

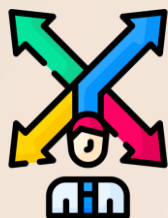
JAVA vs KOTLIN



Maturity



Performance



Versatility



Java™

The Complete Android Developer Course

JAVA vs KOTLIN



Conciseness



Null Safety



Modern Features

The Complete Android Developer Course

JAVA vs KOTLIN



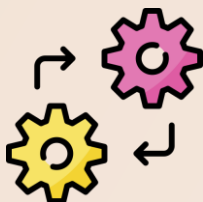
Syntax



Null Safety



Lambda Expressions



Coroutines



Java™

The Complete Android Developer Course

Your Choice?



- Project Requirements
- Team Expertise
- Personal Preferences



The Complete Android Developer Course

KOTLIN SYNTAX

In Kotlin, “fun main() { }” is the entry point of a Kotlin Program.

It serves as the starting point of execution when you run a Kotlin App.

A code editor snippet with a dark background and a yellow dashed border. On the left, there is a green play button icon and a vertical line with a small house icon at the bottom. The code is written in a monospaced font with syntax highlighting: 'fun' is orange, 'main()' is blue, '{' is white, 'println' is green, and the string 'Hello Kotlin Developers' is in quotes. The closing brace '}' is white.

```
fun main(){  
    println("Hello Kotlin Developers")  
}
```

Kotlin Variables

In Kotlin, variables are used to store information that you want to use later in your program.



The Complete Android Developer Course

Kotlin Variables

It's like writing your age on a piece of paper and putting it in the box.



