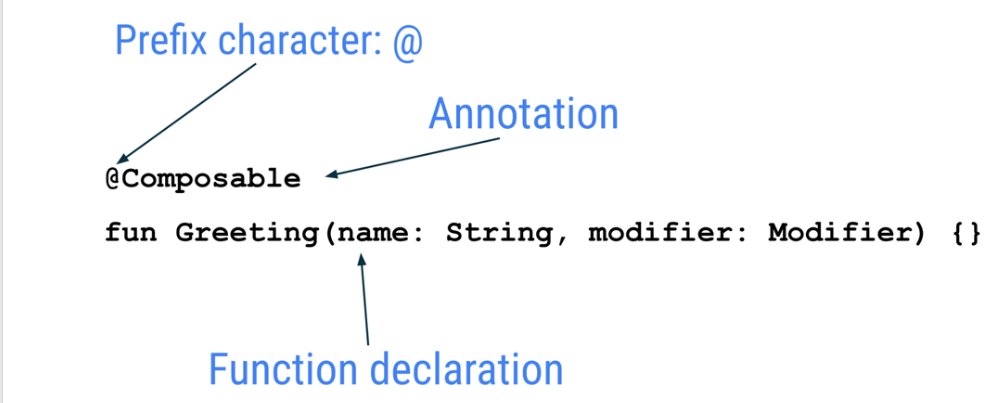




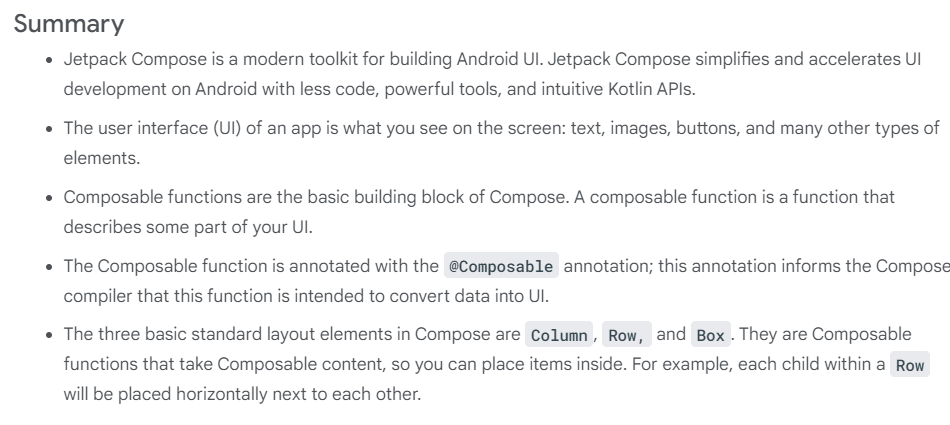
Compose

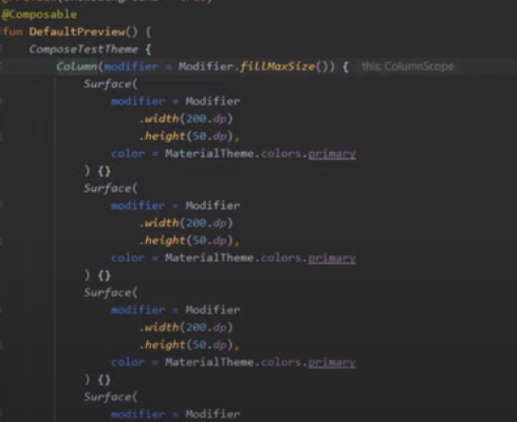


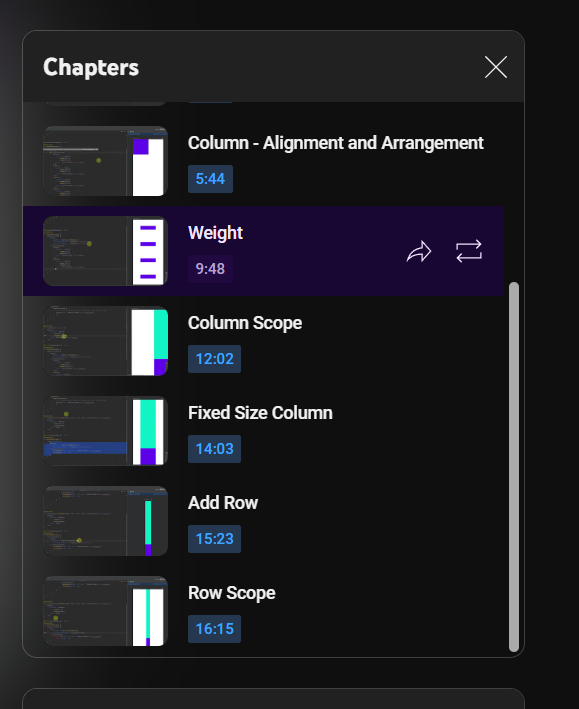


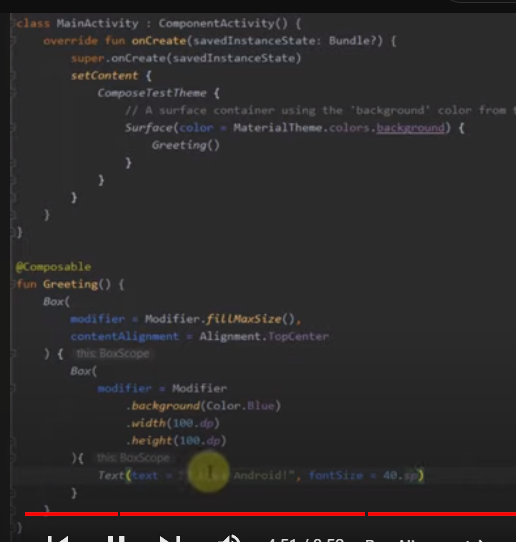
* Jetpack Compose is built around composable functions. These functions let you define your app's UI programmatically by describing how it should look, rather than focusing on the process of the UI's construction. To create a composable function, just add the @Composable annotation to the function name.
* Composable functions can accept arguments, which let the app logic describe or modify the UI. In this case, your UI element accepts a String so that it can greet the user by name.

