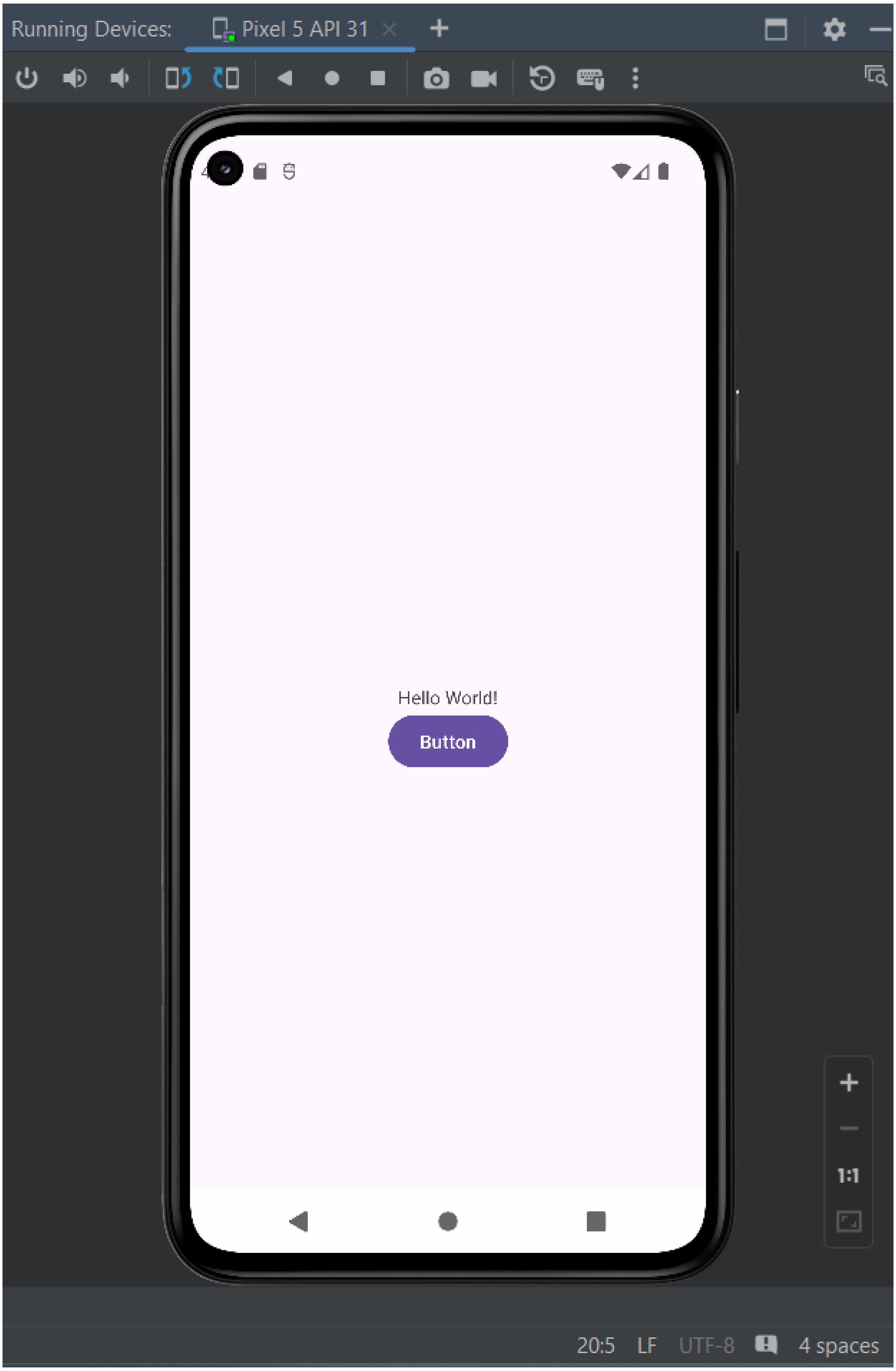




Constraints
Start → StartOf **parent** (0dp)
End → EndOf **parent** (0dp)
Top → BottomOf **textView** (0dp)

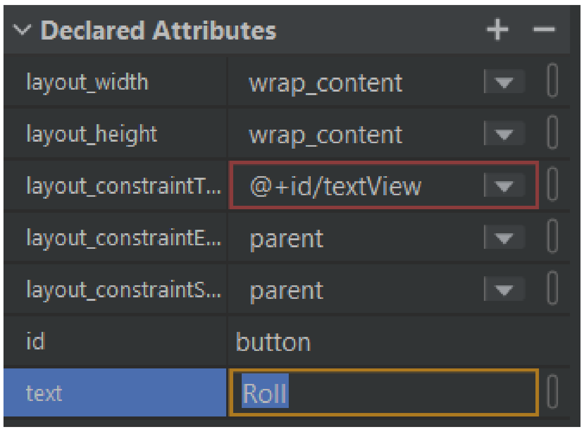
6. With the Button still selected, the Constraint Widget should look like this. Notice two additional constraints that have been added: Start → StartOf parent (0dp) and End → EndOf parent (0dp). This means the Button is horizontally centered in its parent, the ConstraintLayout.

7. Run the app. It should look like the screenshot below. You can click on the Button, but it doesn't do anything yet. Let's keep going!



Change the Button text

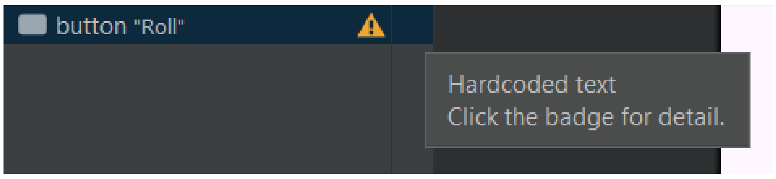
1. In the Layout Editor, with the Button selected, go to Attributes, change the text to Roll, and press the Enter (Return on the Mac) key.



Declared Attributes

layout_width	wrap_content
layout_height	wrap_content
layout_constraintT...	@+id/textView
layout_constraintE...	parent
layout_constraintS...	parent
id	button
text	Roll

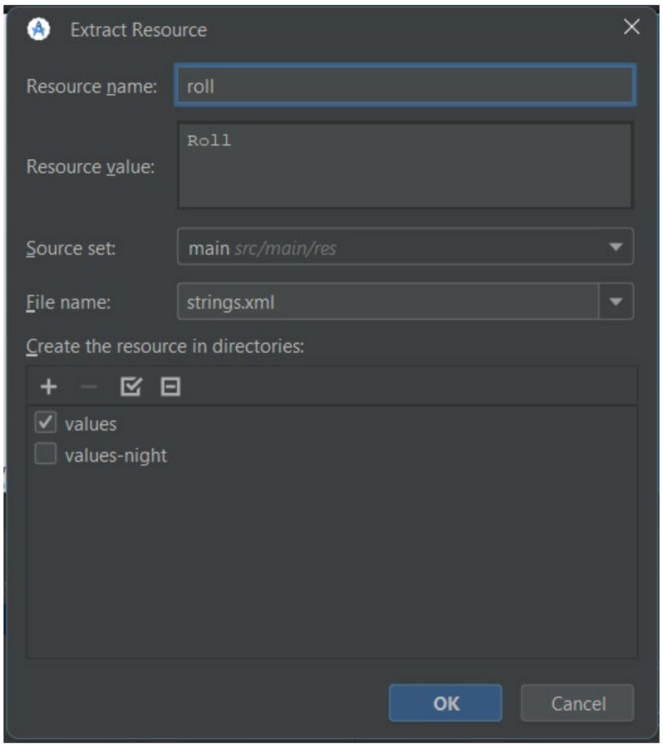
2. In the Component Tree, an orange warning triangle appears next to the Button. If you hover the pointer over the triangle, a message appears. Android Studio has detected a hardcoded string ("Roll") in your app code and suggests using a string resource instead.



button "Roll"

Hardcoded text
Click the badge for detail.

3. In the Component Tree, click on the orange triangle.
4. At the bottom of the message, under Suggested Fix, click the Fix button. (You may need to scroll down.)
5. The Extract Resource dialog opens. To extract a string means to take the "Roll" text and create a string resource called roll in strings.xml (app > res > values > strings.xml). The default values are correct, so click OK.



Extract Resource

Resource name: roll

Resource value: Roll

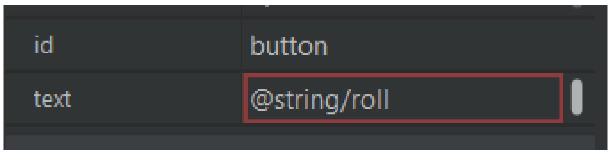
Source set: main src/main/res

File name: strings.xml

Create the resource in directories:

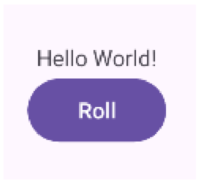
- [✓] values
- [ ] values-night

6. Notice that in Attributes, the text attribute for the Button now says @string/roll, referring to the resource you just created.



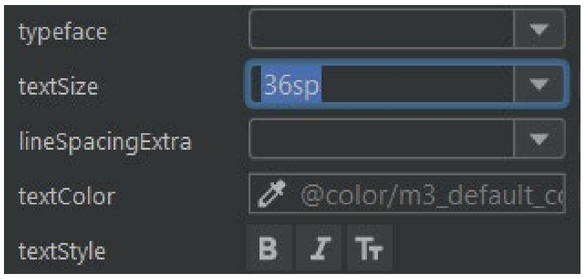
id	button
text	@string/roll

7. In the Design view, the Button should still say Roll on it.



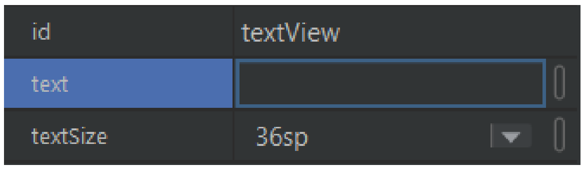
Style the TextView

1. In the Design Editor, select the TextView so that its attributes appear in the Attributes window.
   2.
2. Change the textSize of the TextView to 36sp, so that it's large and easy to read. You may need to scroll to find textSize.



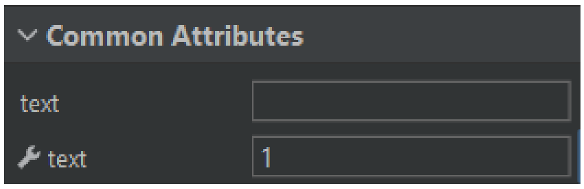
typeface	
textSize	36sp
lineSpacingExtra	
textColor	@color/m3_default_c
textStyle	B I Tт

3. Clear the text attribute of the TextView. You don't need to display anything in the TextView until the user rolls the dice.



id	textView
text	
textSize	36sp

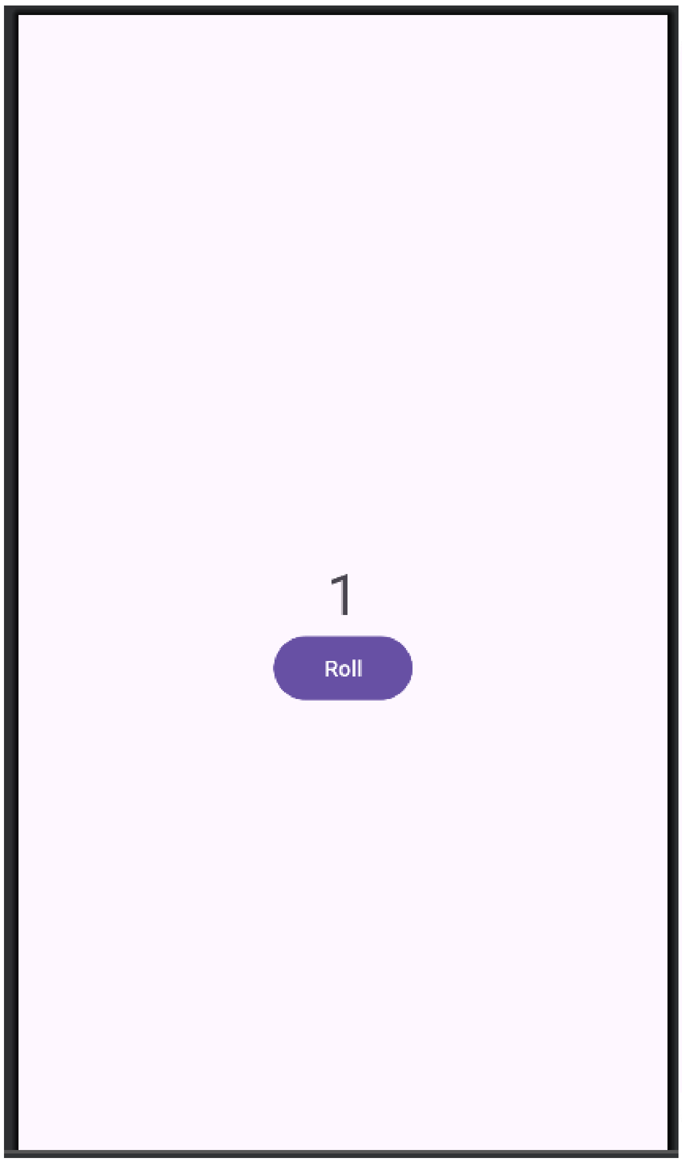
4. Select the TextView in the Component Tree.
5. Under Common Attributes, find the text attribute, and below it, another text attribute with a tool icon. The text attribute is what will be displayed to the user when the app is running. The text attribute with a tool icon is the "tools text" attribute that is just for you as a developer.
6. Set the tools text to be "1" in the TextView (to pretend you have a dice roll of 1). The "1" will only appear in the Design Editor within Android Studio, but it will not appear when you run the app on an actual device or emulator.