The Complete Android Developer Course

Kotlin Variables

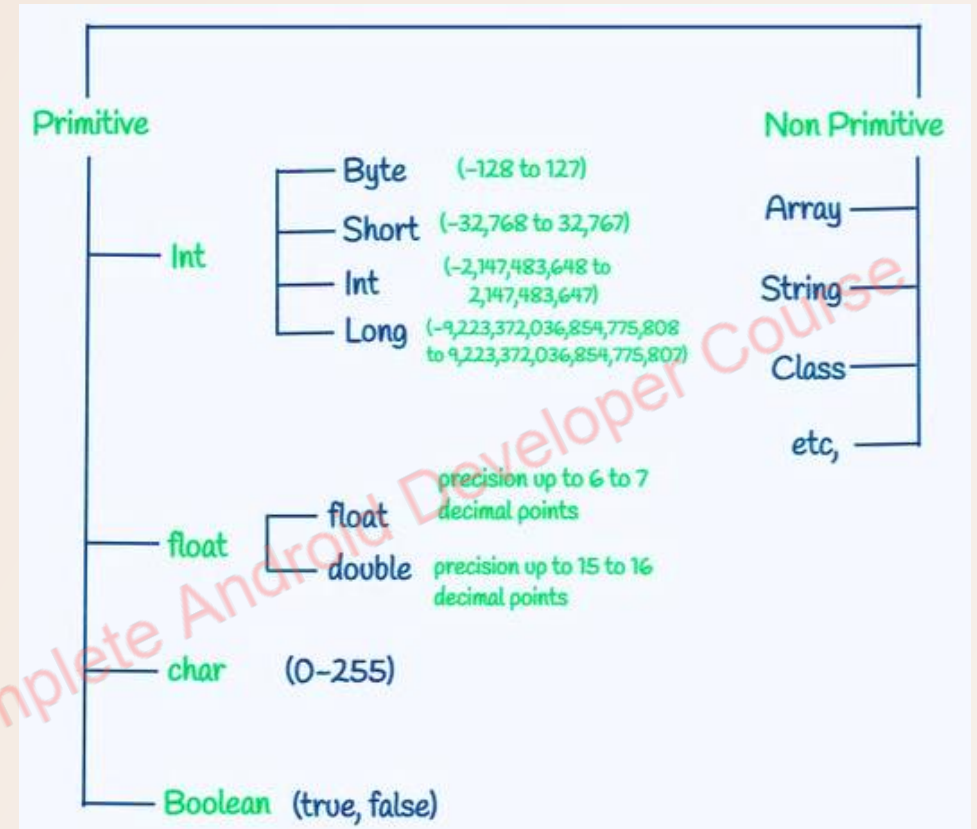


The Complete Android Developer Course

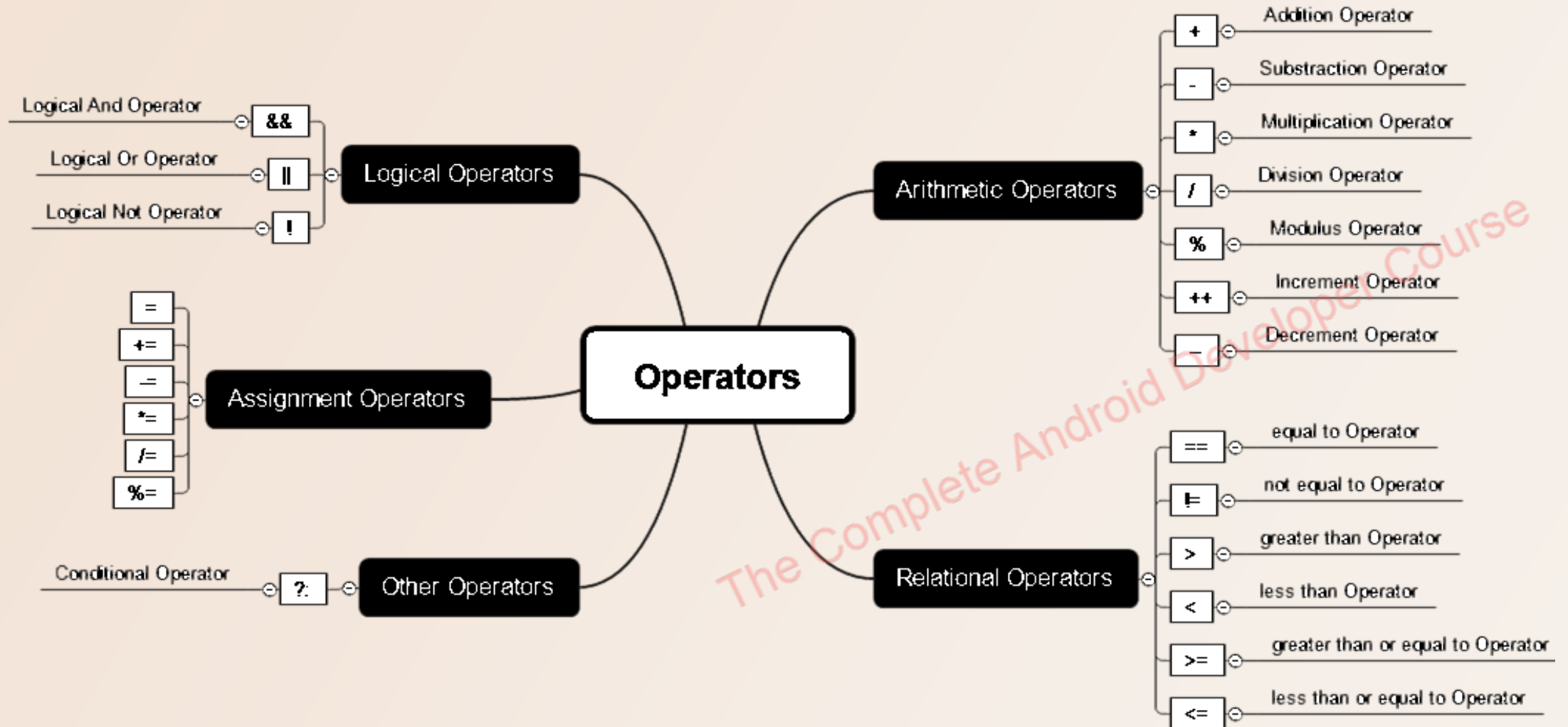
Kotlin Data Types

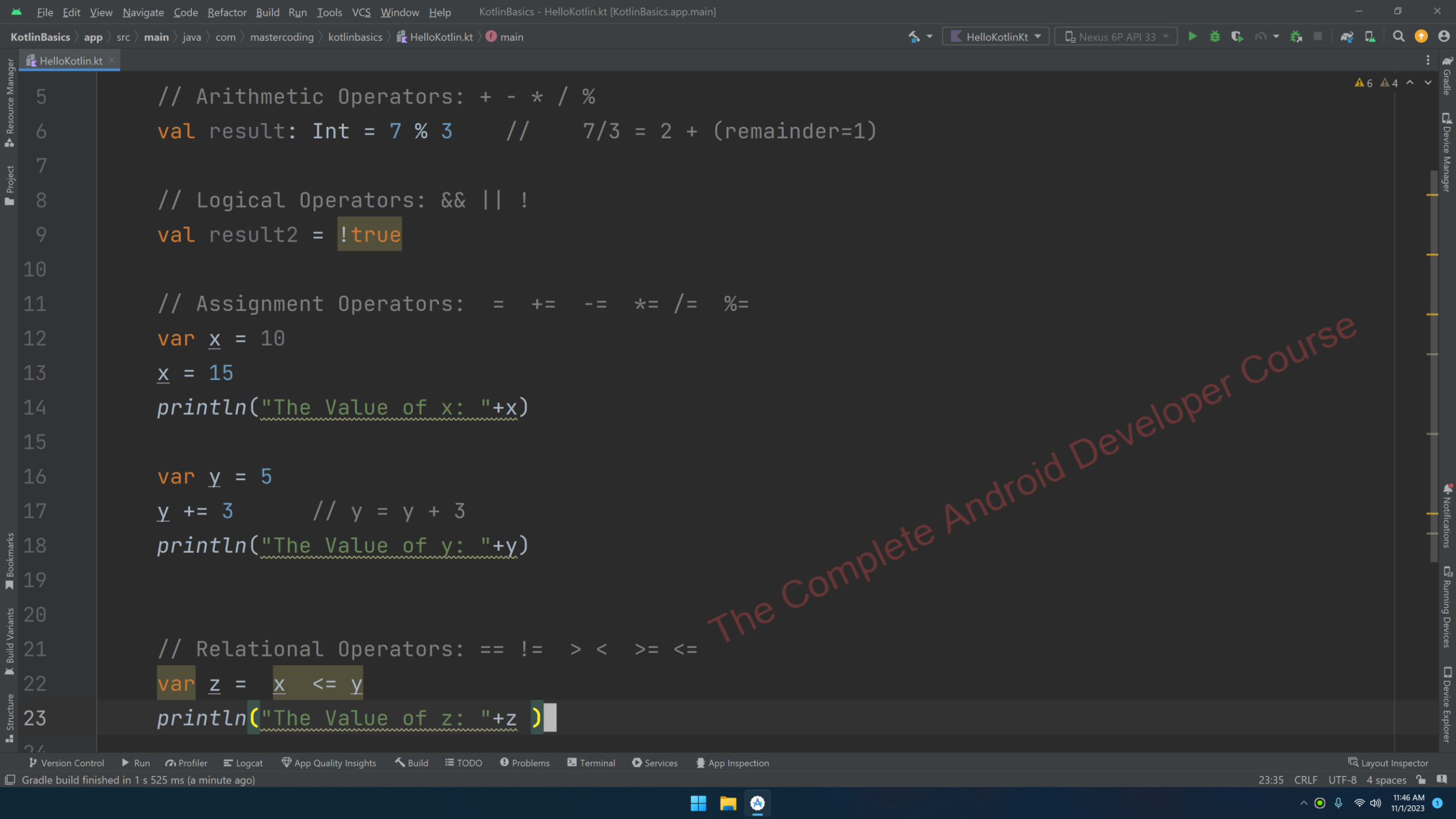
Data types are used to define the type of data that a variable can hold.

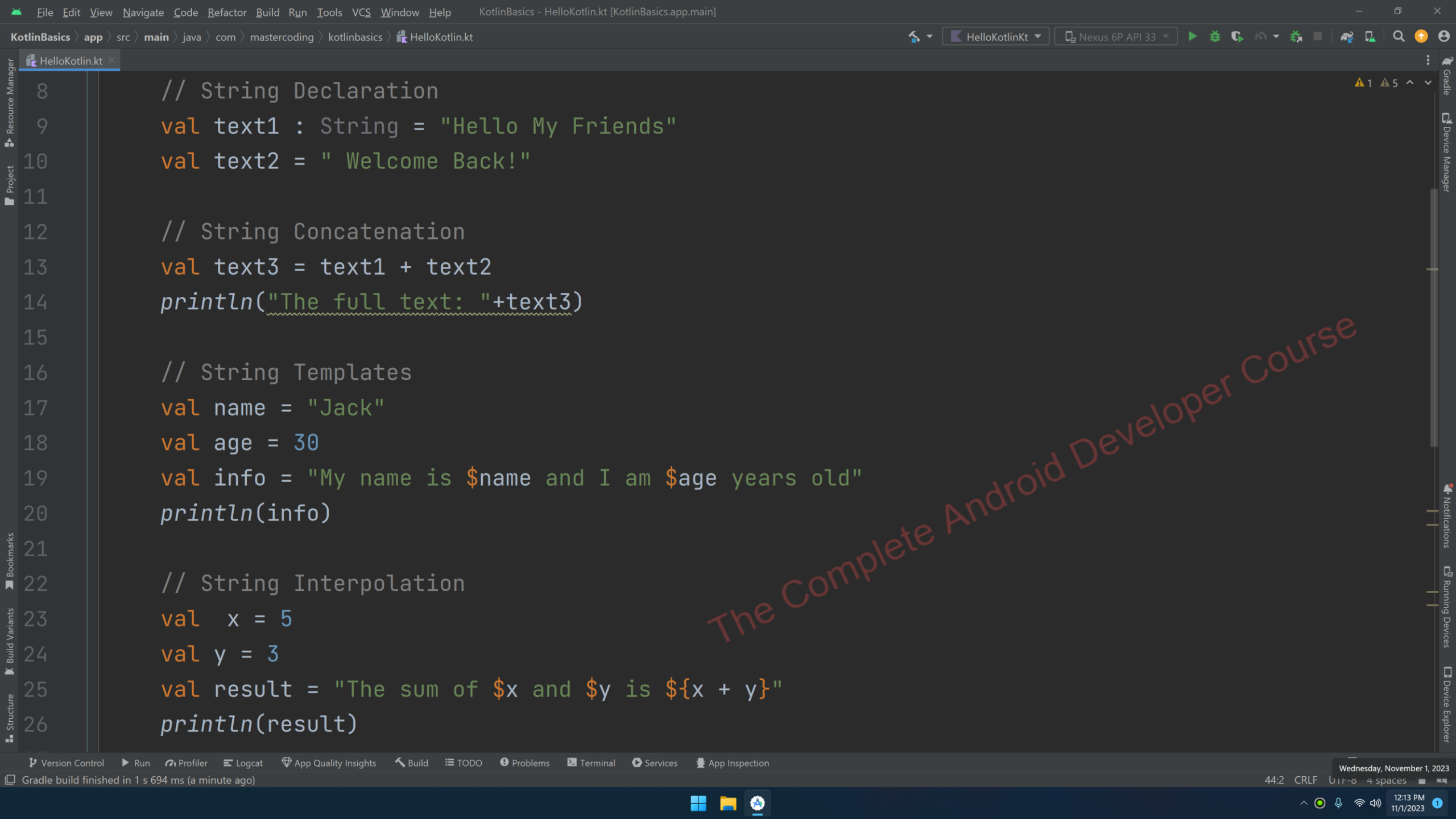
Each data type has specific characteristics and uses.