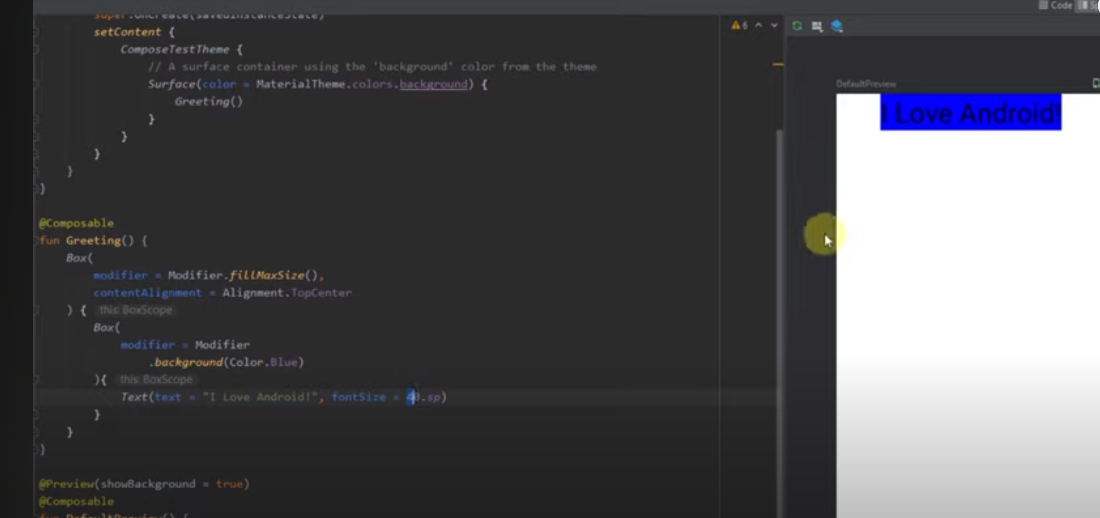


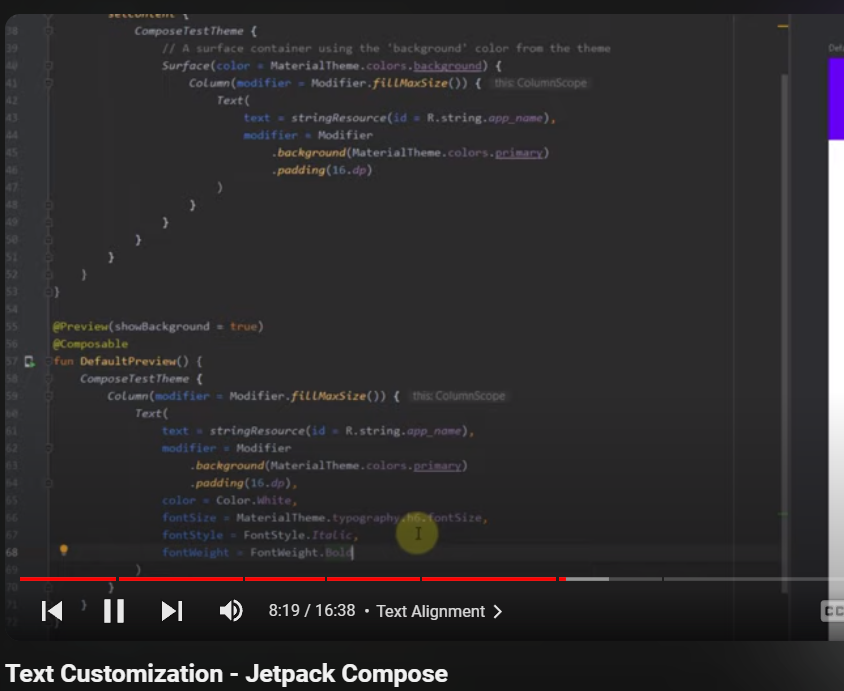


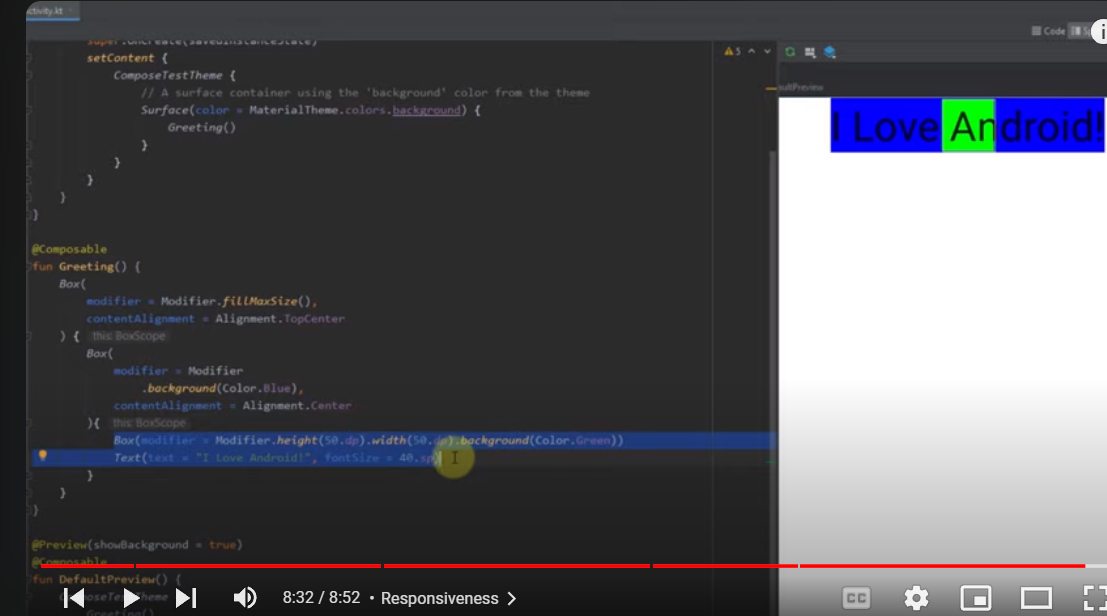


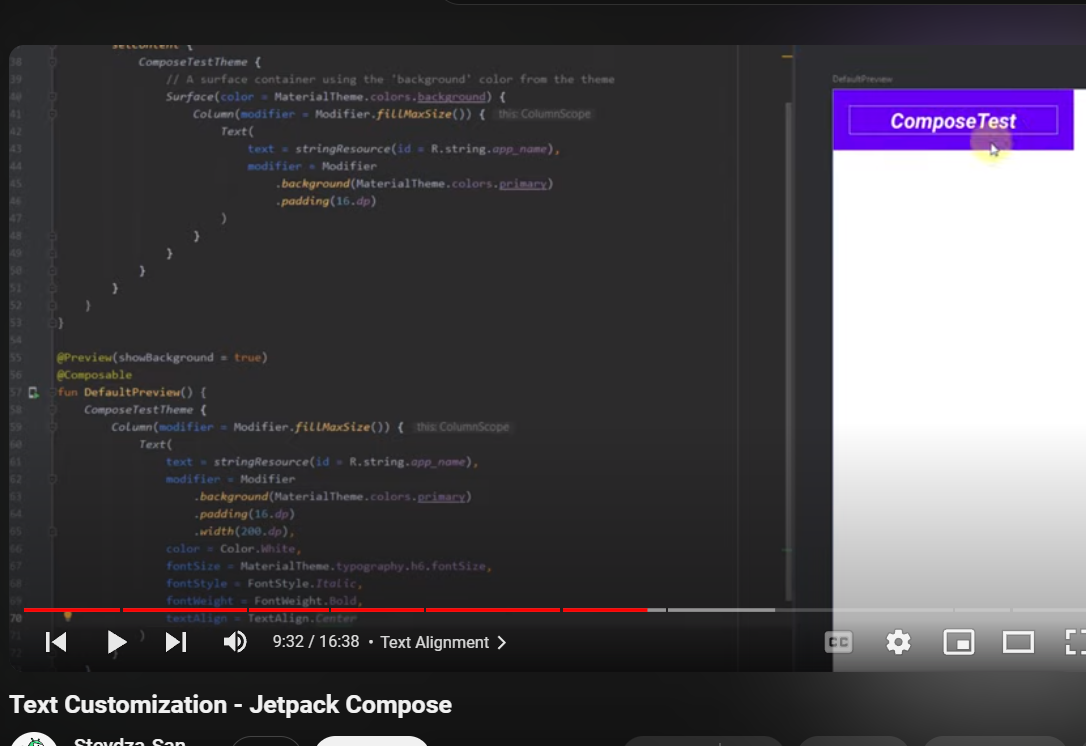


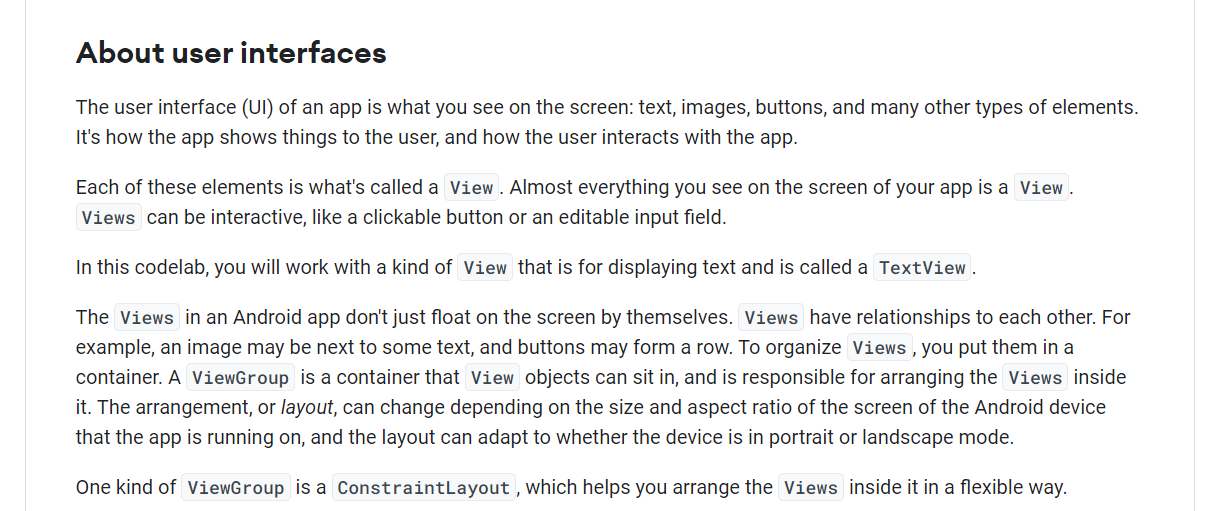