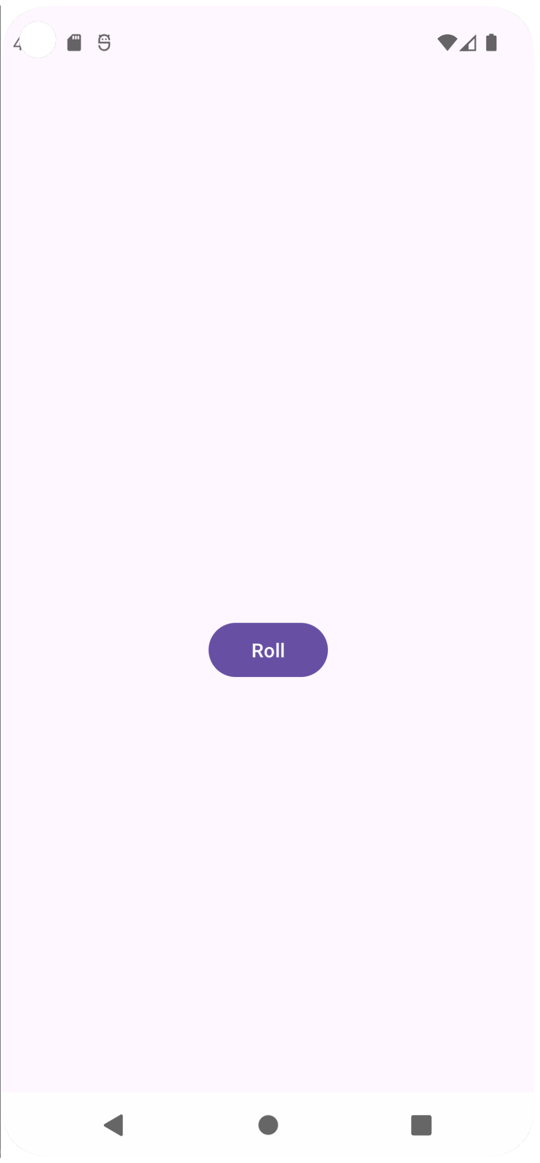


Common Attributes

text	
🔧 text	1

7. Look at your app in the preview. The "1" is showing



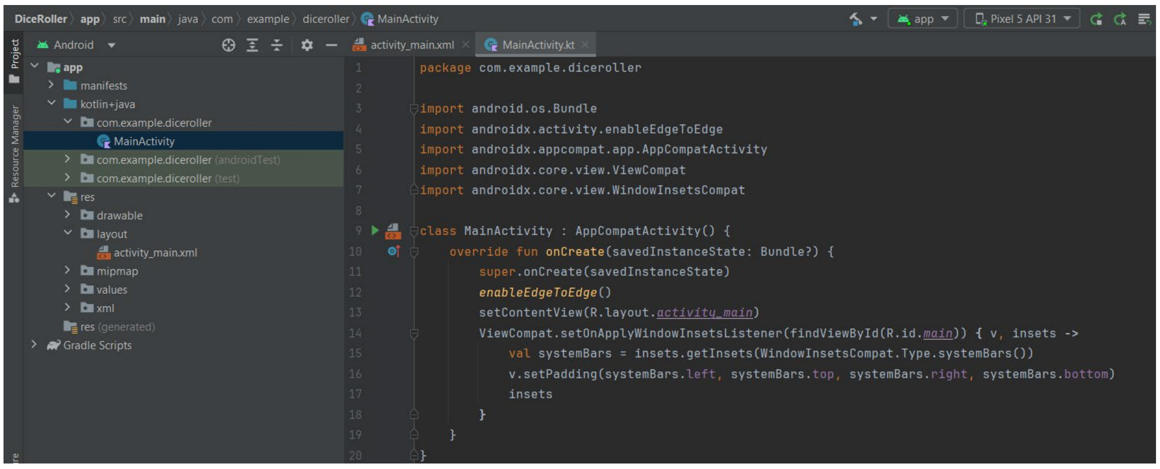


8. Run your app. This is what the app looks like when it's run on an emulator. The "1" is not showing. This is the correct behavior.

Introduction to Activities

Open the MainActivity.kt file

1. Navigate to and open the MainActivity.kt file (app > java > com.example.diceroller > MainActivity.kt). Below is what you should see. If you see import..., click on the ... to expand the imports.

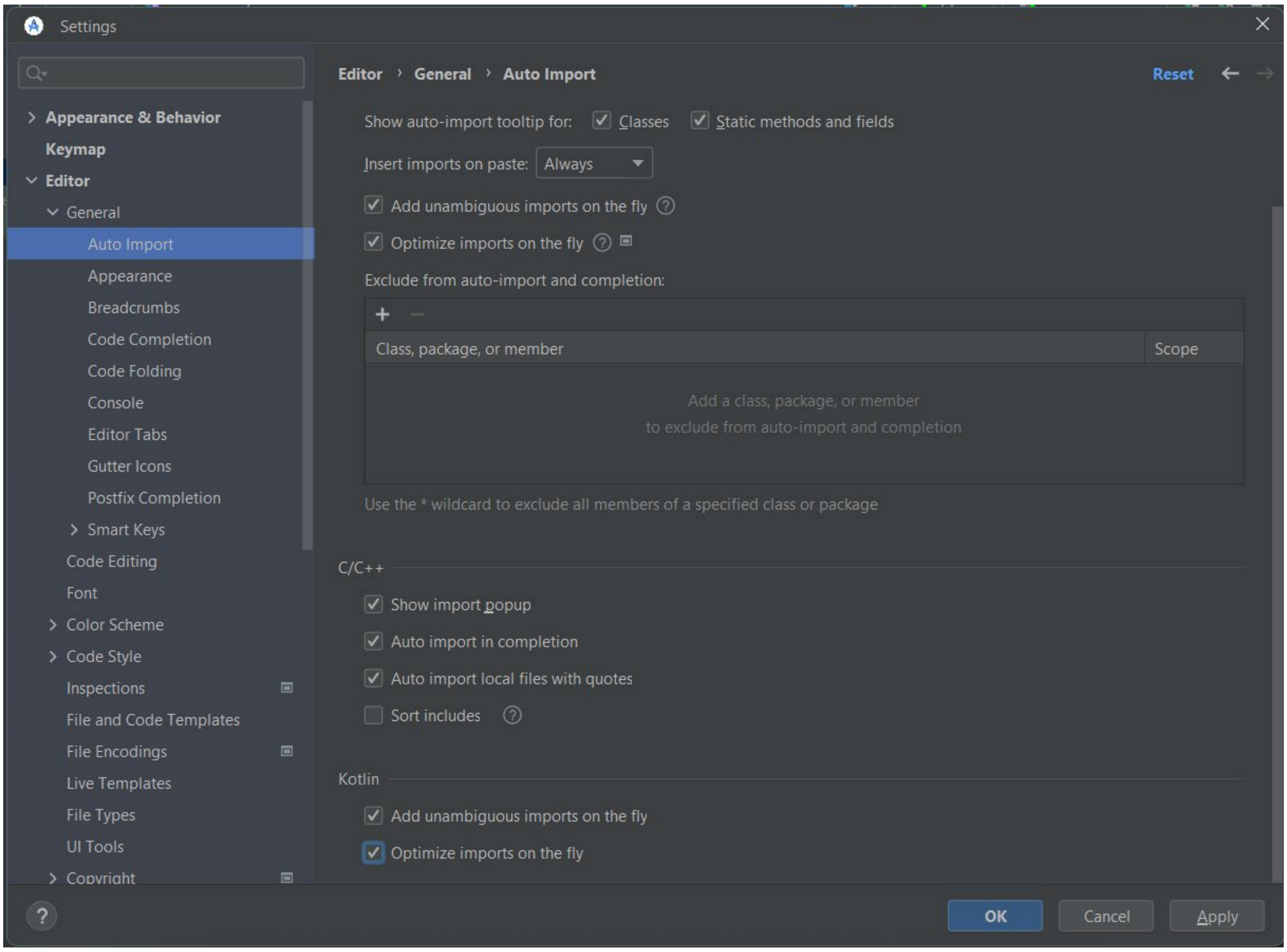


```kotlin
package com.example.diceroller

import android.os.Bundle
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)
            insets
        }
    }
}
```

2. Look at the Kotlin code for the MainActivity class, identified by the keyword class and then the name.
3. Notice that there is no main() function in your MainActivity.
4. Find the onCreate() method, which looks like the code below.
5. Notice the lines beginning with import.

Enable auto imports

In Windows, open the settings by going to File > Settings > Editor > General > Auto Import. In the Java and Kotlin sections, make sure Add unambiguous imports on the fly and Optimize imports on the fly (for current project) are checked. Note that there are two checkboxes in each section. Save the changes and close settings by pressing OK.


Settings

Editor › General › Auto Import

Show auto-import tooltip for: ✓ Classes ✓ Static methods and fields

Insert imports on paste: Always

✓ Add unambiguous imports on the fly

✓ Optimize imports on the fly

Exclude from auto-import and completion:

Class, package, or member	Scope

Add a class, package, or member to exclude from auto-import and completion

Use the * wildcard to exclude all members of a specified class or package

C/C++

✓ Show import popup

✓ Auto import in completion

✓ Auto import local files with quotes

☐ Sort includes

Kotlin

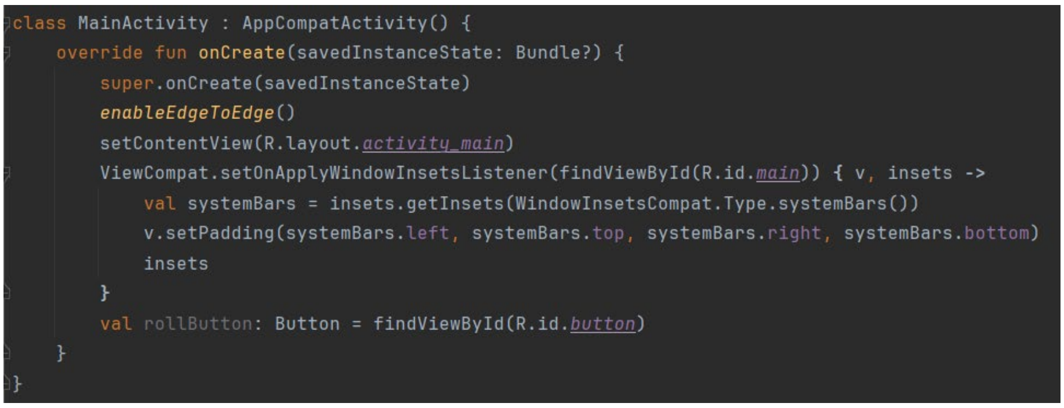
✓ Add unambiguous imports on the fly

✓ Optimize imports on the fly

Make the Button interface

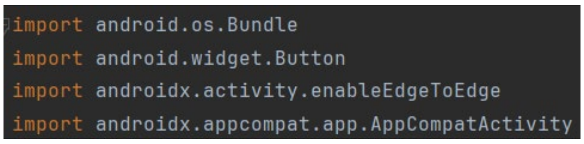
Display a message when the Button is clicked

1. Add the following code to the onCreate() method after the setContentView() call. The findViewById() method finds the Button in the layout. R.id.button is the resource ID for the Button, which is a unique identifier for it.