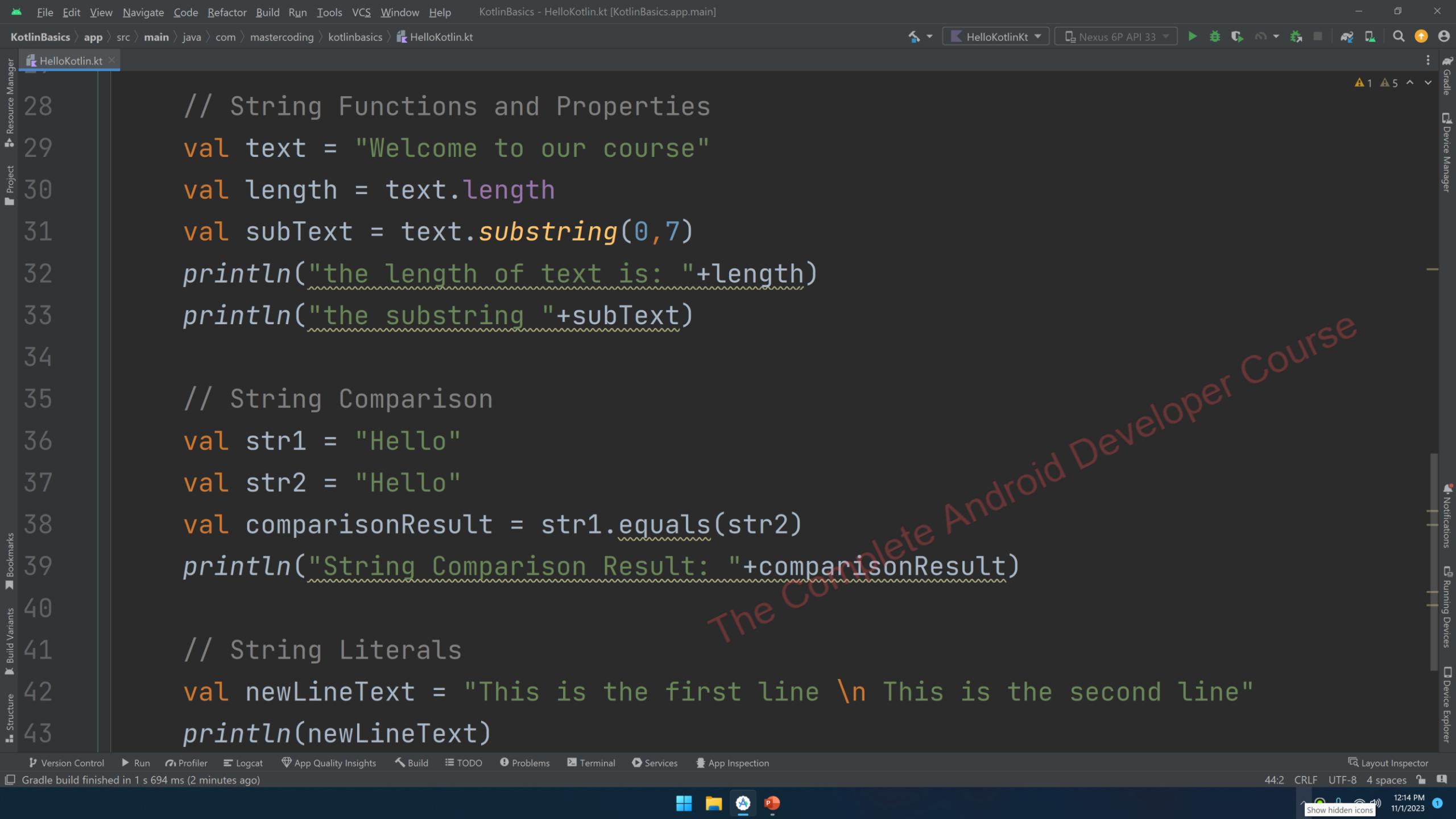


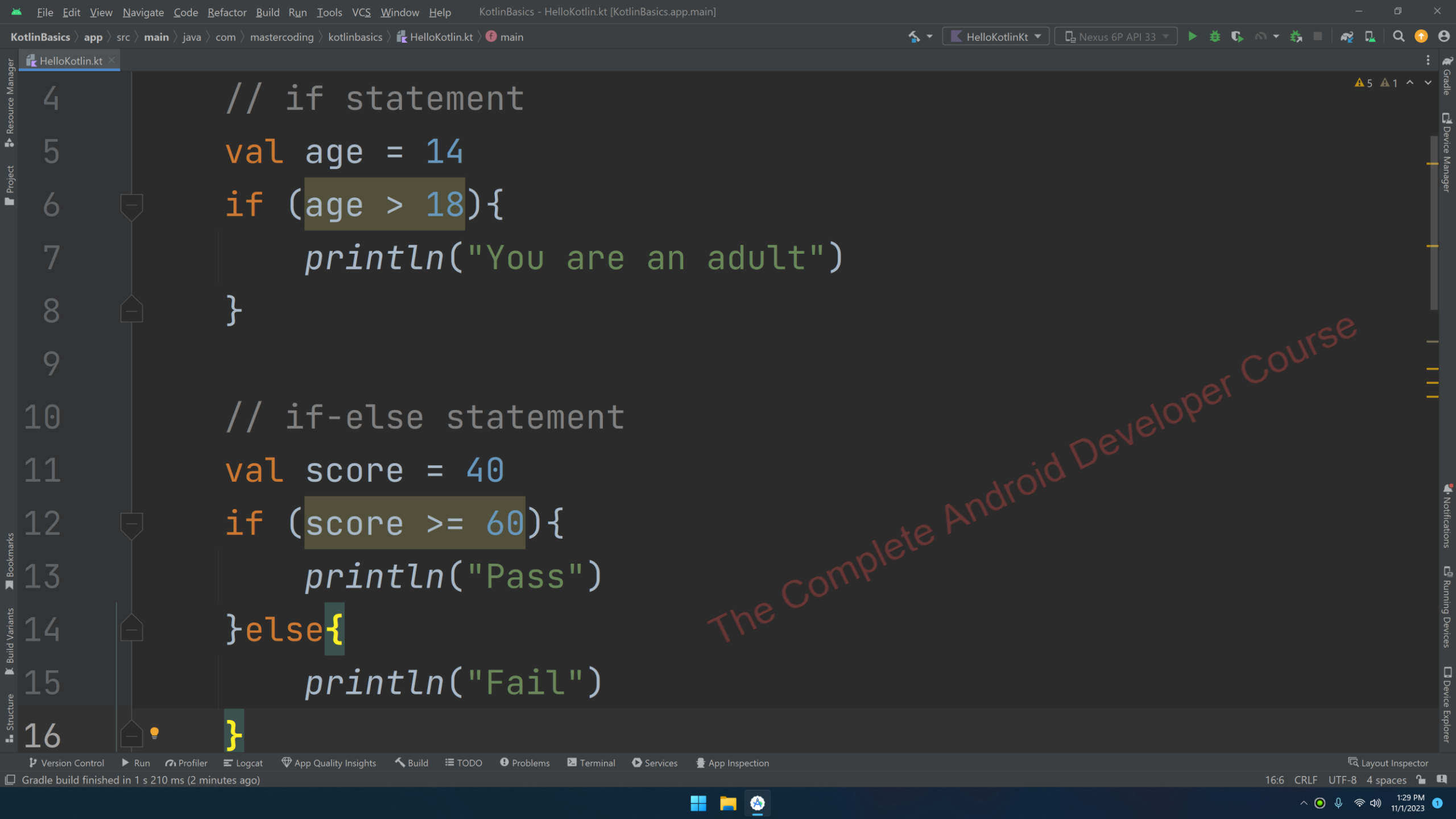
Kotlin Operators





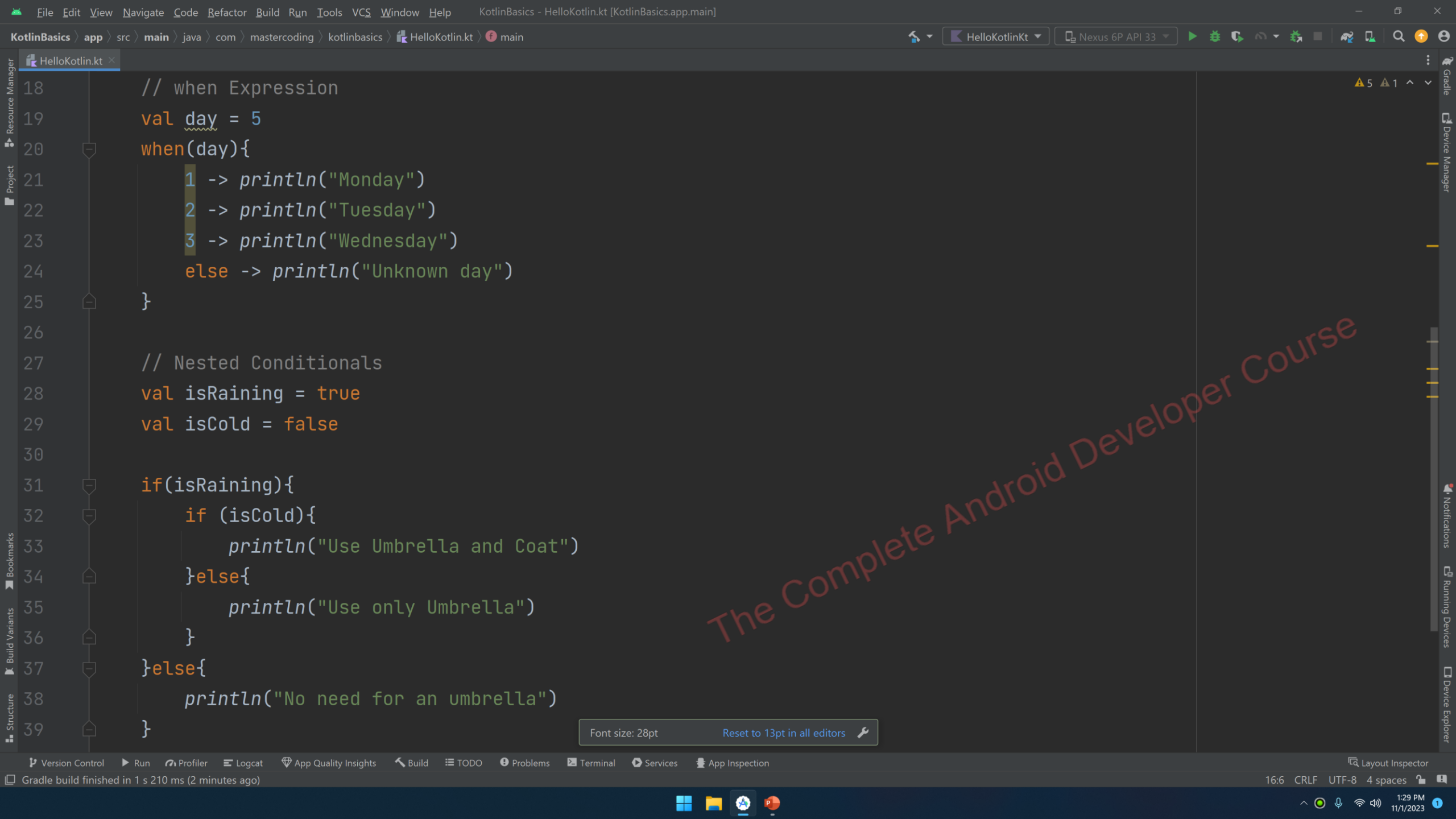