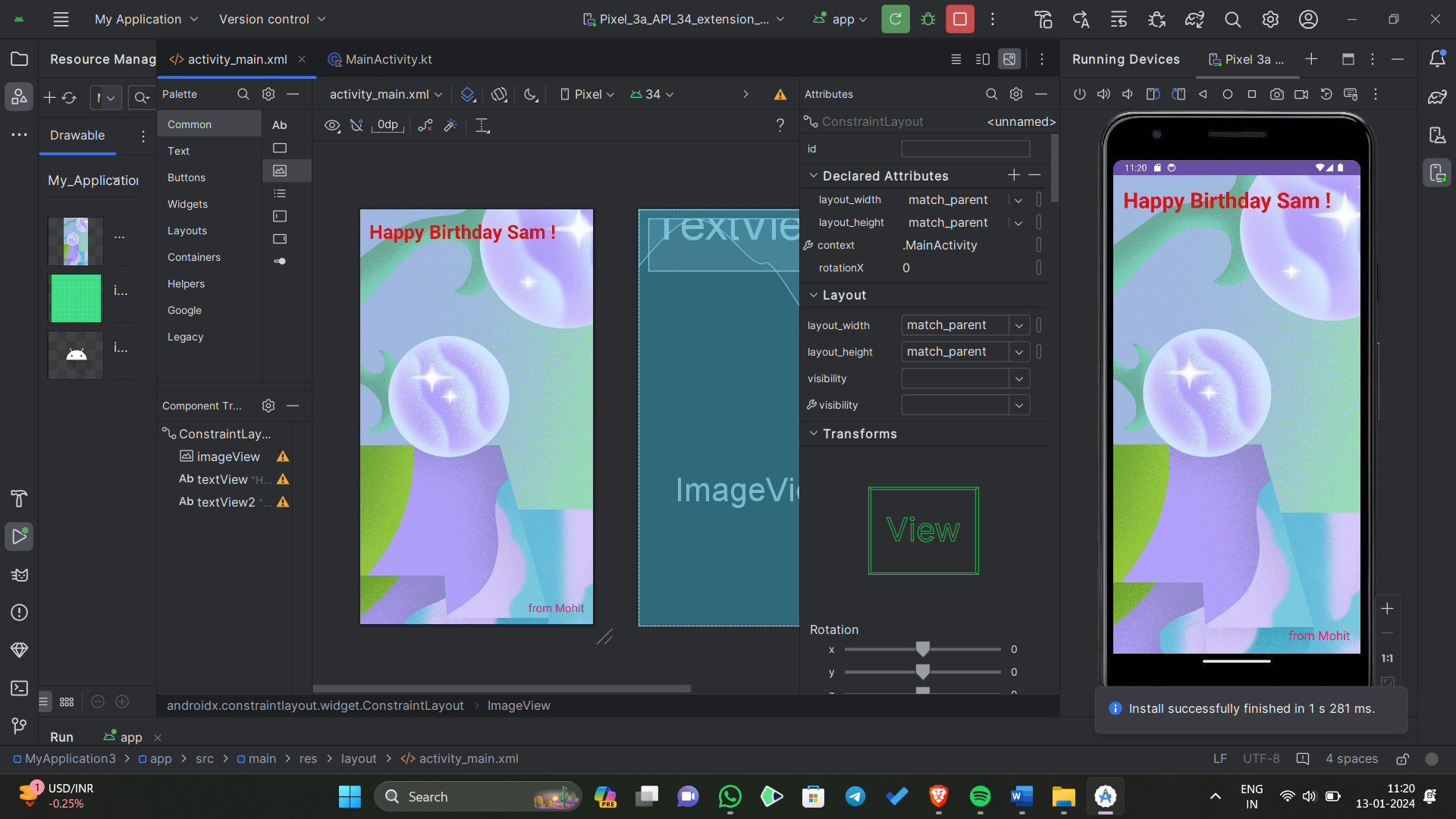












fun main() {

val myFirstDice = Dice(6, "Red")

println("Your ${myFirstDice.numSides} sided ${myFirstDice.color} dice rolled ${myFirstDice.roll()}!")