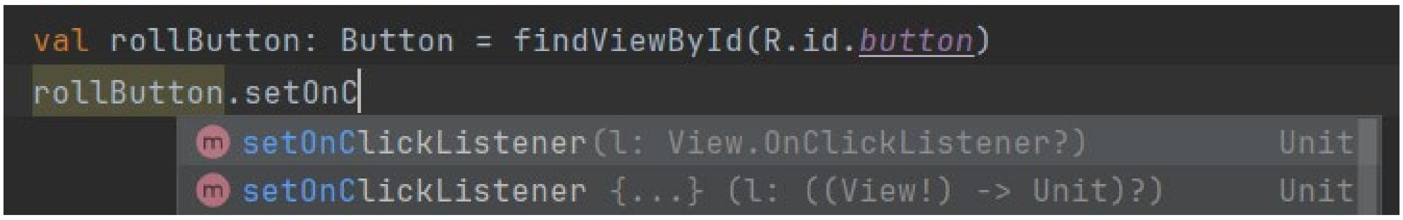
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)
            insets
        }
        val rollButton: Button = findViewById(R.id.button)
    }
}
```

2. Verify that Android Studio automatically added an import statement for the Button. Notice there are 3 import statements now.



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

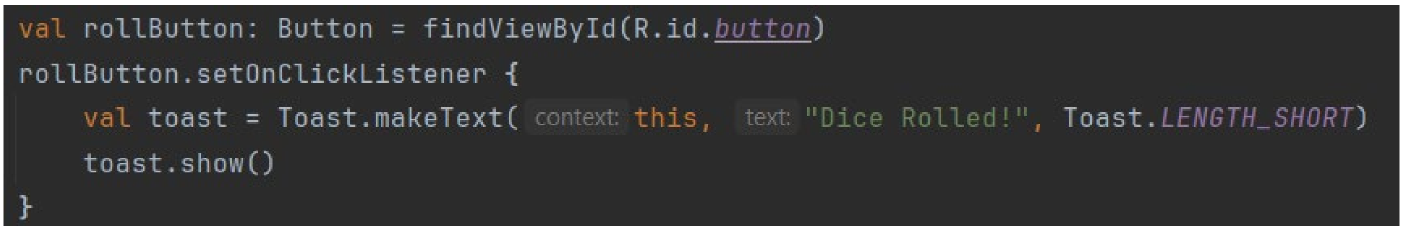
3. Use the rollButton object and set a click listener on it by calling the setOnClickListener() method. Instead of the parentheses following the method name, you will actually be using curly braces following the method name. This is a special syntax for declaring a Lambda. As you type, Android Studio may show multiple suggestions. For this case, choose the setOnClickListener {...} option.



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

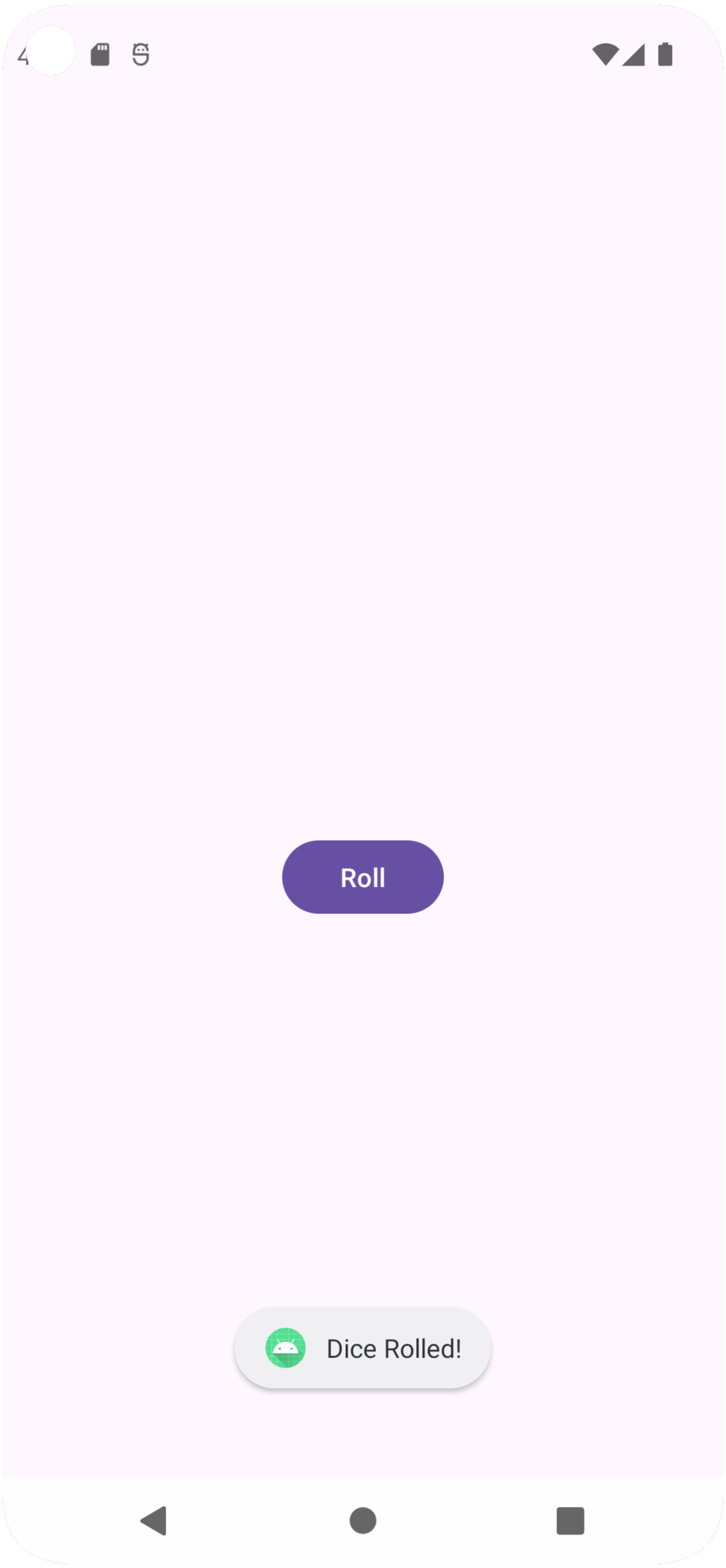
m setOnClickListener(l: View.OnClickListener?) Unit
m setOnClickListener {...} (l: ((View!) -> Unit)?) Unit

4. Create a Toast with the text "Dice Rolled!" by calling Toast.makeText().
5. Then tell the Toast to display itself by calling the show() method.



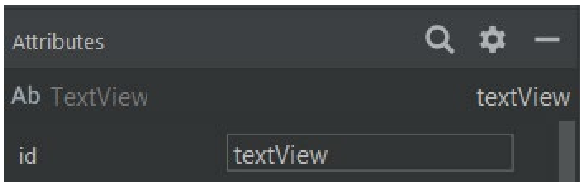
```
val rollButton: Button = findViewById(R.id.button)
rollButton.setOnClickListener {
    val toast = Toast.makeText( context: this,  text: "Dice Rolled!", Toast.LENGTH_SHORT)
    toast.show()
}
```

6. Run the app and click the Roll button. A toast message should pop up at the bottom of the screen and disappear after a short time.


Roll
Dice Rolled!

Update the TextView when the Button is clicked

1. Go back to activity_main.xml (app > res > layout >activity_main.xml)
2. Click on the TextView.
3. Note that the id is textView.

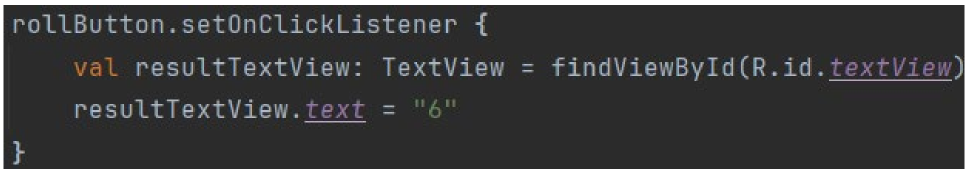


Attributes

Ab TextView textView

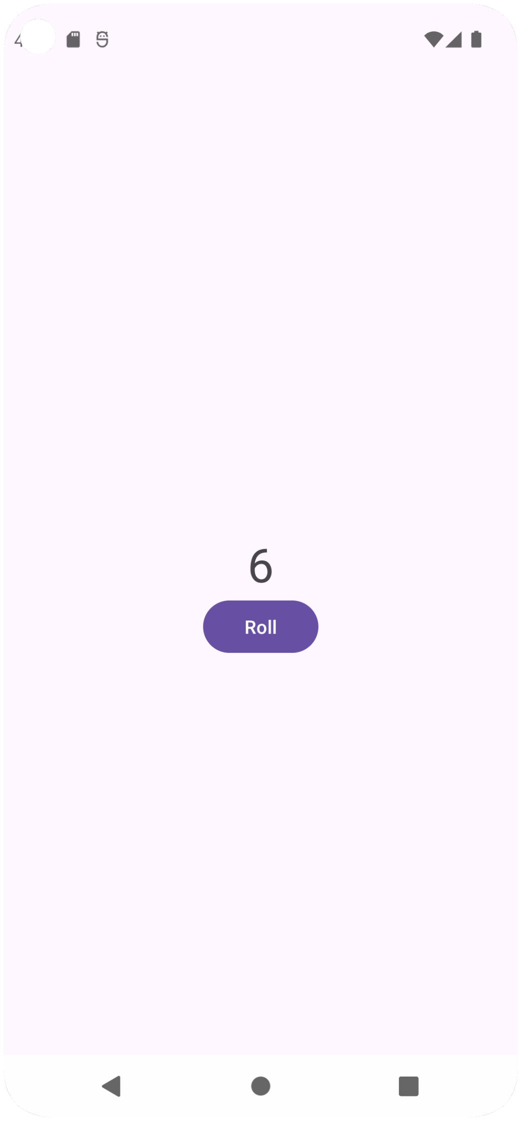
id textView

4. Open MainActivity.kt (app > java > com.example.diceroller > MainActivity.kt)
5. Delete the lines of code that create and show the Toast
6. In their place, create a new variable called resultTextView to store the TextView.
7. Use findViewById() to find textView in the layout using its ID, and store a reference to it.
8. Set the text on resultTextView to be "6" in quotations.



```
rollButton.setOnClickListener {
    val resultTextView: TextView = findViewById(R.id.textView)
    resultTextView.text = "6"
}
```

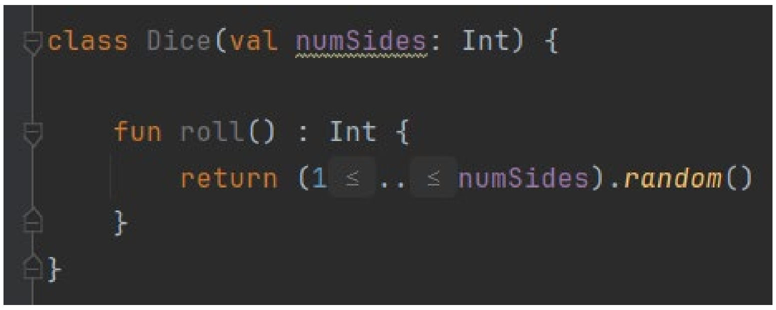
9. Run the app. Click the button. It should update the TextView to "6"



Add the dice roll logic

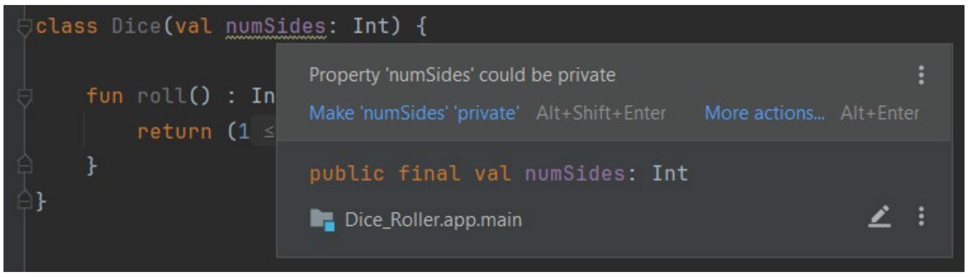
Add the Dice class

1. After the last curly brace in the MainActivity class, create the Dice class with a roll() method.



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

    fun roll() : Int {
        return (1 ≤ .. ≤ numSides).random()
    }
}
```

2. Notice that Android Studio may underline numSides with a wavy gray line. (This may take a moment to appear.)
3. Hover your pointer over numSides, and a popup appears saying Property ‘numSides' could be private.