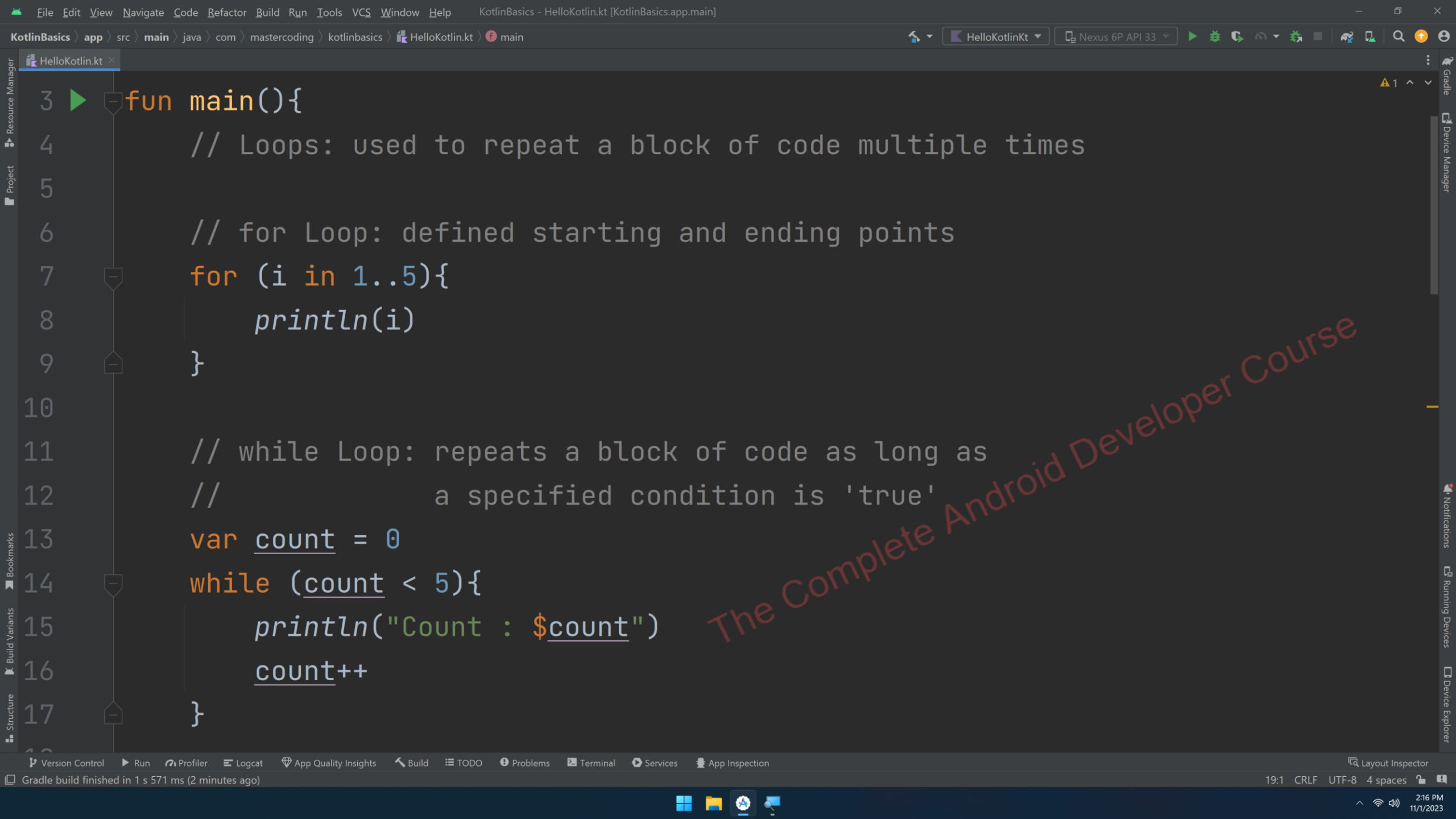


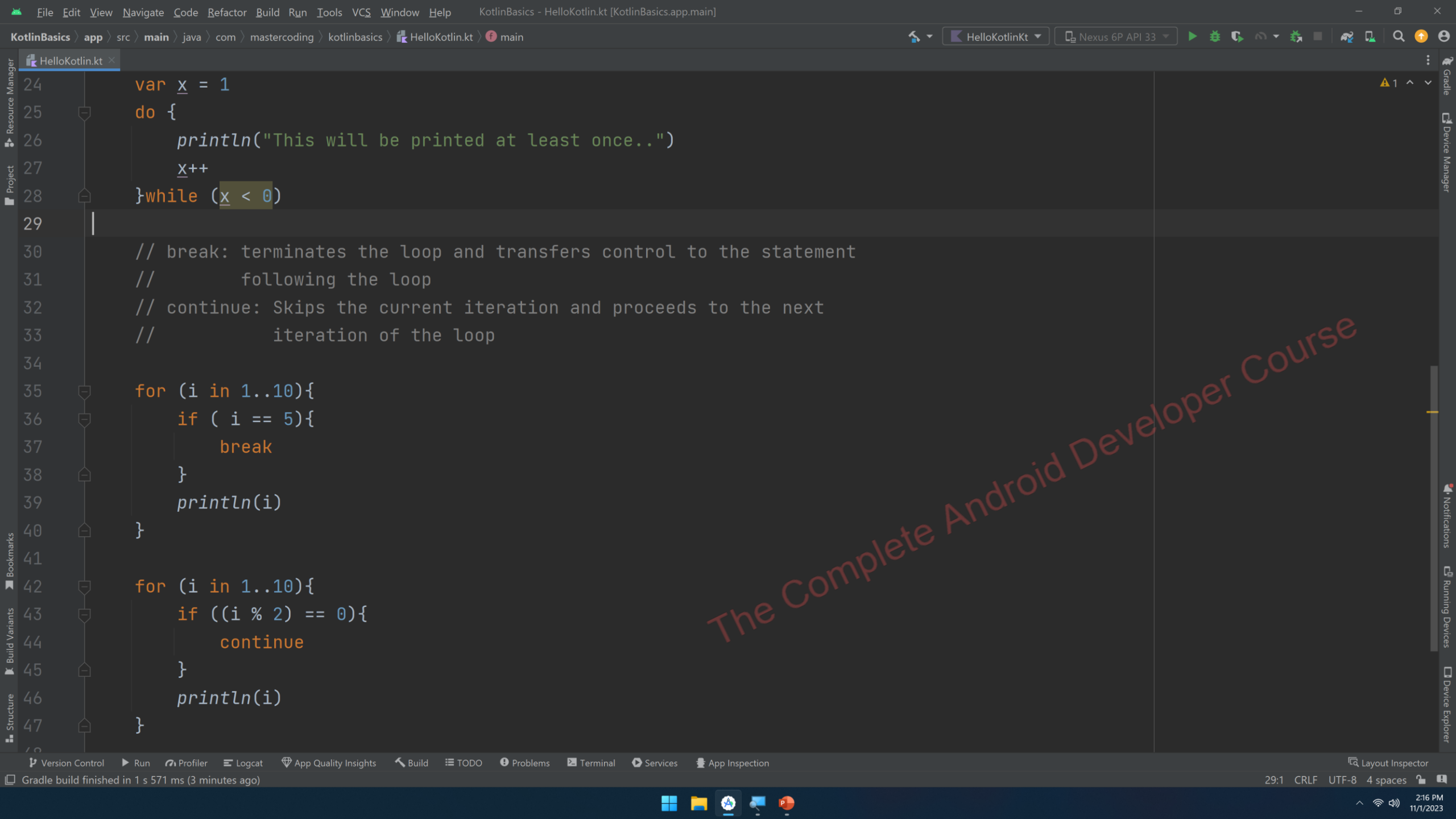


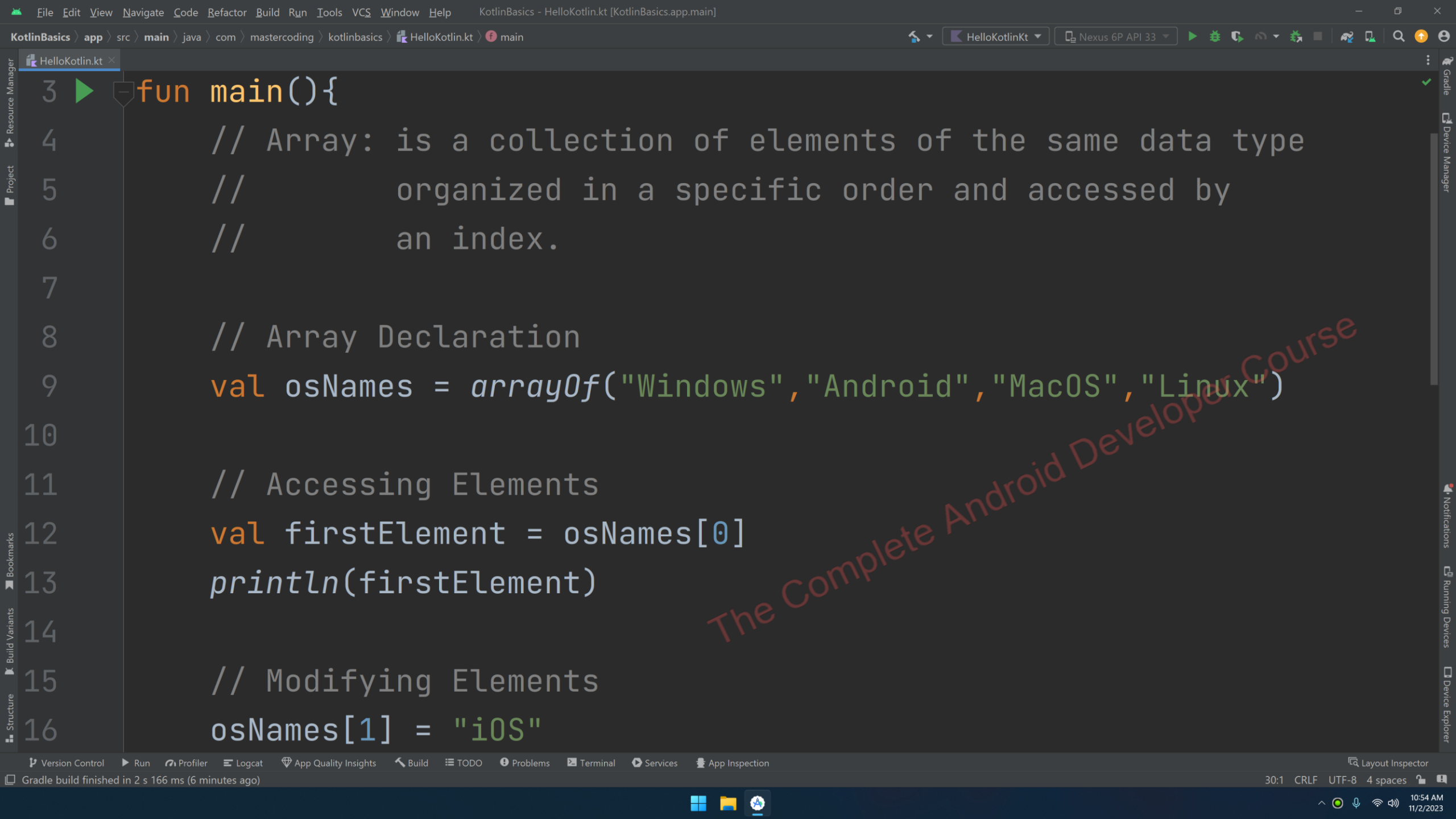
```
// if statement
val age = 14
if (age > 18){