val mySecondDice = Dice(20, "Blue")

println("Your ${mySecondDice.numSides} sided ${mySecondDice.color} dice rolled ${mySecondDice.roll()}!")

val myCoin = Coin()

println("Your coin landed on ${myCoin.flip()}")

}

class Dice(val numSides: Int, val color: String) {

fun roll(): Int {

return (1..numSides).random()

}

}

class Coin {

fun flip(): String {

val result = if ((1..2).random() == 1) "Heads" else "Tails"

return result

}

}

fun main() {  
    val myFirstDice = Dice(6)  
    val rollResult = myFirstDice.roll()  
    val luckyNumber = 4  
  
    when (rollResult) {  
        luckyNumber -> println("You won!")  
        1 -> println("So sorry! You rolled a 1. Try again!")  
        2 -> println("Sadly, you rolled a 2. Try again!")  
        3 -> println("Unfortunately, you rolled a 3. Try again!")  
        5 -> println("Don't cry! You rolled a 5. Try again!")  
        6 -> println("Apologies! You rolled a 6. Try again!")  
    }  
}

fun main() {

val myFirstDice = Dice(8)

val rollResult = myFirstDice.roll()

val luckyNumber = 4

when (rollResult) {

luckyNumber -> println("You won!")

1 -> println("So sorry! You rolled a 1. Try again!")

2 -> println("Sadly, you rolled a 2. Try again!")

3 -> println("Unfortunately, you rolled a 3. Try again!")

4 -> println("Congratulations! You rolled a 4, which is the lucky number!")

5 -> println("Don't cry! You rolled a 5. Try again!")

6 -> println("Apologies! You rolled a 6. Try again!")

7 -> println("Lucky number 7! You rolled a 7. Try again!")

8 -> println("Wow! You rolled an 8. Try again!")

else -> println("You rolled an unexpected number. Try again!")

}

}

class Dice(val numSides: Int) {

fun roll(): Int {

return (1..numSides).random()

}

}

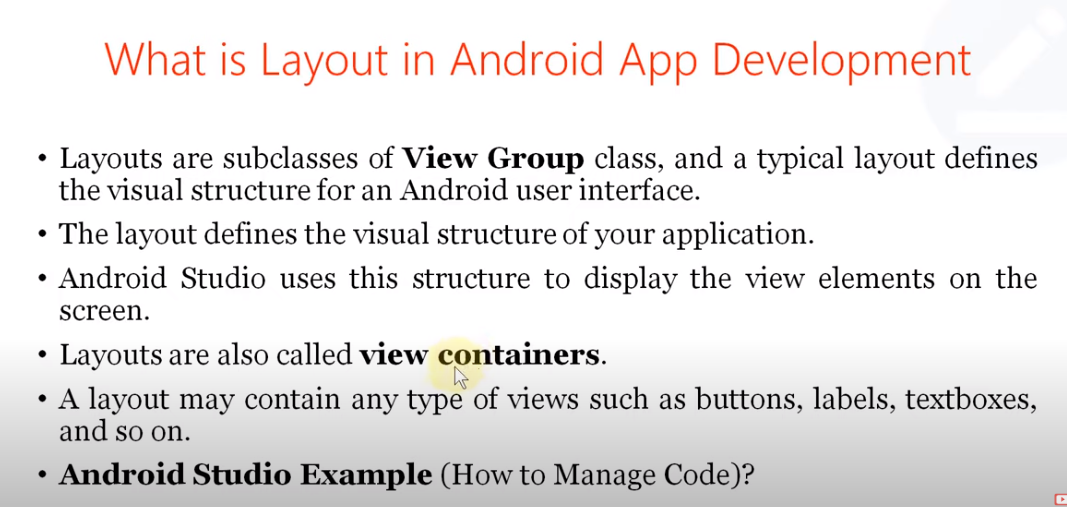
Following the annotation, you have a function declaration, in this case the addition\_isCorrect()function. Inside the function, the assertEquals() function asserts that an expected value should equal an actual value obtained through business logic. Assertion methods are the end goal of a unit test. Ultimately, you want to assert that a result obtained from your code is in a particular state. If the state of the result matches the expected state, the test passes. If the state of the result doesn't match the expected state, the test fails. In this case, the code is comparing two values, so the assertEquals() method takes two parameters—an expected value and an actual value. True to its name, the expected value is what you expect a particular result to be,in this case 4. The actual value represents the result of an actual piece of code. Generally, this would test a piece of code from the app itself. In this case, it's only an arbitrary piece of code, for example, 2 + 2. Without further ado, run this test to see what happens.

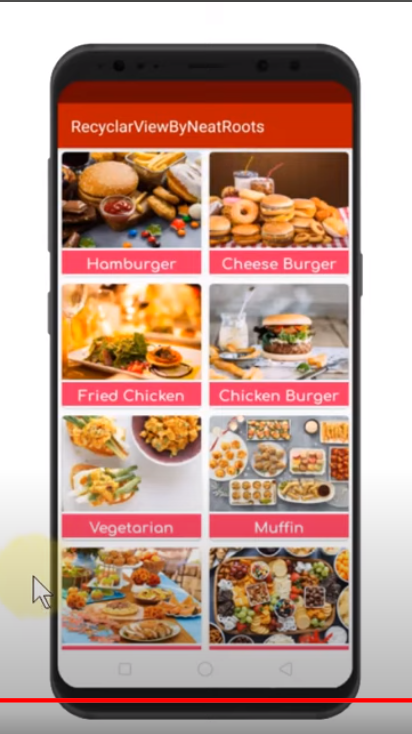
**Note**: There are many assertions in the JUnit library. Some common assertions you might encounter are:

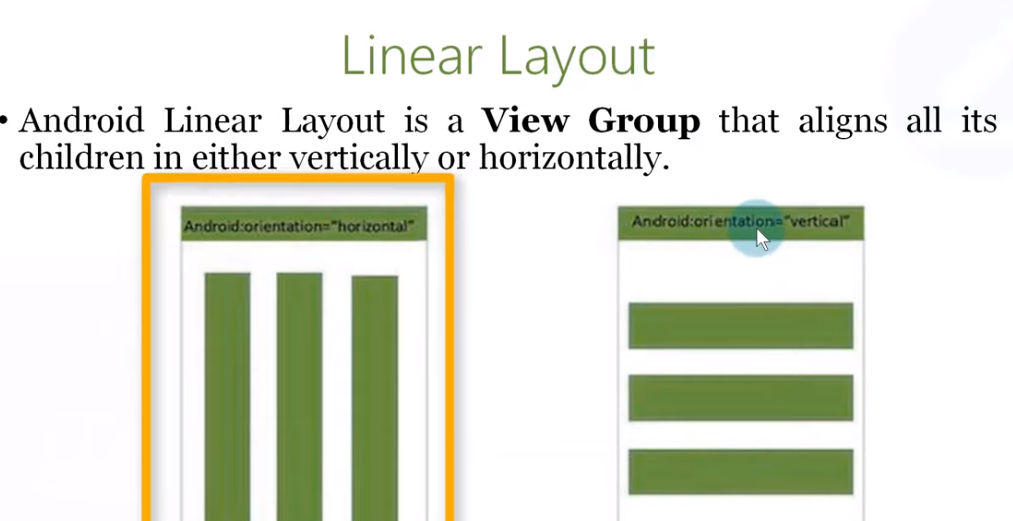
* assertEquals()
* assertNotEquals()
* assertThat()
* assertTrue()
* assertFalse()
* assertNull()
* assertNotNull()

package com.example.myapplication  
  
import org.junit.Assert.assertTrue  
import org.junit.Test  
  
class ExampleUnitTest{  
@Test  
fun generates\_number() {  
 val dice = Dice(8)  
 val rollResult = dice.roll()  
 assertTrue("The value of rollResult was not between 1 and 6", rollResult in 1..6)  
}  
}

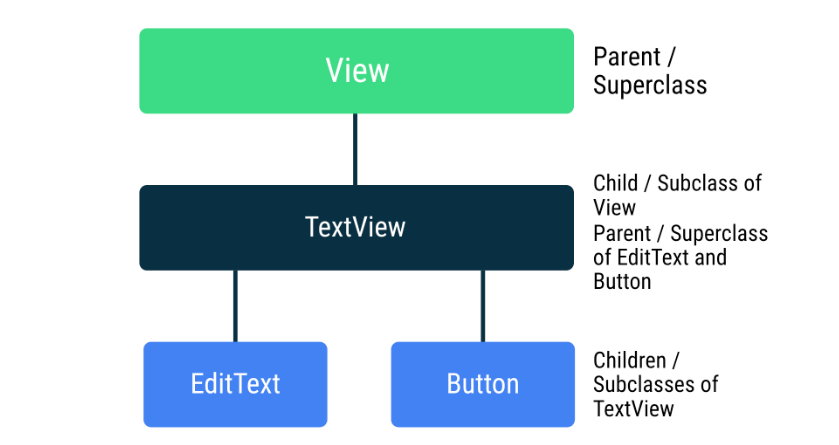
zw

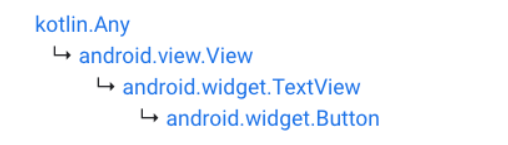
****

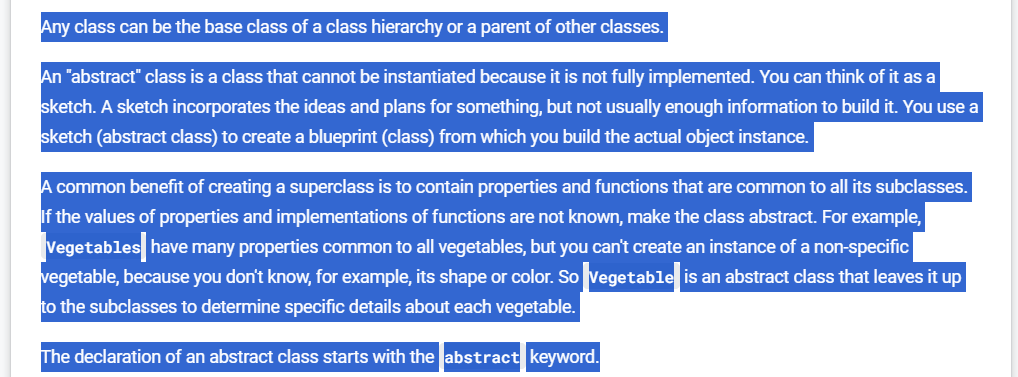
****

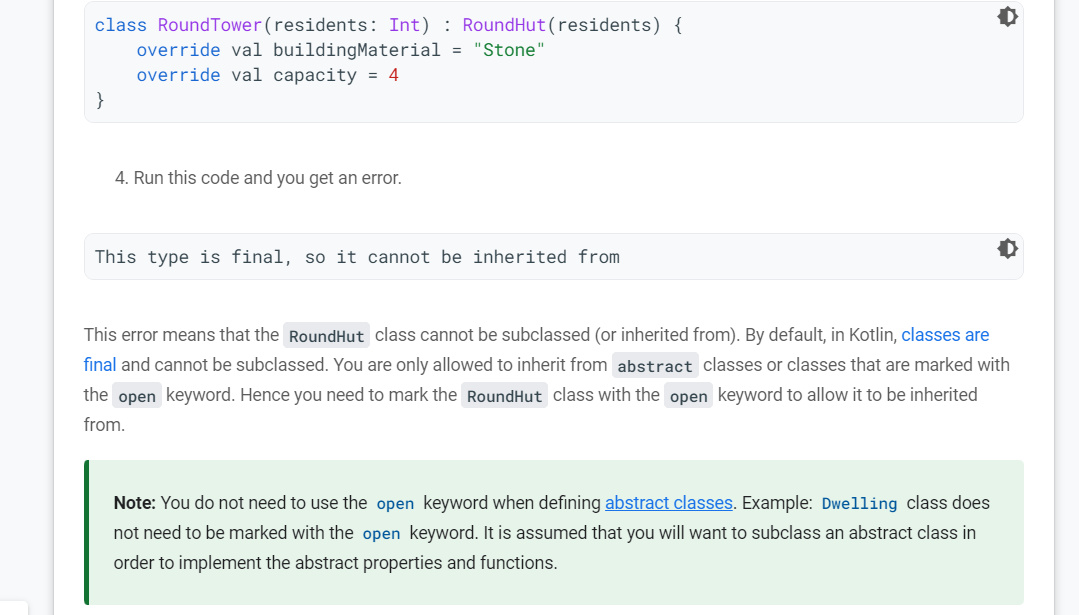
****











fun main() {  
      
    val squareCabin = SquareCabin(6)  
    val roundHut = RoundHut(3)  
    val roundTower = RoundTower(4)  
  
    with(squareCabin) {  
        println("\nSquare Cabin\n============")  
        println("Capacity: ${capacity}")  
        println("Material: ${buildingMaterial}")  
        println("Has room? ${hasRoom()}")  
    }  
         