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

    fun roll() : In
        return (1 ≤
    }
}
```

Property 'numSides' could be private

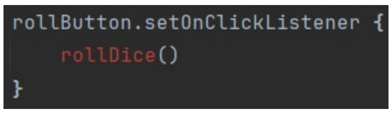
Make 'numSides' 'private' Alt+Shift+Enter
More actions... Alt+Enter

```
public final val numSides: Int
```

4. Go ahead and make the suggested fix from Android Studio by clicking Make ‘numSides' ‘private'.

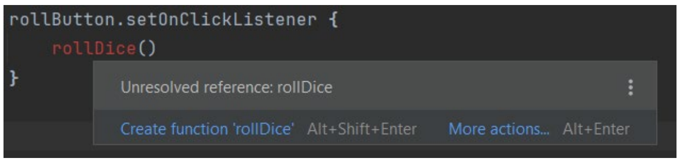
Create a rollDice() method

1. Replace the code in the click listener that sets the text to "6" with a call to rollDice().



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

2. Because rollDice() isn't defined yet, Android Studio flags an error and shows rollDice() in red.
3. If you hover your pointer over rollDice(), Android Studio displays the problem and some possible solutions.

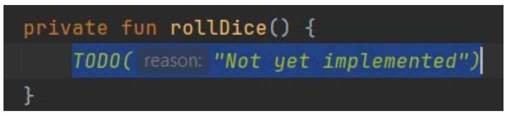


```
rollButton.setOnClickListener {
    rollDice()
}
```

Unresolved reference: rollDice

Create function 'rollDice' Alt+Shift+Enter
More actions... Alt+Enter

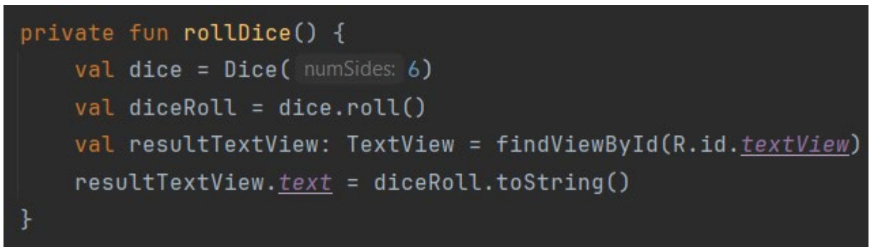
4. Click on More actions... which brings up a menu. Android Studio offers to do more work for you!
5. Select Create function ‘rollDice'. Android Studio creates an empty definition for the function inside MainActivity.



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

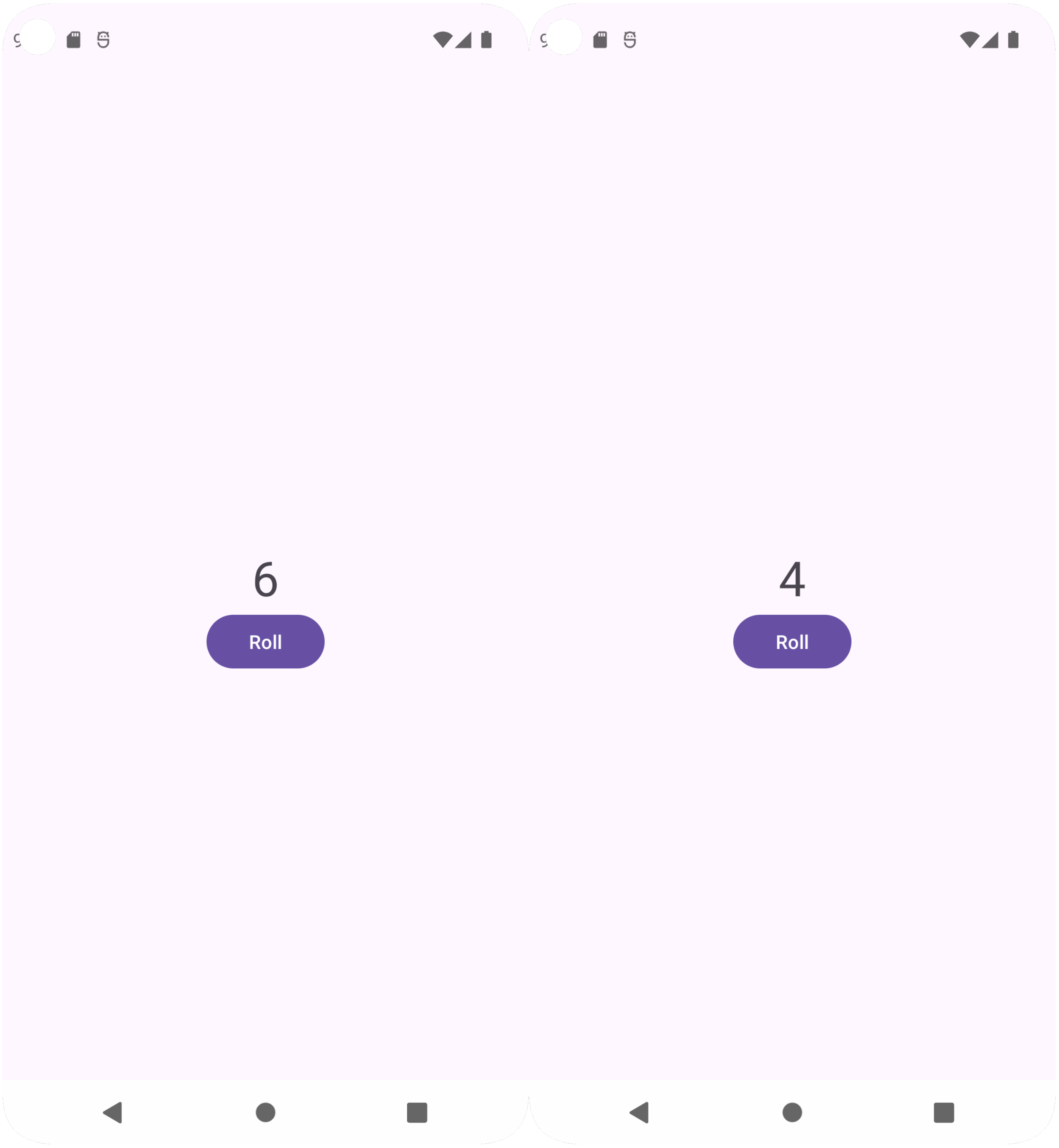
Create a new Dice object instance

1. Inside rollDice(), delete the TODO() call
2. Add code to create a dice with 6 sides.
3. Roll the dice by calling the roll() method, and save the result in a variable called diceRoll
4. Find the TextView by calling findViewById().
5. Convert diceRoll to a string and use that to update the text of the resultTextView.



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

6. Run your app



4.6. Add images to the Dice Roller app

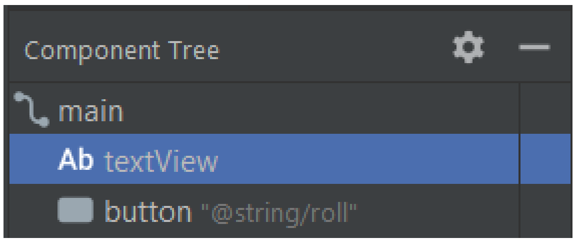
Update the layout for the app

Open Dice Roller app

Open activity_main.xml (app > res > layout > activity_main.xml). This opens the Layout Editor.

Delete the TextView

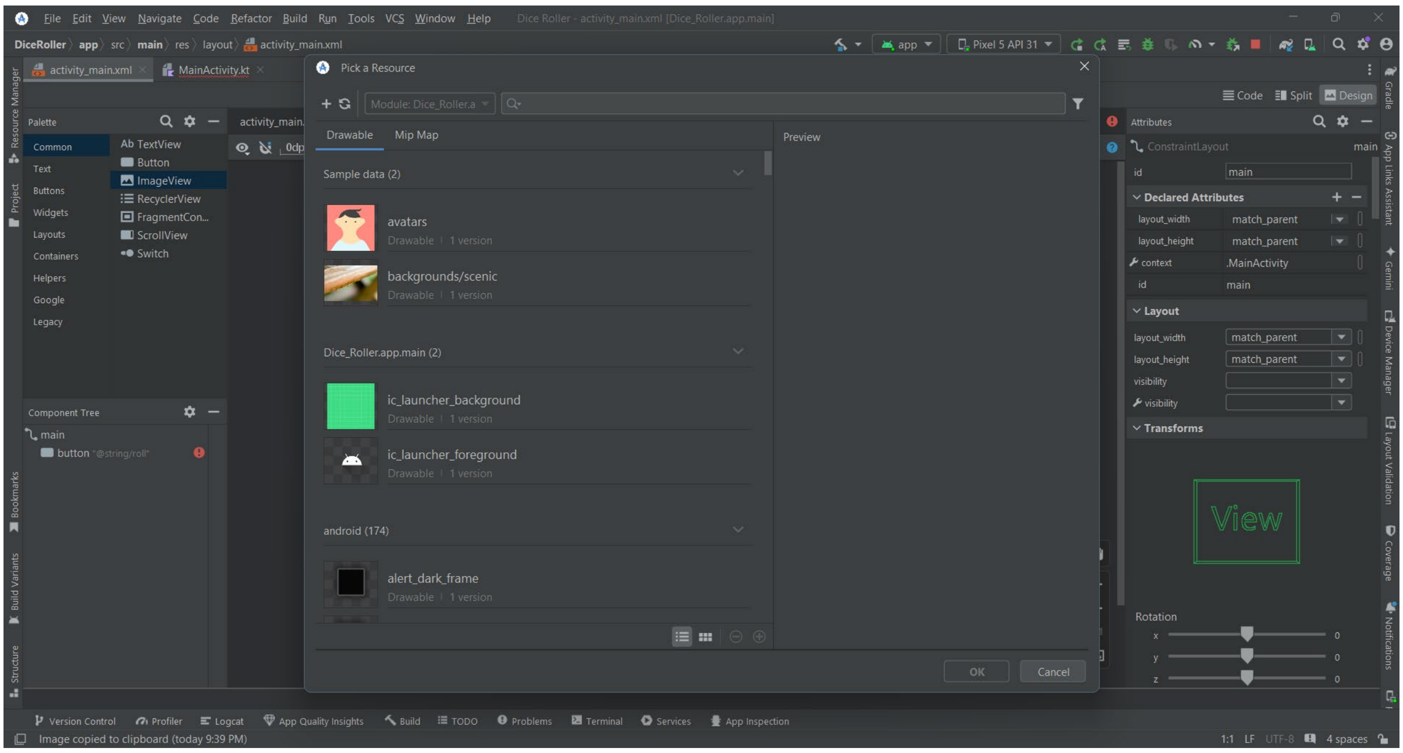
1. In the Layout Editor, select the TextView in the Component Tree.



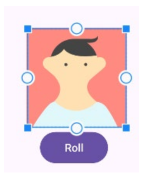
2. Right-click and choose Delete or press the Delete key.
3. Ignore the warning on the Button for now. You'll fix that in the next step.

Add an ImageView to the layout

1. Drag an ImageView from the Palette onto the Design view, positioning it above the Button.
2. In the Pick a Resource dialog, select avatars under Sample data. This is the temporary image you will use until you add the dice images in the next task.



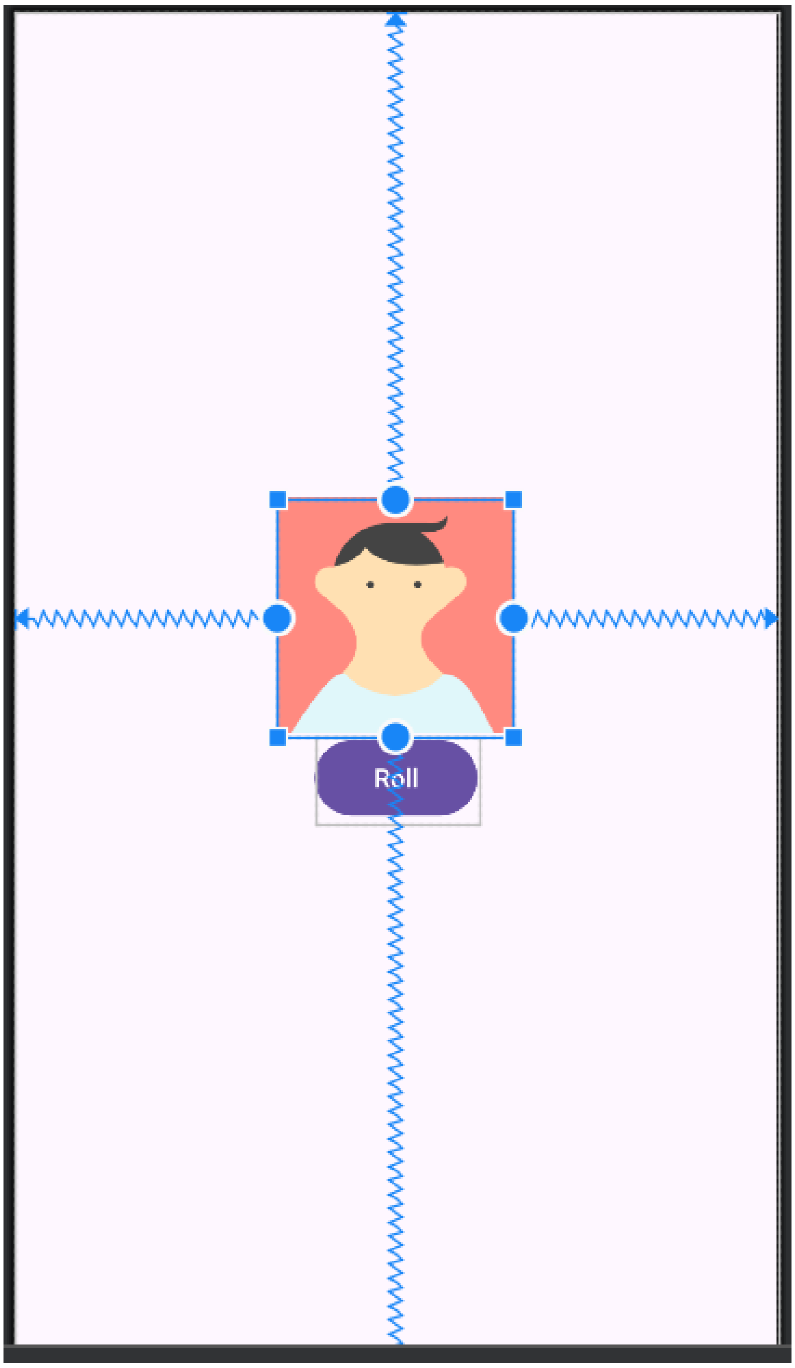
3. Press OK. The Design view of your app should look like this



4. In the Component Tree, you will notice two errors. The Button is not vertically constrained, and the ImageView is neither vertically nor horizontally constrained.

Position the ImageView and Button

1. Add horizontal constraints to the ImageView. Connect the left side of the ImageView to the left edge of the parent ConstraintLayout.
2. Connect the right side of the ImageView to the right edge of the parent. This will horizontally center the ImageView within the parent.
3. Add a vertical constraint to the ImageView, connecting the top of the ImageView to the top of the parent. The ImageView will slide up to the top of the ConstraintLayout.
4. Add a vertical constraint to the Button, connecting the top of the Button to the bottom of the ImageView. The Button will slide up beneath the ImageView.
5. Now select the ImageView again and add a vertical constraint connecting the bottom of the ImageView to the bottom of the parent. This centers the ImageView vertically in the ConstraintLayout.

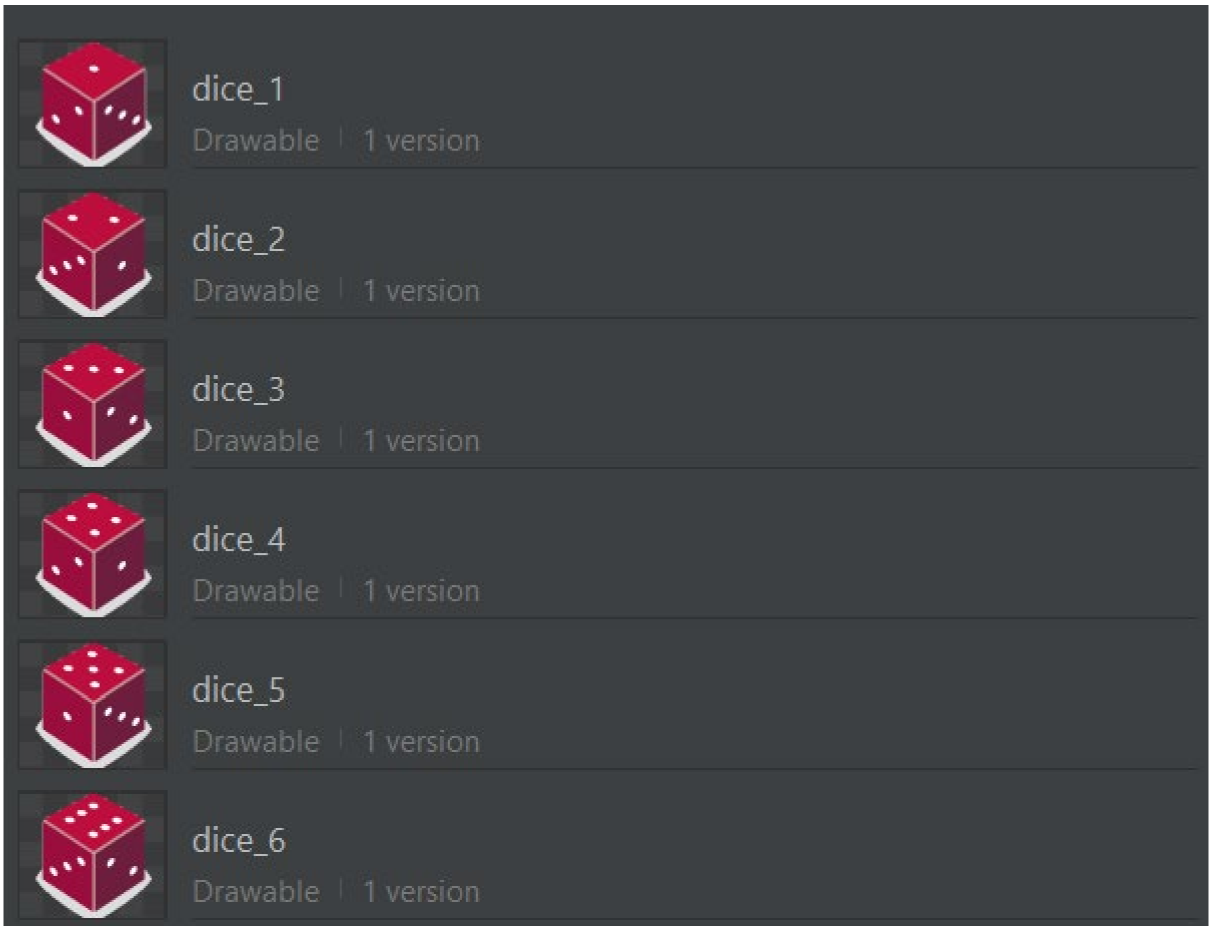


Add the dice images

Download the dice images

Add dice images to your app

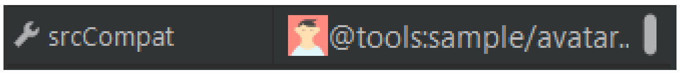
1. In Android Studio, click on View > Tool Windows > Resource Manager in the menus or click on the Resource Manager tab to the left of the Project window.
2. Click the + below Resource Manager, and select Import Drawables. This opens a file browser
3. Find and select the 6 dice image files. You can select the first file, then while holding down the Shift key, select the other files.
4. Click Open.
5. Click Next and then Import to confirm that you want to import these 6 resources


dice_1
Drawable | 1 version
dice_2
Drawable | 1 version
dice_3
Drawable | 1 version
dice_4
Drawable | 1 version
dice_5
Drawable | 1 version
dice_6
Drawable | 1 version

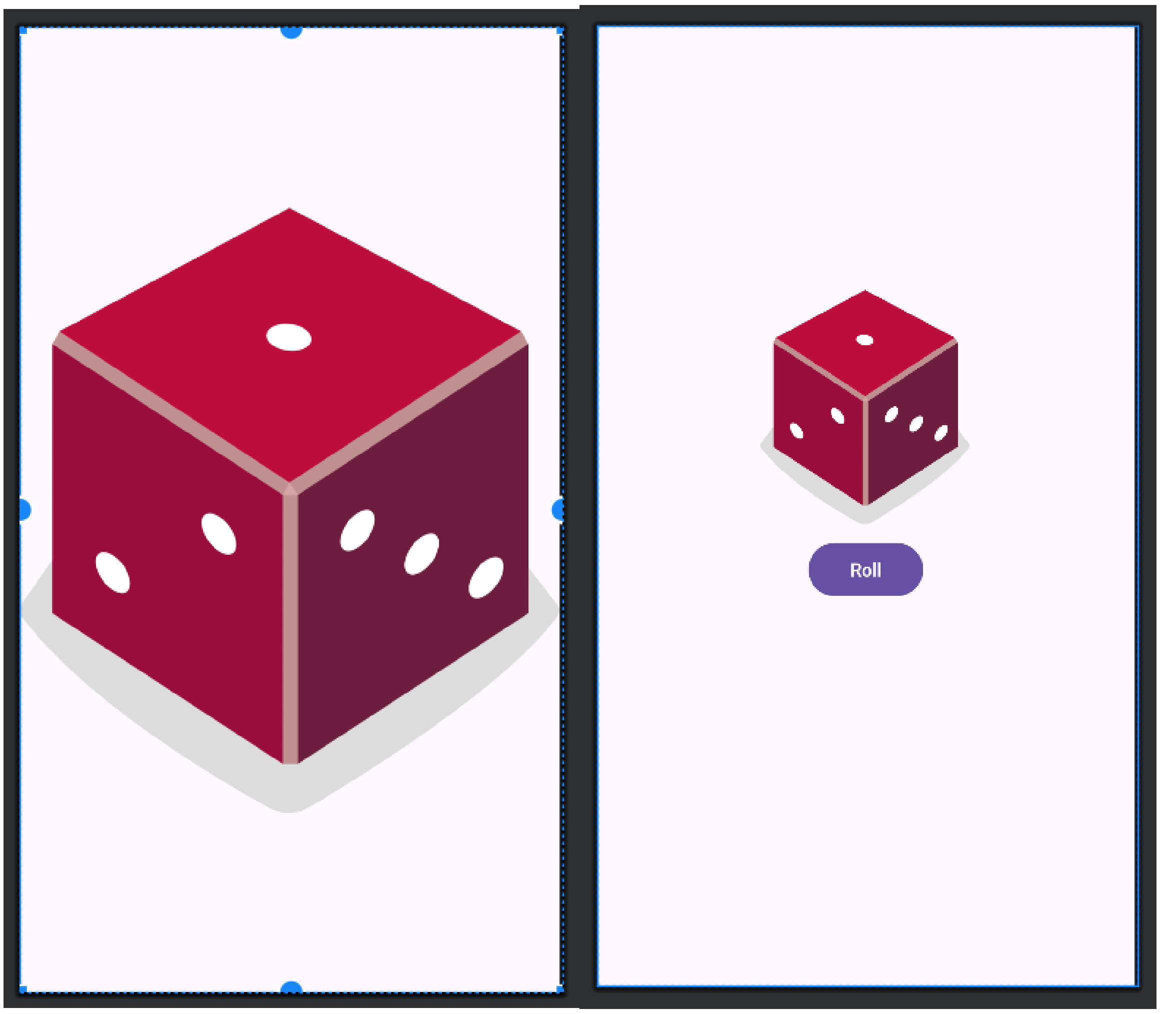
Use the dice images

Replace the sample avatar image

1. In the Design Editor, select the ImageView.
2. In Attributes in the Declared Attributes section, find the tool srcCompat attribute, which is set to the avatar image.
3. Click the tiny preview of the avatar. This opens a dialog to pick a new resource to use for this ImageView.


srcCompat @tools:sample/avatar..

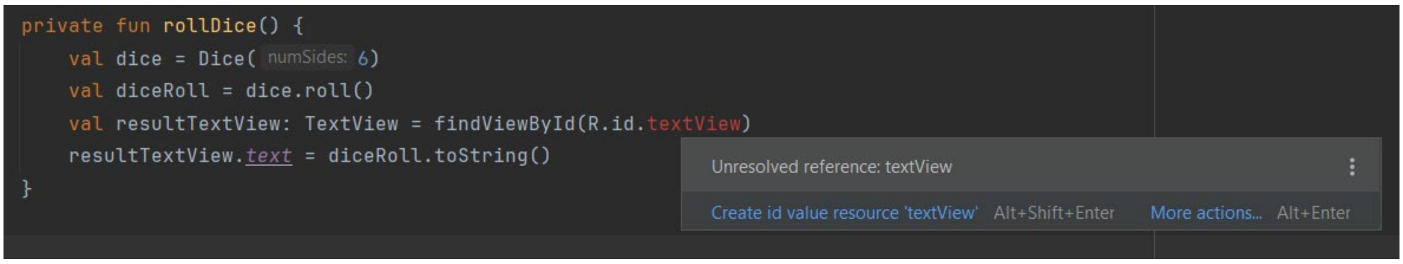
4. Select the dice_1 drawable and click OK.



5. In the Attributes window under the Constraints Widget, locate the layout_width and layout_height attributes. They are currently set to wrap_content, meaning that the ImageView will be as tall and as wide as the content (the source image) inside it.
6. Instead, set a fixed width of 160dp and fixed height of 200dp on the ImageView. Press Enter.
7. Add a top margin to the button of 16dp by setting it in the Constraint Widget.

Change the dice image when the button is clicked

1. Open MainActivity.kt (app > java > com.example.diceroller > MainActivity.kt)
2. Within the rollDice() method, select any code that refers to TextView and delete it.

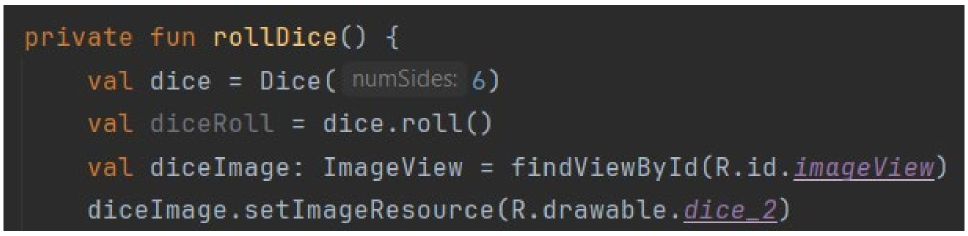