    println("You are an adult")
}

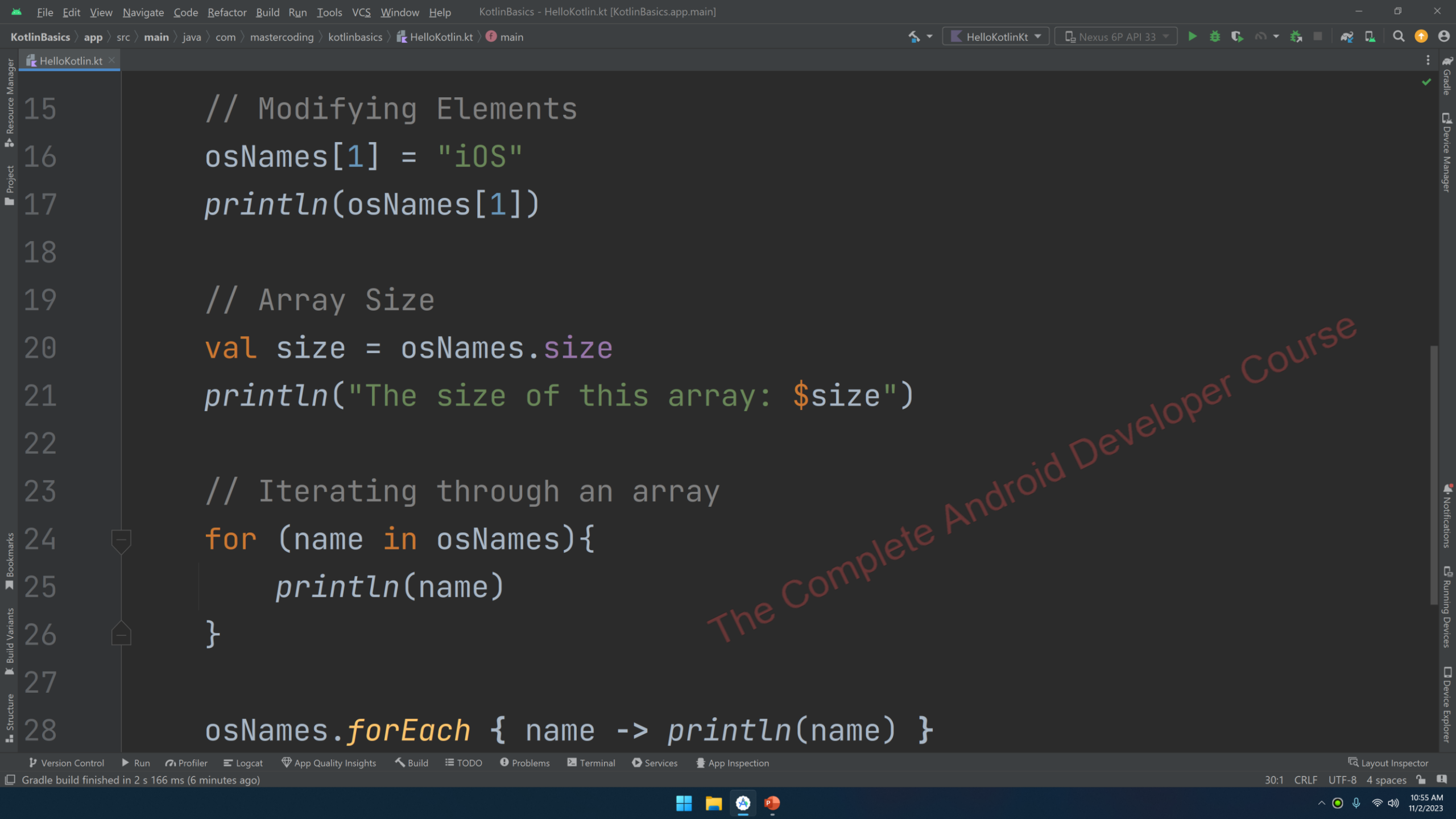
// if-else statement
val score = 40
if (score >= 60){
    println("Pass")
}else{
    println("Fail")
}
```