    with(roundHut) {  
        println("\nRound Hut\n=========")  
        println("Material: ${buildingMaterial}")  
        println("Capacity: ${capacity}")  
        println("Has room? ${hasRoom()}")  
    }  
         
    with(roundTower) {  
        println("\nRound Tower\n==========")  
        println("Material: ${buildingMaterial}")  
        println("Capacity: ${capacity}")  
        println("Has room? ${hasRoom()}")  
    }  
}  
  
abstract class Dwelling(private var residents: Int) {  
    abstract val buildingMaterial: String  
    abstract val capacity: Int  
         
    fun hasRoom(): Boolean {  
       return residents < capacity  
   }  
}  
  
class SquareCabin(residents: Int) : Dwelling(residents) {  
    override val buildingMaterial = "Wood"  
    override val capacity = 6  
}  
  
open class RoundHut(residents: Int) : Dwelling(residents) {  
    override val buildingMaterial = "Straw"  
    override val capacity = 4  
}  
  
class RoundTower(  
    residents: Int,  
    val floors: Int = 2) : RoundHut(residents) {  
  
    override val buildingMaterial = "Stone"  
    override val capacity = 4 \* floors  
}

classes and objects in Kotlin

import kotlin.math.PI

import kotlin.math.sqrt

fun main() {

val squareCabin = SquareCabin(6, 50.0)

val roundHut = RoundHut(3, 10.0)

val roundTower = RoundTower(4, 15.5)

with(squareCabin) {

println("\nSquare Cabin\n============")

println("Capacity: ${capacity}")

println("Material: ${buildingMaterial}")

println("Floor area: ${floorArea()}")

}

with(roundHut) {

println("\nRound Hut\n=========")

println("Material: ${buildingMaterial}")

println("Capacity: ${capacity}")

println("Floor area: ${floorArea()}")

println("Has room? ${hasRoom()}")

getRoom()

println("Has room? ${hasRoom()}")

getRoom()

println("Carpet size: ${calculateMaxCarpetLength()}")

}

with(roundTower) {

println("\nRound Tower\n==========")

println("Material: ${buildingMaterial}")

println("Capacity: ${capacity}")

println("Floor area: ${floorArea()}")

println("Carpet Length: ${calculateMaxCarpetLength()}")

}

}

abstract class Dwelling(private var residents: Int) {

abstract val buildingMaterial: String

abstract val capacity: Int

abstract fun floorArea(): Double

fun hasRoom(): Boolean {

return residents < capacity

}

fun getRoom() {

if (capacity > residents) {

residents++

println("You got a room!")

} else {

println("Sorry, at capacity and no rooms left.")

}

}

}

class SquareCabin(residents: Int, val length: Double) : Dwelling(residents) {

override val buildingMaterial = "Wood"

override val capacity = 6

override fun floorArea(): Double {

return length \* length

}

}

open class RoundHut(

residents: Int, val radius: Double) : Dwelling(residents) {

override val buildingMaterial = "Straw"

override val capacity = 4

override fun floorArea(): Double {

return PI \* radius \* radius

}

fun calculateMaxCarpetLength(): Double {

return sqrt(2.0) \* radius

}

}

class RoundTower(

residents: Int,

radius: Double,

val floors: Int = 2) : RoundHut(residents, radius) {

override val buildingMaterial = "Stone"