```
private fun rollDice() {
    val dice = Dice( numSides: 6)
    val diceRoll = dice.roll()
    val resultTextView: TextView = findViewById(R.id.textView)
    resultTextView.text = diceRoll.toString()
}
```

Unresolved reference: textView

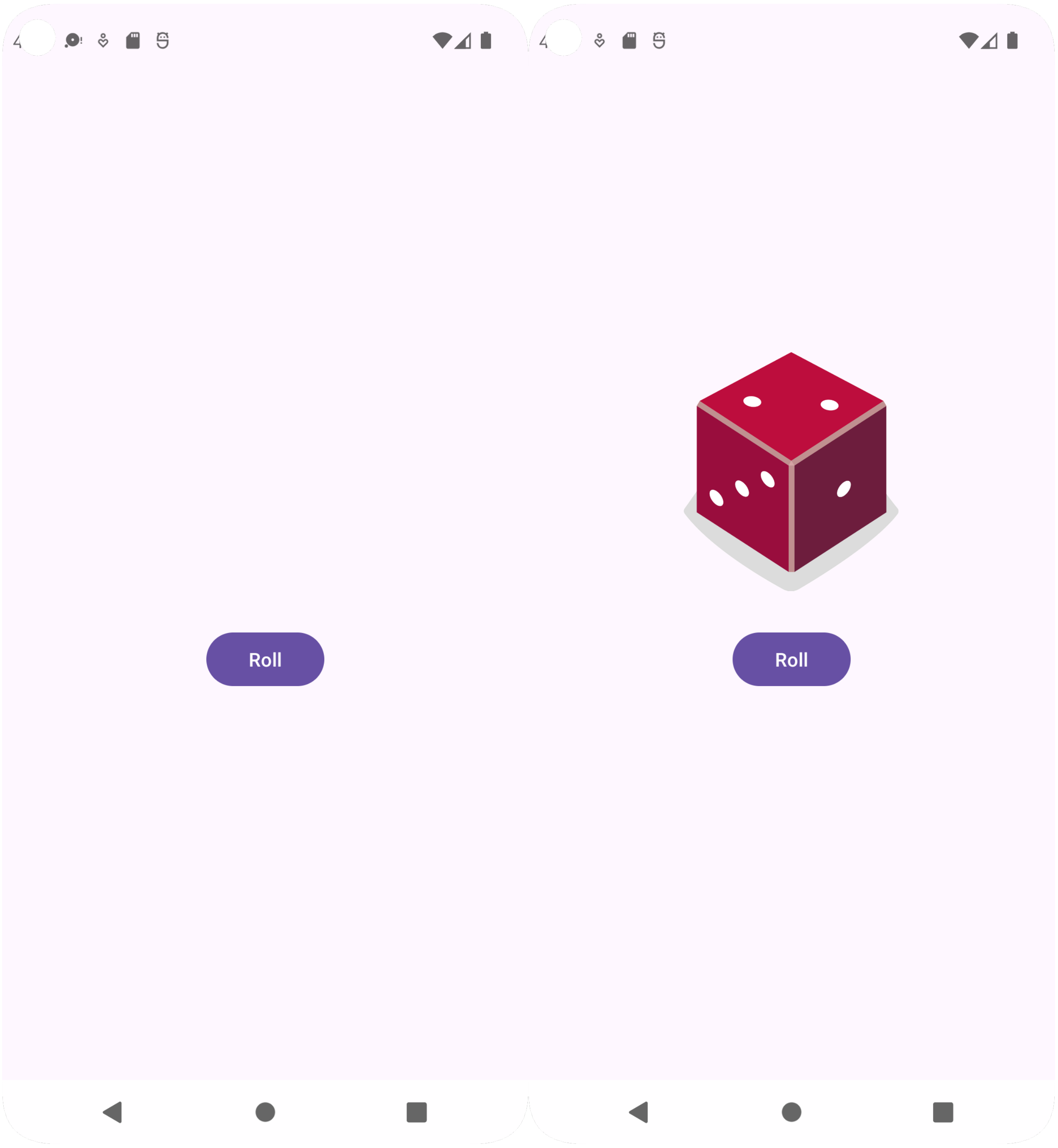
Create id value resource 'textView' Alt+Shift+Enter   More actions... Alt+Enter

3. Still within rollDice(), create a new variable called diceImage of type ImageView. Set it equal to the ImageView from the layout. Use the findViewById() method and pass in the resource ID for the ImageView, R.id.imageView, as the input argument.
4. Add this line of code to test that you can correctly update the ImageView when the button is clicked. The dice roll will not always be "2" but just use the dice_2 image for testing purposes