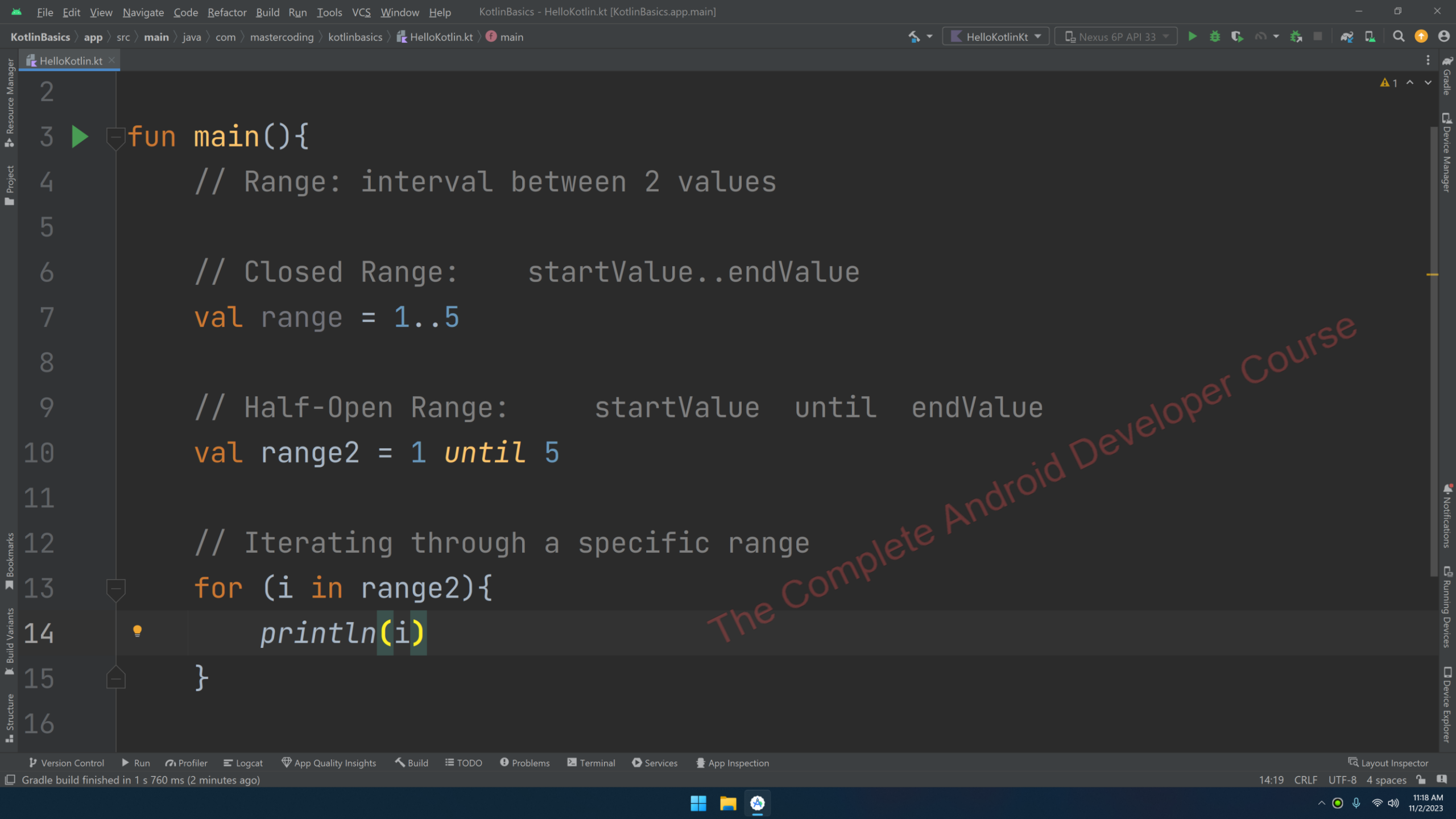




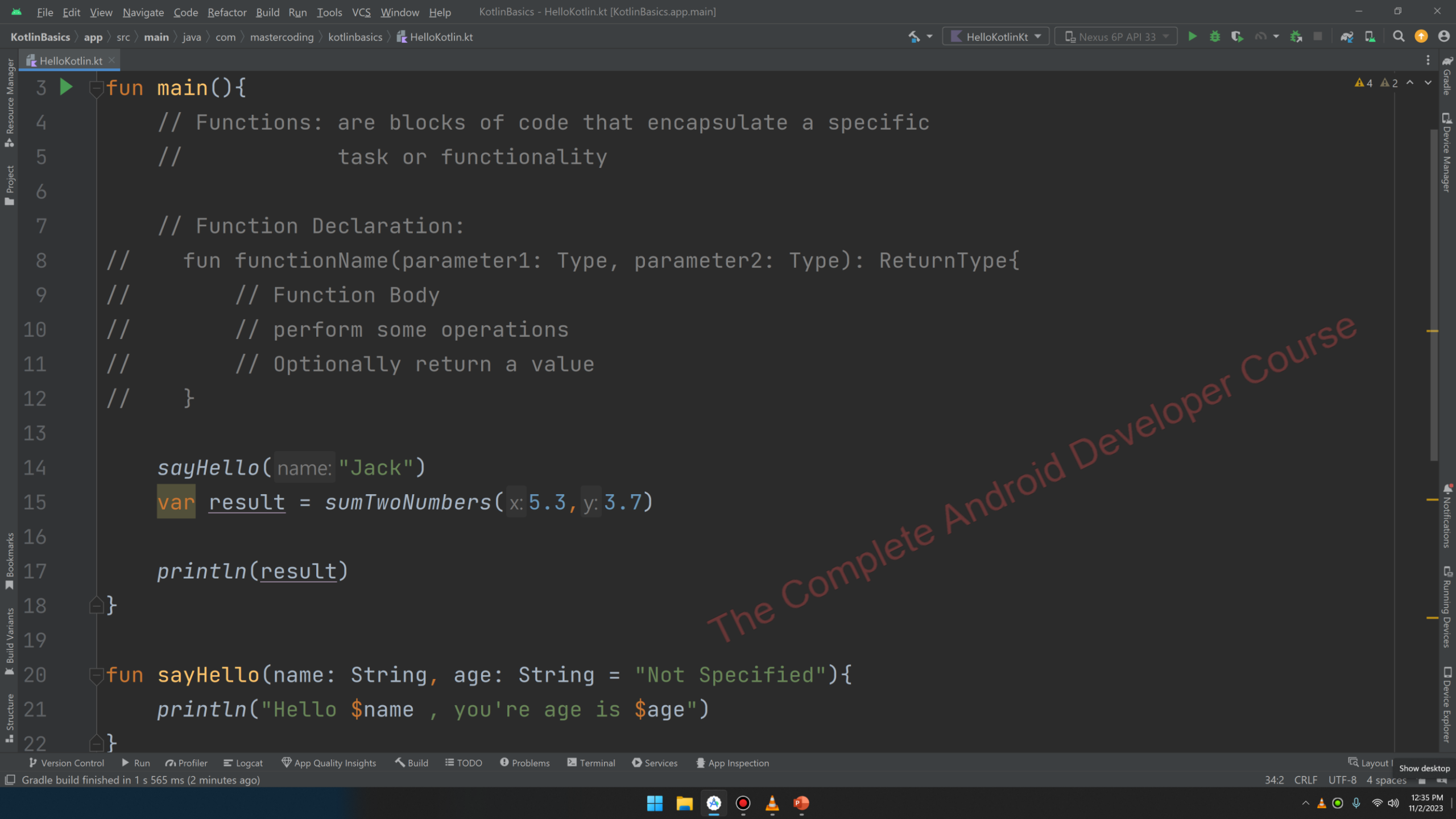


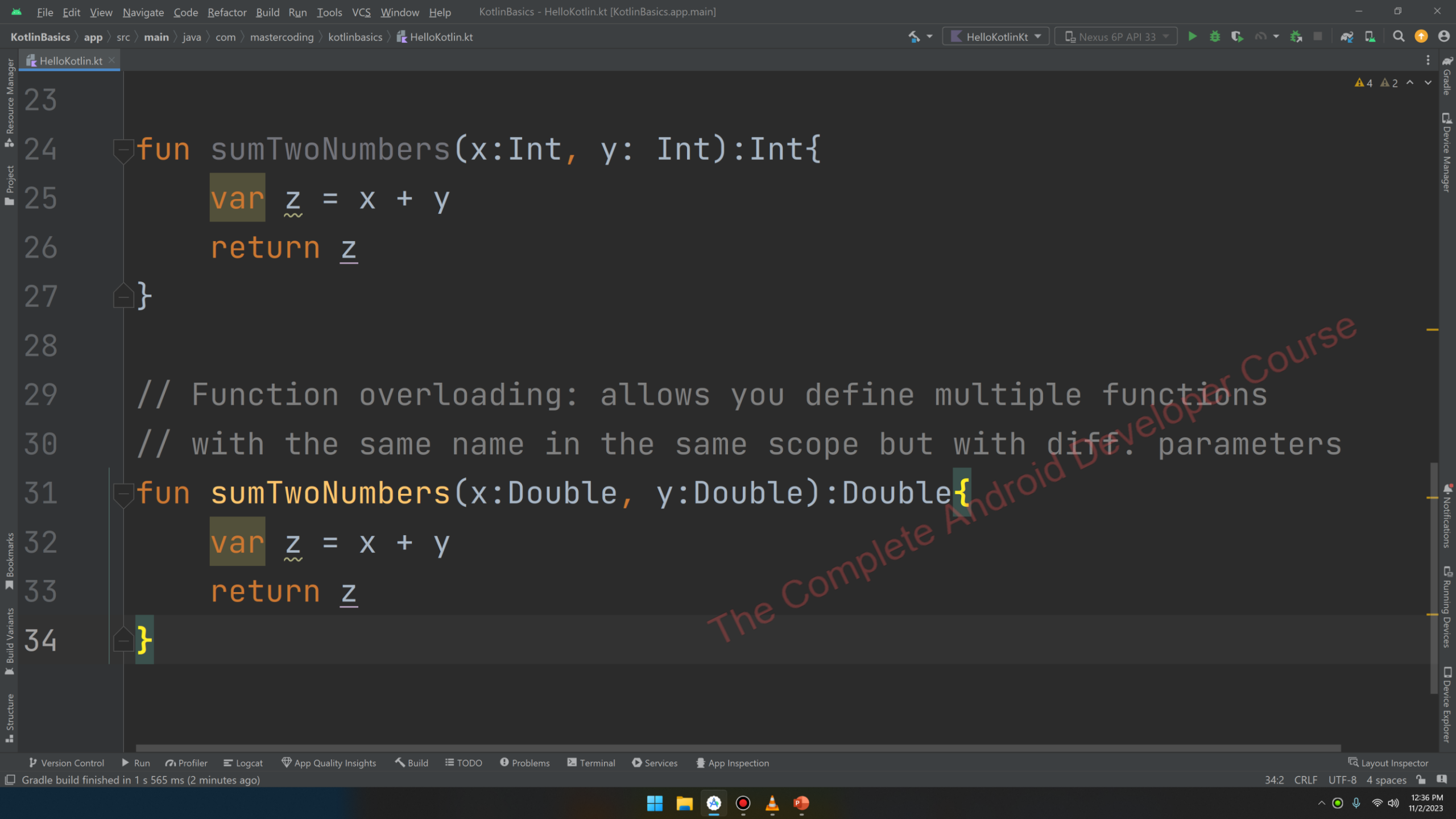






```
2  
3 fun main(){  
4     // Range: interval between 2 values  
5  
6     // Closed Range:      startValue..endValue  
7     val range = 1..5  
8  
9     // Half-Open Range:    startValue until endValue  
10    val range2 = 1 until 5  
11  
12    // Iterating through a specific range  
13    for (i in range2){  
14        println(i)  
15    }  
16
```



```
23
24 fun sumTwoNumbers(x:Int, y: Int):Int{
25     var z = x + y
26     return z
27 }
28
29 // Function overloading: allows you define multiple functions
30 // with the same name in the same scope but with diff. parameters
31 fun sumTwoNumbers(x:Double, y:Double):Double{
32     var z = x + y
33     return z
34 }
```