override val capacity = floors \* 4

override fun floorArea(): Double {

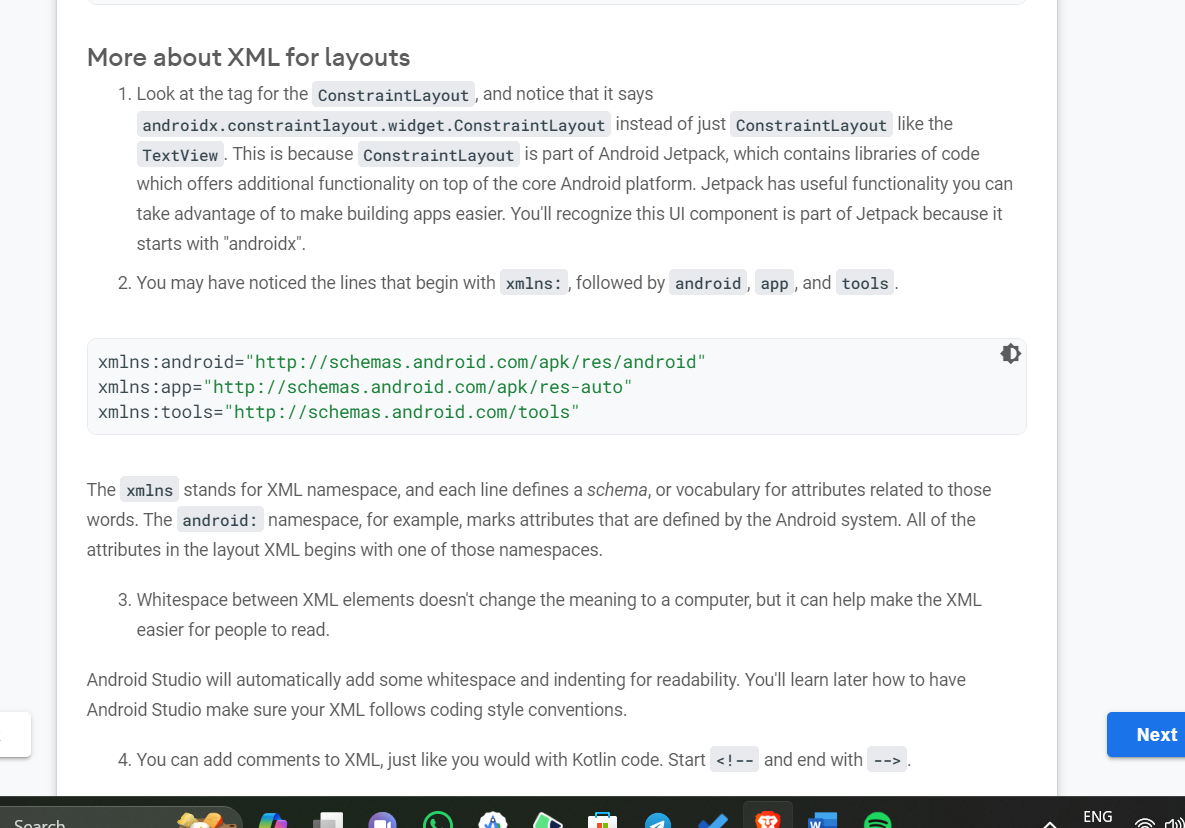
return super.floorArea() \* floors

}

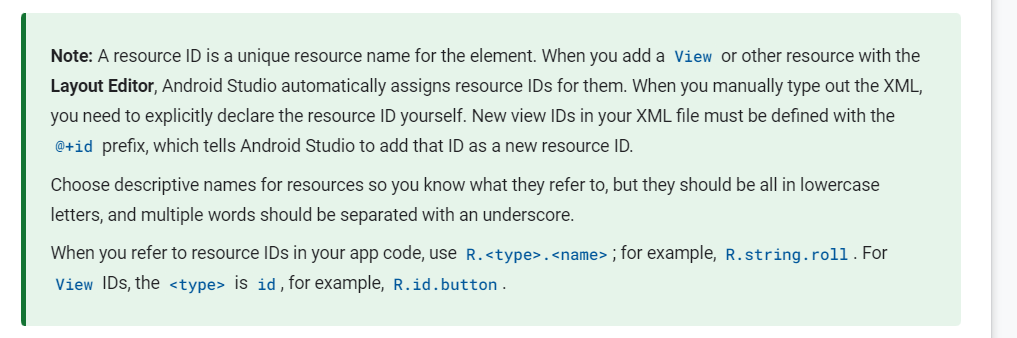
}

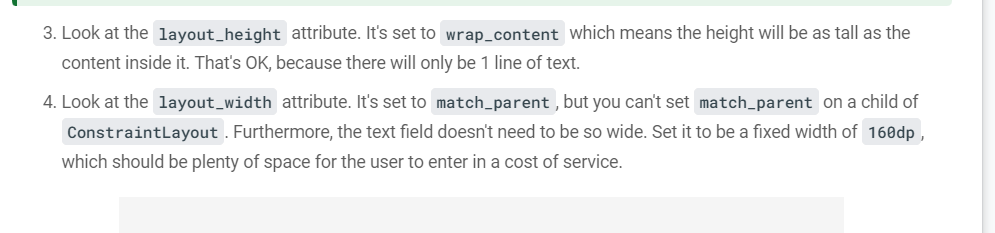
In this codelab you learned how to:

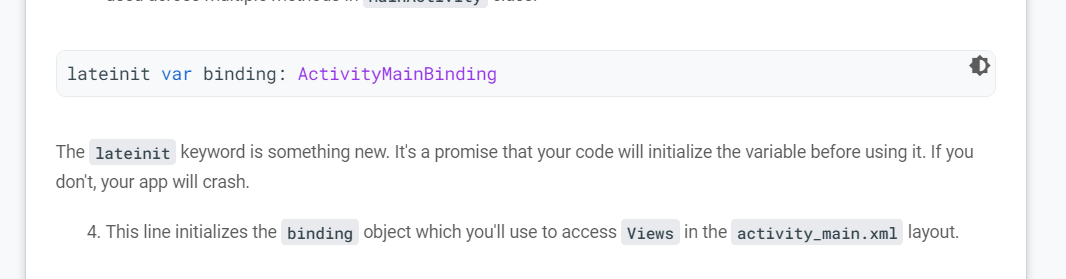
* Create a class hierarchy, that is a tree of classes where children inherit functionality from parent classes. Properties and functions are inherited by subclasses.
* Create an abstract class where some functionality is left to be implemented by its subclasses. An abstract class can therefore not be instantiated.
* Create subclasses of an abstract class.
* Use override keyword to override properties and functions in subclasses.
* Use the super keyword to reference functions and properties in the parent class.
* Make a class open so that it can be subclassed.
* Make a property private, so it can only be used inside the class.
* Use the with construct to make multiple calls on the same object instance.
* Import functionality from the kotlin.math library