```
private fun rollDice() {
    val dice = Dice( numSides: 6)
    val diceRoll = dice.roll()
    val diceImage: ImageView = findViewById(R.id.imageView)
    diceImage.setImageResource(R.drawable.dice_2)
```

5. Run your app to verify that it runs without errors. The app should start off with a blank screen except for the Roll button. Once you tap the button, a dice image displaying the value 2 will appear.

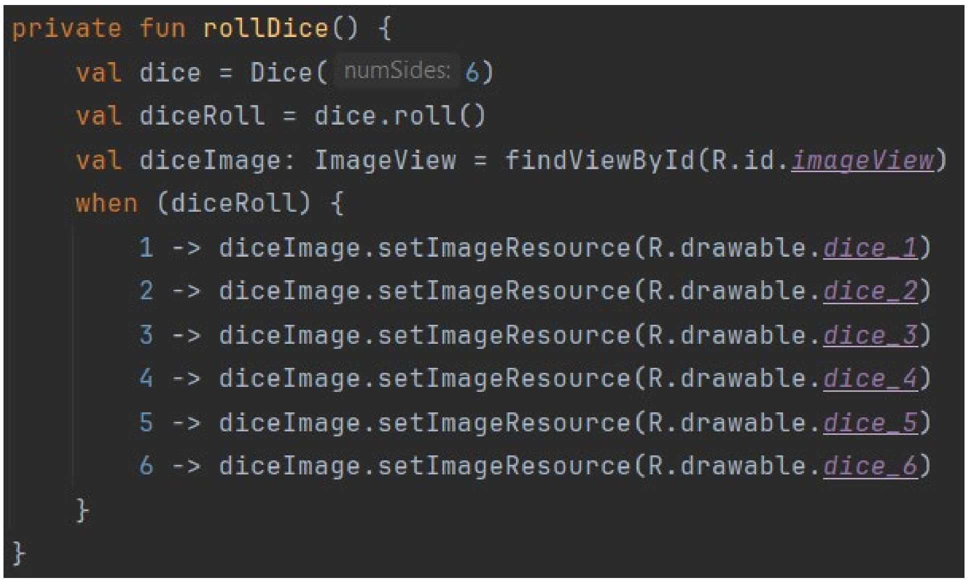


Display the correct dice image based on the dice roll

Update the rollDice() method

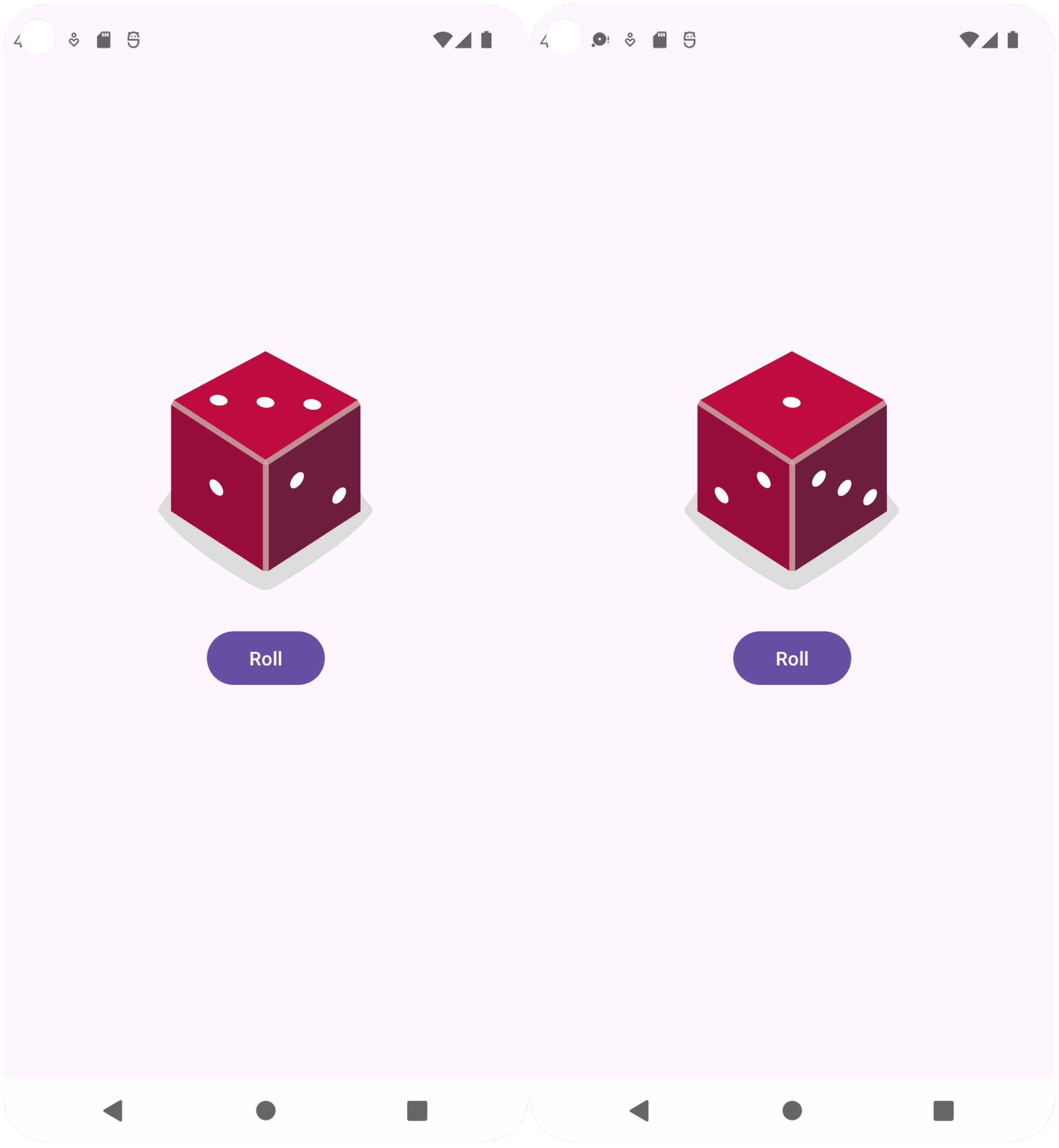
1. In the rollDice() method, delete the line of code that sets the image resource ID to dice_2 image every time.

2. Replace it with a "when" statement that updates the ImageView based on the diceRoll value.



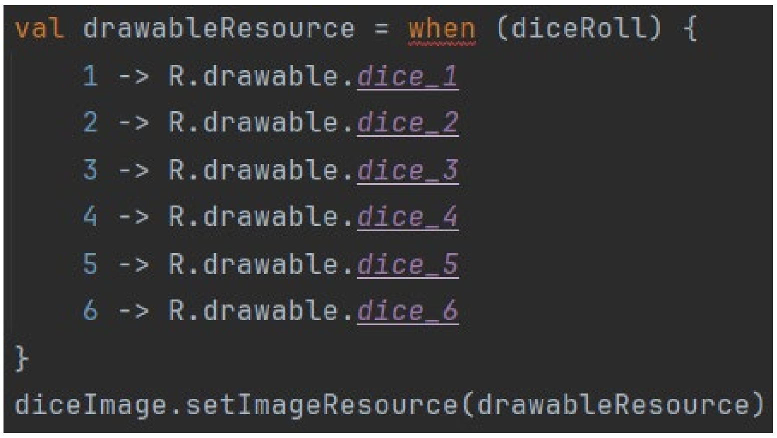
```
private fun rollDice() {
    val dice = Dice( numSides: 6)
    val diceRoll = dice.roll()
    val diceImage: ImageView = findViewById(R.id.imageView)
    when (diceRoll) {
        1 -> diceImage.setImageResource(R.drawable.dice_1)
        2 -> diceImage.setImageResource(R.drawable.dice_2)
        3 -> diceImage.setImageResource(R.drawable.dice_3)
        4 -> diceImage.setImageResource(R.drawable.dice_4)
        5 -> diceImage.setImageResource(R.drawable.dice_5)
        6 -> diceImage.setImageResource(R.drawable.dice_6)
    }
}
```

3. Run the app.



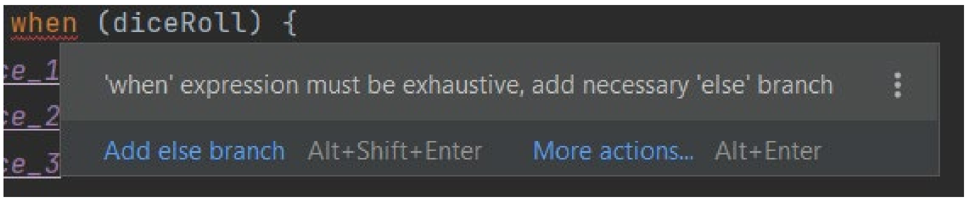
Optimize your code

1. Replace the code above with the following



```
val drawableResource = when (diceRoll) {
    1 -> R.drawable.dice_1
    2 -> R.drawable.dice_2
    3 -> R.drawable.dice_3
    4 -> R.drawable.dice_4
    5 -> R.drawable.dice_5
    6 -> R.drawable.dice_6
}
diceImage.setImageResource(drawableResource)
```

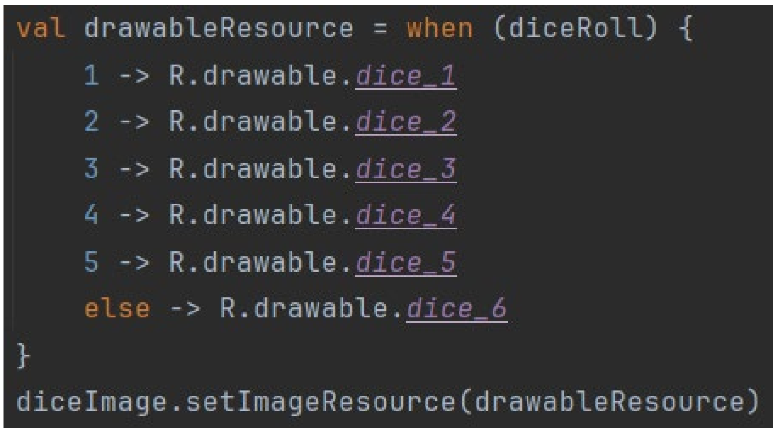
2. Notice that when is now underlined in red. If you hover your pointer over it, you'll see an error message: ‘when' expression must be exhaustive, add necessary ‘else' branch.