Object-Oriented Programming

You can create different things by combining these bricks in various ways...

In programming, OOP is a bit like playing with LEGO bricks. Instead of bricks, you have objects

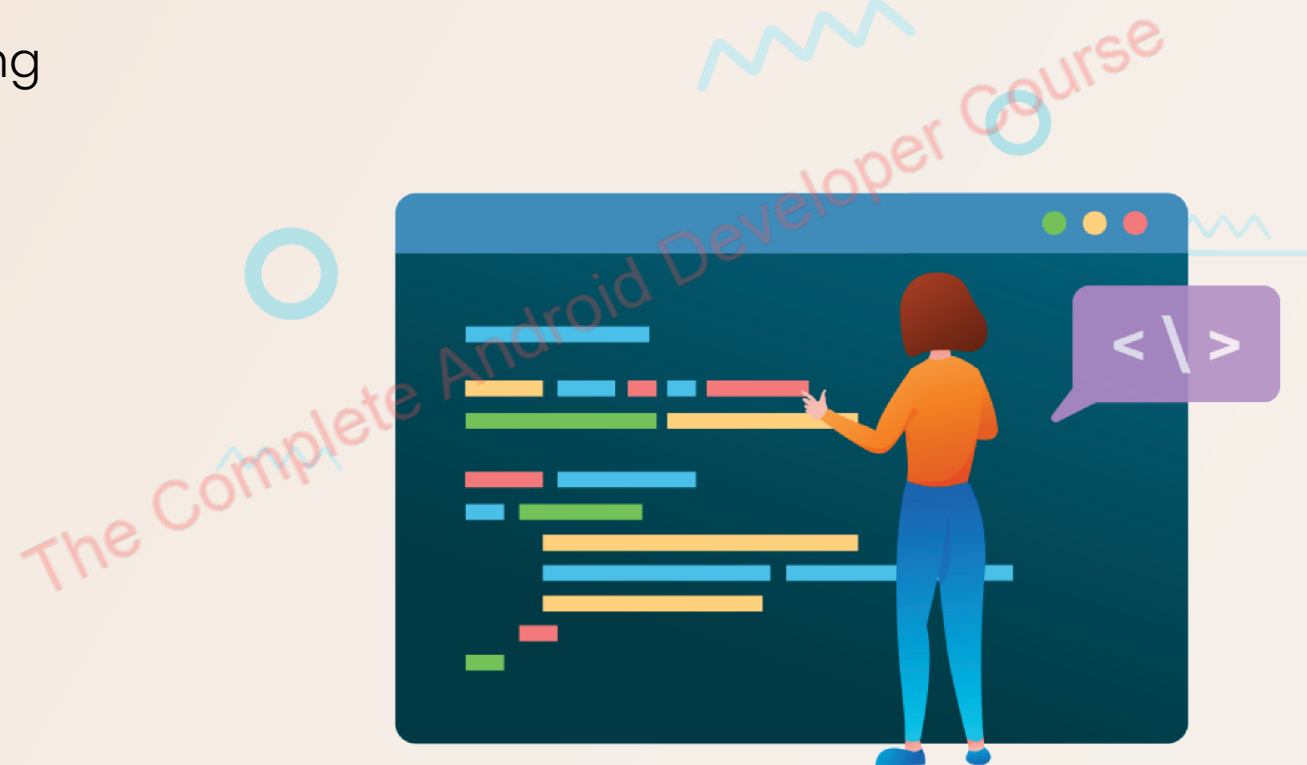


The Complete Android Developer Course

Object-Oriented Programming

OOP stands for Object-Oriented Programming.

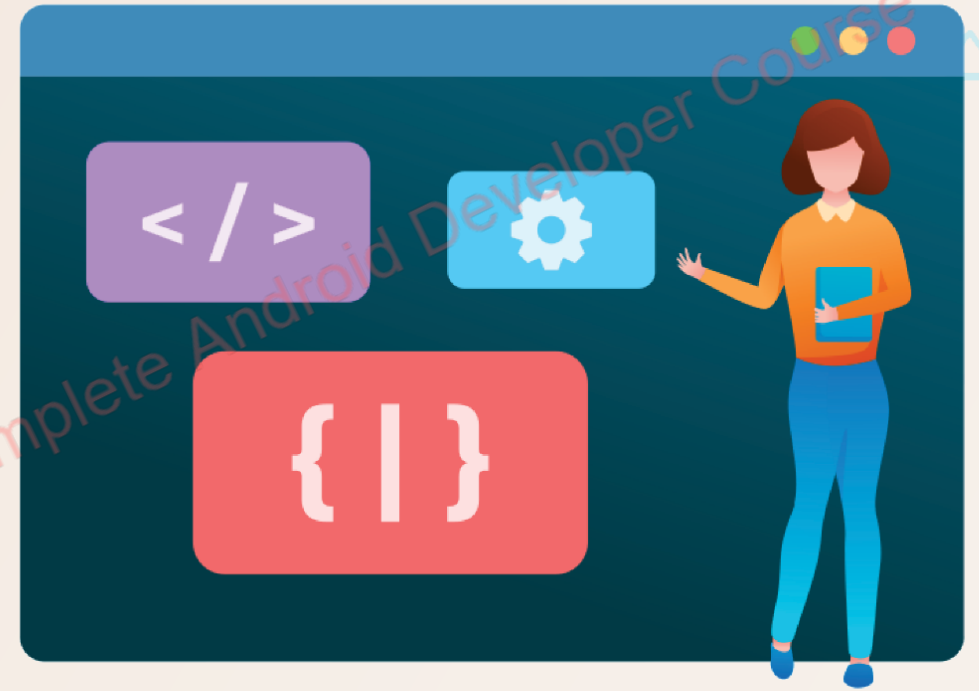
Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.