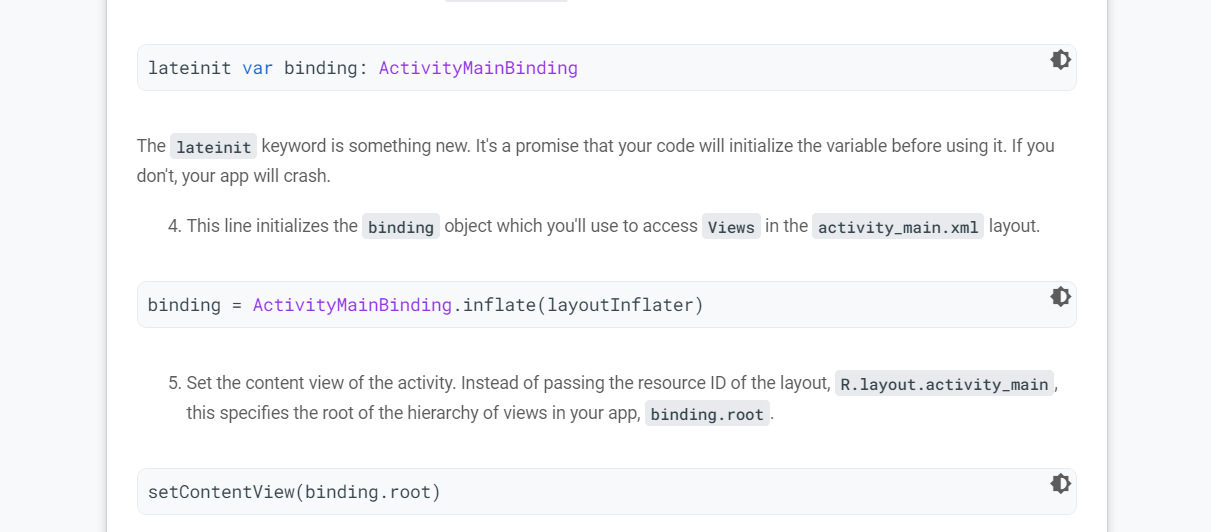
****

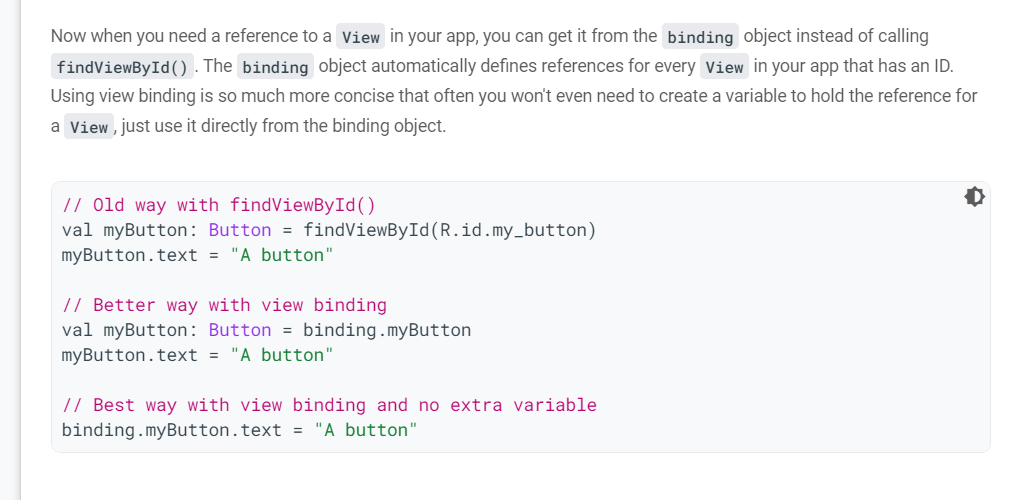
****

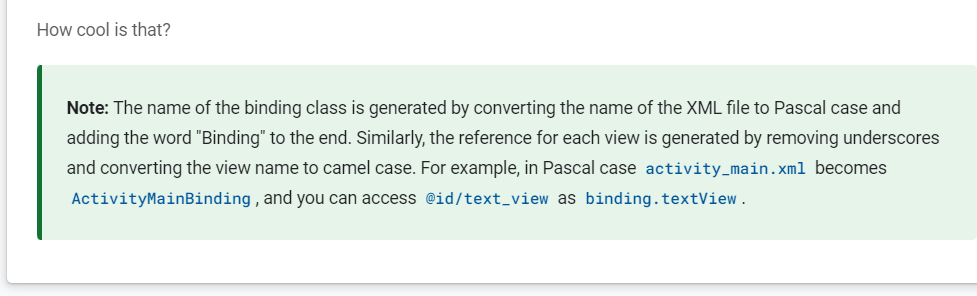
****

****

****

****

****

****

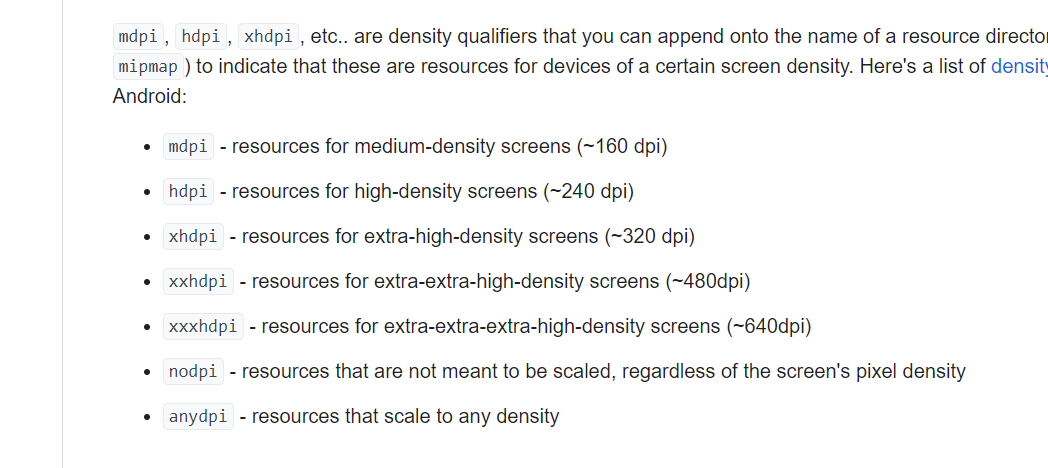
* View binding lets you more easily write code that interacts with the UI elements in your app.
* The Double data type in Kotlin can store a decimal number.
* Use the checkedRadioButtonId attribute of a RadioGroup to find which RadioButton is selected.
* Use NumberFormat.getCurrencyInstance() to get a formatter to use for formatting numbers as currency.
* You can use string parameters like %s to create dynamic strings that can still be easily translated into other languages.
* Testing is important!
* You can use **Logcat** in Android Studio to troubleshoot problems like the app crashing.
* A stack trace shows a list of methods that were called. This can be useful if the code generates an exception.
* Exceptions indicate a problem that code didn't expect.
* Null means "no value."
* Not all code can handle null values, so be careful using it.
* Use **Analyze > Inspect Code** for suggestions to improve your code.

Explanation of the binding part:

1. **binding: ActivityMainBinding**: This variable is declared to hold the reference to the generated binding class **ActivityMainBinding**, which is automatically generated by the View Binding feature in Android. It provides direct references to the views in your layout.
2. **binding = ActivityMainBinding.inflate(layoutInflater)**: This line inflates the layout using the **ActivityMainBinding**. **ActivityMainBinding.inflate(layoutInflater)** creates an instance of the binding class, and **setContentView(binding.root)** sets the content view of the activity to the root view of the inflated layout.
3. **binding.calculateButton.setOnClickListener { calculateTip() }**: This line sets a click listener on the "Calculate" button using the **calculateTip()** function. It uses the **binding** reference to directly access the views defined in the layout file.

In summary, the binding part is leveraging View Binding to directly reference views from the layout file (**costOfService**, **tipOptions**, **roundUpSwitch**, etc.) in a type-safe manner, making it easier to manipulate and interact with UI elements in your activity.

 It is **used to bind the views to the code in** other words we can simply say it replaces findViewById in android which reduces the boilerplate code .

****

**fun main() {**

**val numbers = listOf(1, 2, 3, 4, 5, 6)**

**println("List: $numbers")**

**println("Size: ${numbers.size}")**

**// Access elements of the list**

**println("First element: ${numbers[0]}")**

**println("Second element: ${numbers[1]}")**

**println("Last index: ${numbers.size - 1}")**

**println("Last element: ${numbers[numbers.size - 1]}")**

**println("First: ${numbers.first()}")**

**println("Last: ${numbers.last()}")**

**// Use the contains() method**

**println("Contains 4? ${numbers.contains(4)}")**

**println("Contains 7? ${numbers.contains(7)}")**

**val colors = listOf("green", "orange", "blue")**

**println("Reversed list: ${colors.reversed()}")**

**println("List: $colors")**

**}**

fun main() {  
    val entrees = mutableListOf<String>()  
    println("Entrees: $entrees")  
  
    // Add individual items using add()  
    println("Add noodles: ${entrees.add("noodles")}")  
    println("Entrees: $entrees")  
    println("Add spaghetti: ${entrees.add("spaghetti")}")  
    println("Entrees: $entrees")  
  
    // Add a list of items using addAll()  
    val moreItems = listOf("ravioli", "lasagna", "fettuccine")  
    println("Add list: ${entrees.addAll(moreItems)}")  
    println("Entrees: $entrees")  
  
    // Remove an item using remove()  
    println("Remove spaghetti: ${entrees.remove("spaghetti")}")  
    println("Entrees: $entrees")  
    println("Remove item that doesn't exist: ${entrees.remove("rice")}")  
    println("Entrees: $entrees")  
  
    // Remove an item using removeAt() with an index  
    println("Remove first element: ${entrees.removeAt(0)}")  
    println("Entrees: $entrees")  
  
    // Clear out the list  
    entrees.clear()  
    println("Entrees: $entrees")  
  
    // Check if the list is empty  
    println("Empty? ${entrees.isEmpty()}")  
}