```
when (diceRoll) {
```

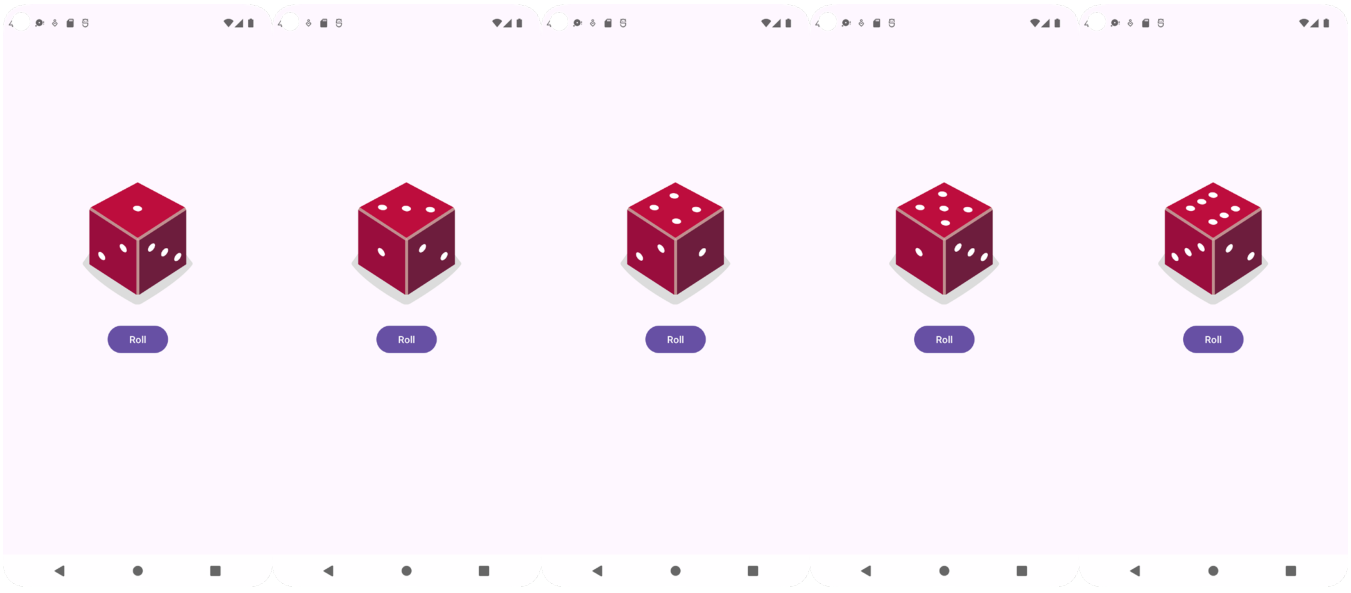
'when' expression must be exhaustive, add necessary 'else' branch

Add else branch Alt+Shift+Enter More actions... Alt+Enter



```
val drawableResource = when (diceRoll) {
    1 -> R.drawable.dice_1
    2 -> R.drawable.dice_2
    3 -> R.drawable.dice_3
    4 -> R.drawable.dice_4
    5 -> R.drawable.dice_5
    else -> R.drawable.dice_6
}
diceImage.setImageResource(drawableResource)
```

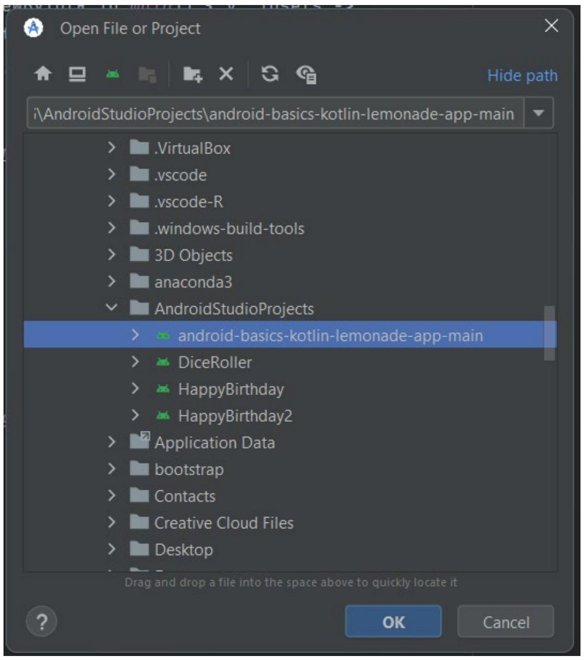
3. Run the app to make sure it still works correctly.



Project: Lemonade app

Open the project in Android Studio

1. Select the File > Open menu option.
2. In the file browser, navigate to where the unzipped project folder is located.
3. Double-click on that project folder.



Open File or Project

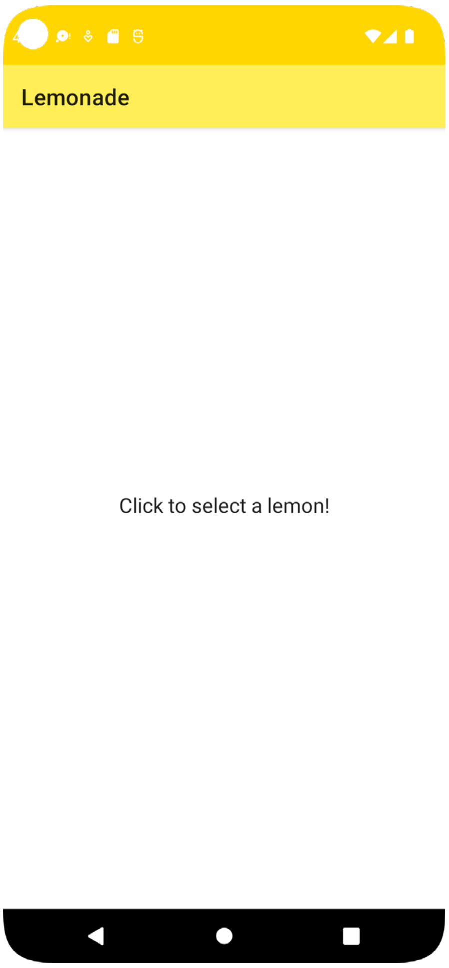
Hide path

\AndroidStudioProjects\android-basics-kotlin-lemonade-app-main

```
> .VirtualBox
> .vscode
> .vscode-R
> .windows-build-tools
> 3D Objects
> anaconda3
v AndroidStudioProjects
    > android-basics-kotlin-lemonade-app-main
    > DiceRoller
    > HappyBirthday
    > HappyBirthday2
> Application Data
> bootstrap
> Contacts
> Creative Cloud Files
> Desktop
```

Drag and drop a file into the space above to quickly locate it

4. Click the Run to build and run the app.



Build your user interface

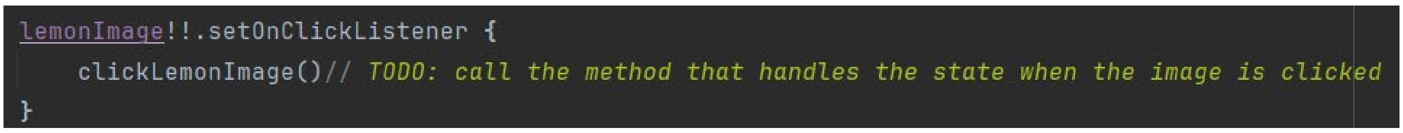
Open file activity_main.xml

Make your app interactive

Open file ActivityMain.kt

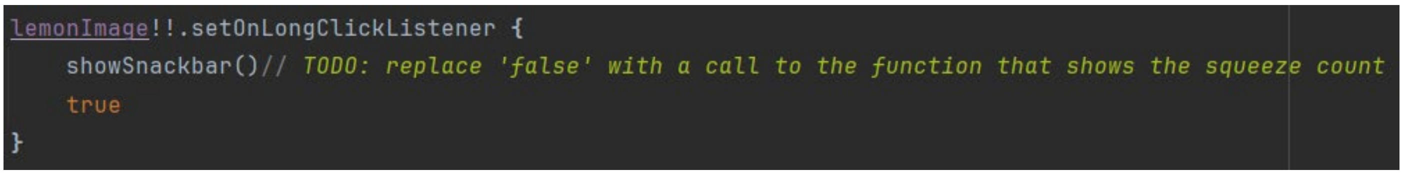
Step 1: Configure the ImageView

1. setOnClickListener() should update the app's state. The method to do this is clickLemonImage().



```kotlin
lemonImage!!.setOnClickListener {
    clickLemonImage()// TODO: call the method that handles the state when the image is clicked
}
```

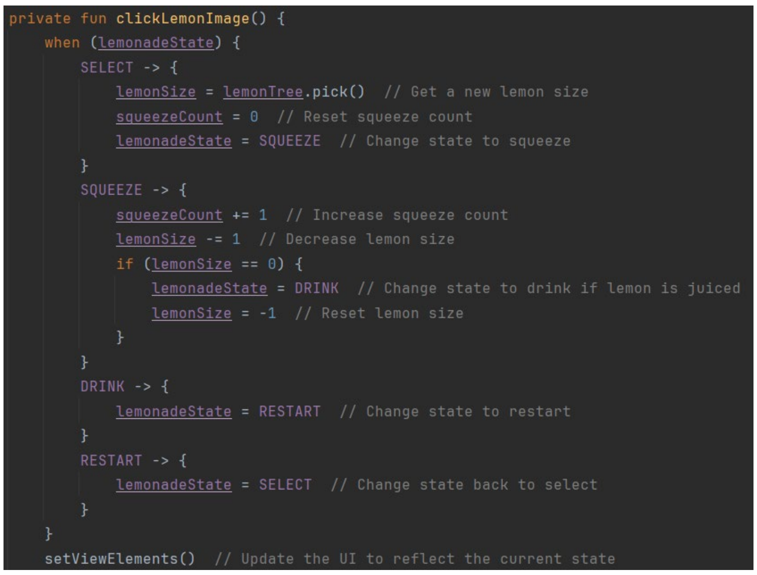
2. setOnLongClickListener() responds to events where the user long presses on an image (e.g. the user taps on the image and doesn't immediately release their finger). For long press events, a widget, called a snackbar, appears at the bottom of the screen letting the user know how many times they've squeezed the lemon. This is done with the showSnackbar() method.



```kotlin
lemonImage!!.setOnLongClickListener {
    showSnackbar()// TODO: replace 'false' with a call to the function that shows the squeeze count
    true
}
```

Step 2: Implement clickLemonImage()

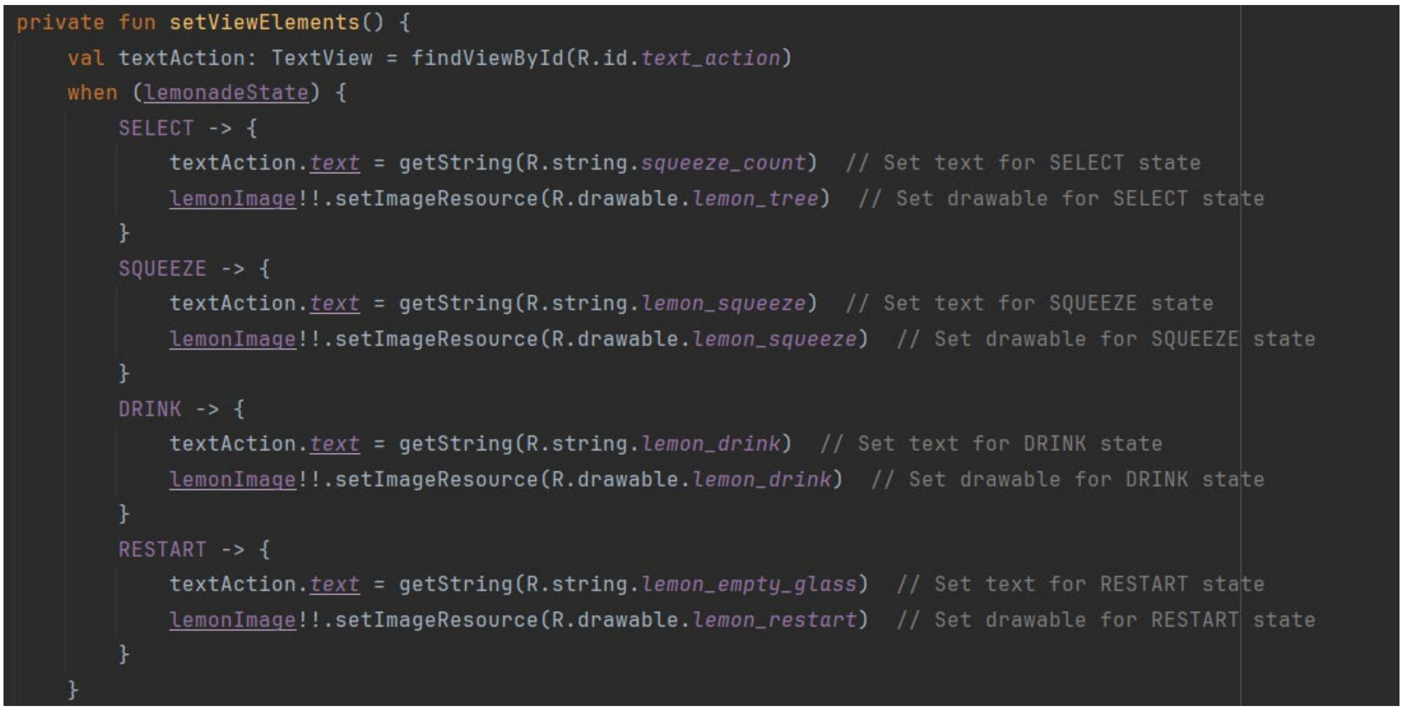
1. SELECT: Transition to the SQUEEZE state, set the lemonSize (the number of squeezes needed) by calling the pick() method, and setting the squeezeCount (the number of times the user has squeezed the lemon) to 0.
2. SQUEEZE: Increment the squeezeCount by 1 and decrement the lemonSize by 1. Remember that a lemon will require a variable number of squeezes before the app can transition its state. Transition to the DRINK state only if the new lemonSize is equal to 0. Otherwise, the app should remain in the SQUEEZE state.
3. DRINK: Transition to the RESTART state and set the lemonSize to -1.
4. RESTART: Transition back to the SELECT state.



```kotlin
private fun clickLemonImage() {
    when (lemonadeState) {
        SELECT -> {
            lemonSize = lemonTree.pick()  // Get a new lemon size
            squeezeCount = 0  // Reset squeeze count
            lemonadeState = SQUEEZE  // Change state to squeeze
        }
        SQUEEZE -> {
            squeezeCount += 1  // Increase squeeze count
            lemonSize -= 1  // Decrease lemon size
            if (lemonSize == 0) {
                lemonadeState = DRINK  // Change state to drink if lemon is juiced
                lemonSize = -1  // Reset lemon size
            }
        }
        DRINK -> {
            lemonadeState = RESTART  // Change state to restart
        }
        RESTART -> {
            lemonadeState = SELECT  // Change state back to select
        }
    }
    setViewElements()  // Update the UI to reflect the current state
```

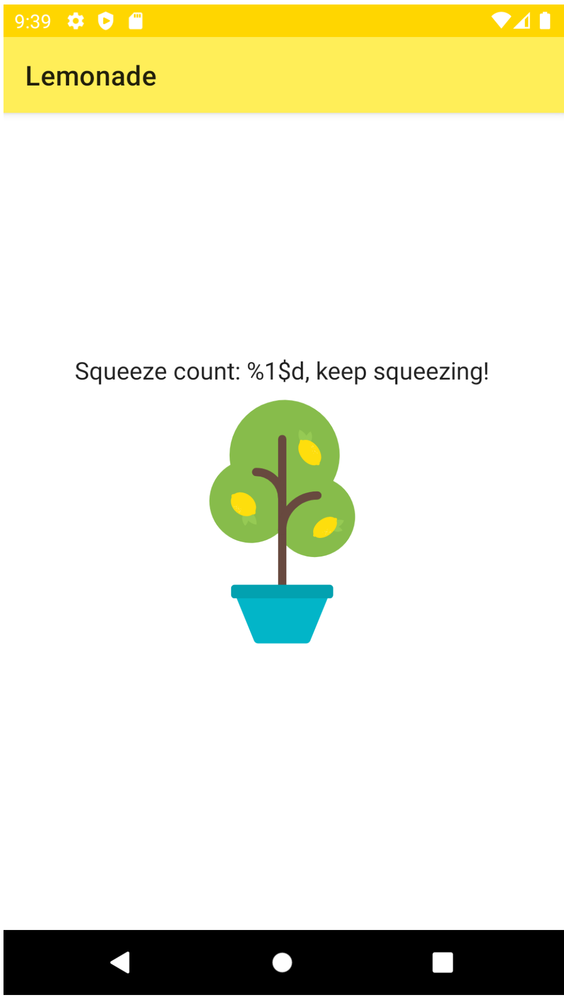
Step 3: Implement setViewElements()

1. SELECT:
   - Text: Click to select a lemon!
   - Image: R.drawable.lemon_tree
2. SQUEEZE:
   - Text: Click to juice the lemon!
   - Image: R.drawable.lemon_squeeze
3. DRINK:
   - Text: Click to drink your lemonade!
   - Image: R.drawable.lemon_drink
4. RESTART:
   - Text: Click to start again!
   - Image: R.drawable.lemon_restart

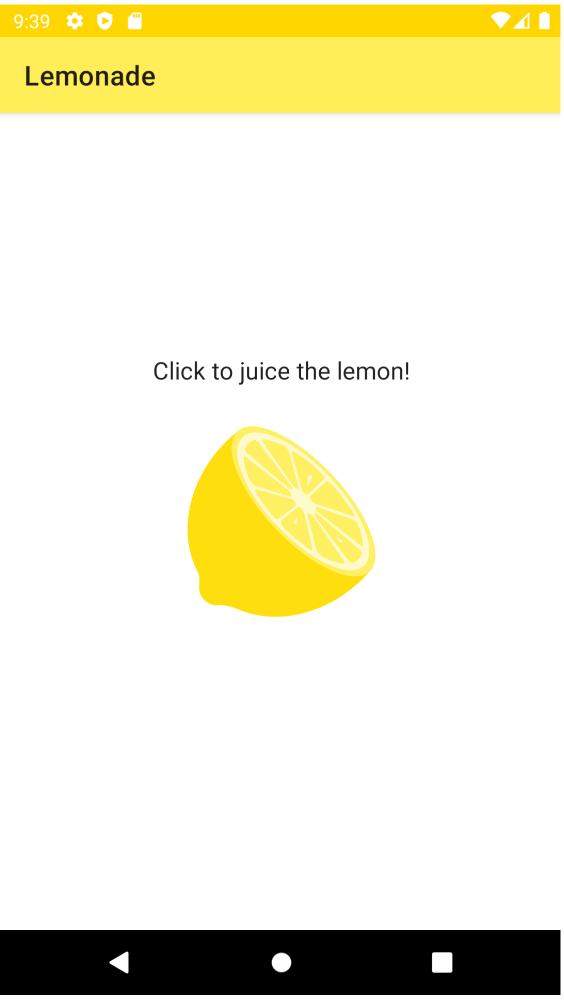


```kotlin
private fun setViewElements() {
    val textAction: TextView = findViewById(R.id.text_action)
    when (lemonadeState) {
        SELECT -> {
            textAction.text = getString(R.string.squeeze_count)  // Set text for SELECT state
            lemonImage!!.setImageResource(R.drawable.lemon_tree)  // Set drawable for SELECT state
        }
        SQUEEZE -> {
            textAction.text = getString(R.string.lemon_squeeze)  // Set text for SQUEEZE state
            lemonImage!!.setImageResource(R.drawable.lemon_squeeze)  // Set drawable for SQUEEZE state
        }
        DRINK -> {
            textAction.text = getString(R.string.lemon_drink)  // Set text for DRINK state
            lemonImage!!.setImageResource(R.drawable.lemon_drink)  // Set drawable for DRINK state
        }
        RESTART -> {
            textAction.text = getString(R.string.lemon_empty_glass)  // Set text for RESTART state
            lemonImage!!.setImageResource(R.drawable.lemon_restart)  // Set drawable for RESTART state
        }
    }
```

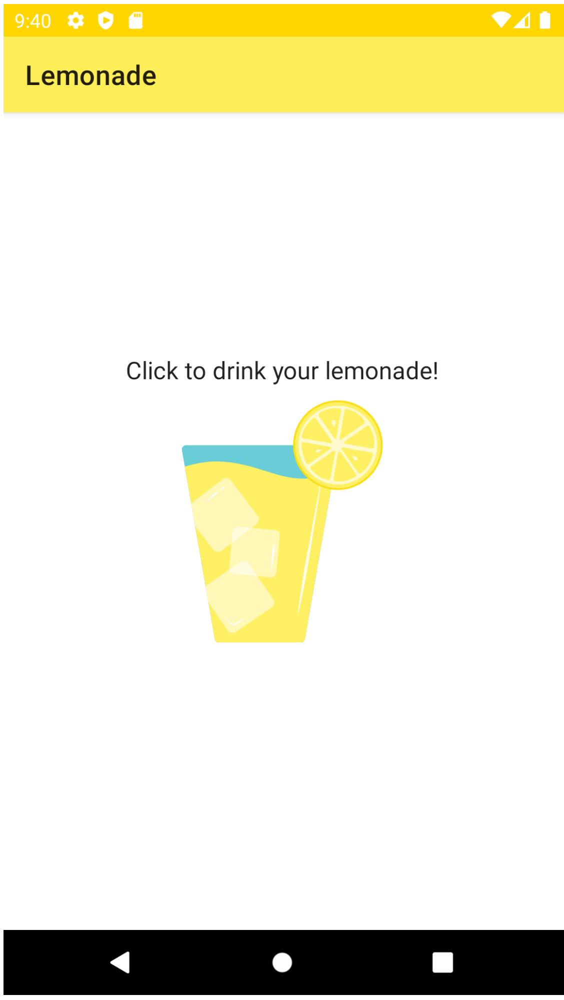
Run your app



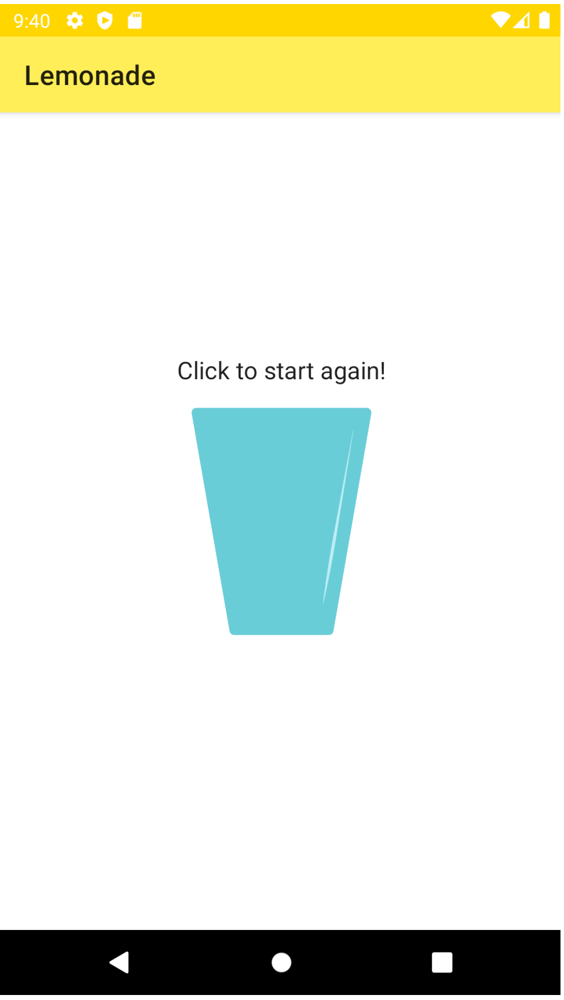
Squeeze count: %1$d, keep squeezing!



Click to juice the lemon!



Click to drink your lemonade!



Click to start again!