Object-Oriented Programming

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Kotlin code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time



OOP Concepts

- Classes
- Objects
- Constructors
- Class Functions
- Inheritance
- Polymorphism
- Encapsulation
- Interfaces
- Abstraction



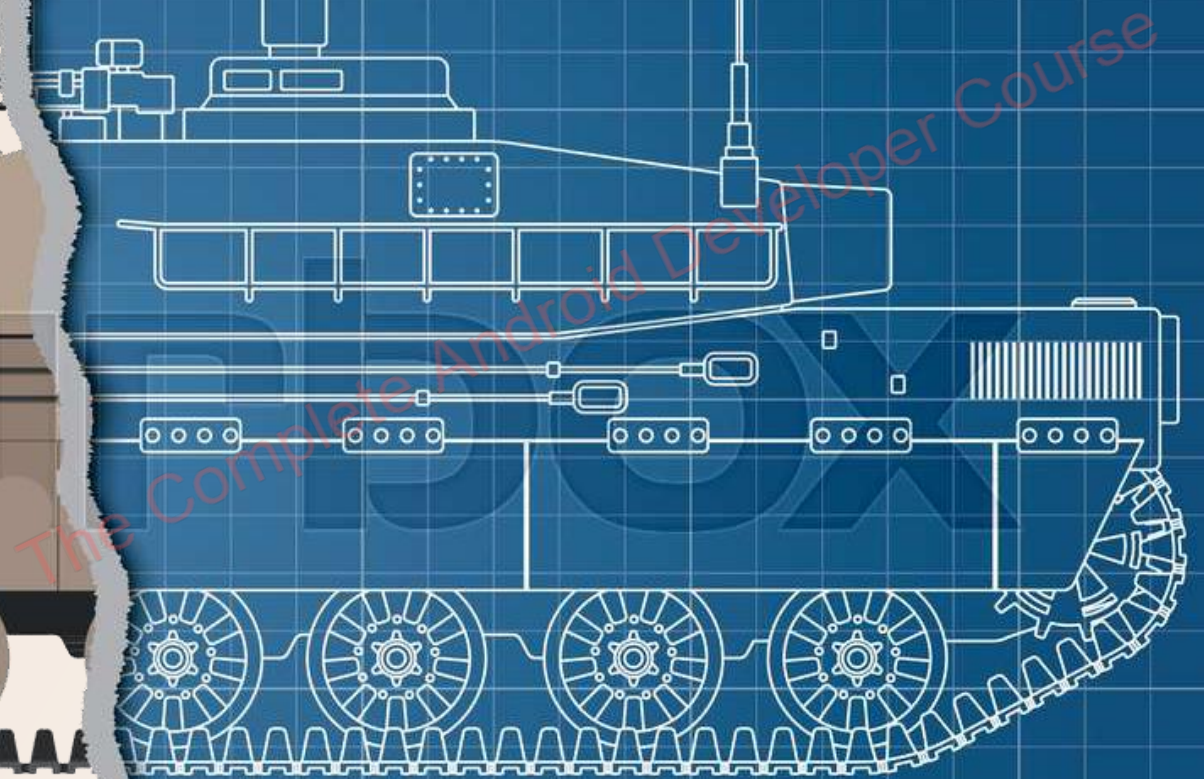
Class

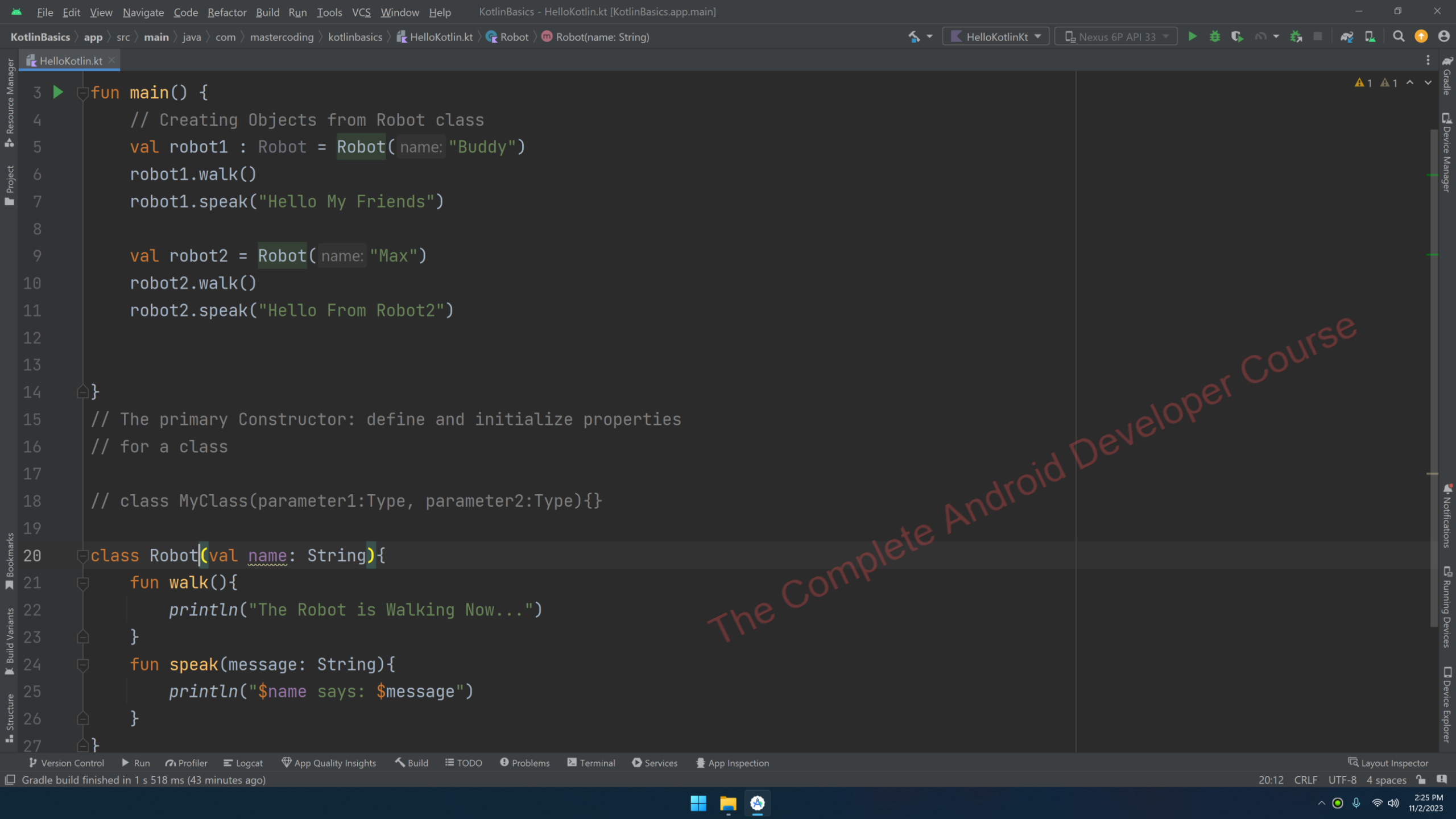
A class is a blueprint that defines the properties and behaviors of objects. It encapsulates data and functions that operate on the data.



Object

Class





KotlinBasics > app > src > main > java > com > mastercoding > kotlinbasics > HelloKotlin.kt > Robot > Robot(name: String)

HelloKotlinKt Nexus 6P API 33

HelloKotlin.kt

```
3 fun main() {
4     // Creating Objects from Robot class
5     val robot1 : Robot = Robot(name: "Buddy")
6     robot1.walk()
7     robot1.speak("Hello My Friends")
8
9     val robot2 = Robot(name: "Max")
10    robot2.walk()
11    robot2.speak("Hello From Robot2")
12
13
14 }
15 // The primary Constructor: define and initialize properties
16 // for a class
17
18 // class MyClass(parameter1:Type, parameter2:Type){}
19
20 class Robot(val name: String){
21     fun walk(){
22         println("The Robot is Walking Now...")
23     }
24     fun speak(message: String){
25         println("$name says: $message")
26     }
27 }
```

Version Control Run Profiler Logcat App Quality Insights Build TODO Problems Terminal Services App Inspection

Gradle build finished in 1 s 518 ms (43 minutes ago)

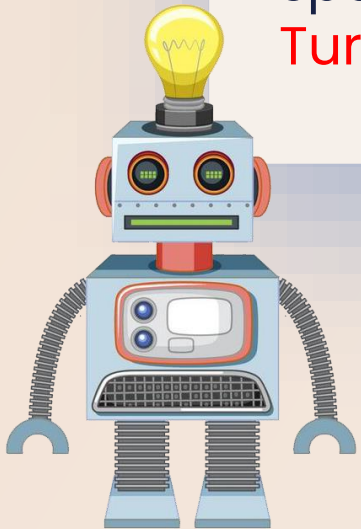
20:12 CRLF UTF-8 4 spaces Layout Inspector

Inheritance

The Problem: If you want to add a new feature “ShutDown”, you need to implement the same code for each robot. Moreover, we need to duplicate the code for walk(), speak(), and ShutDown().

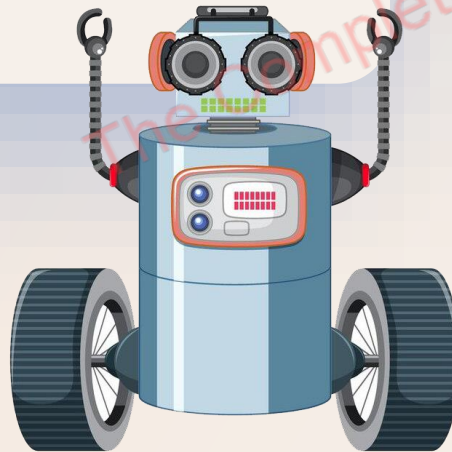
Robot

walk()
speak()
TurnOnLight()



SuperRobot

walk()
speak()
CleanHouse()

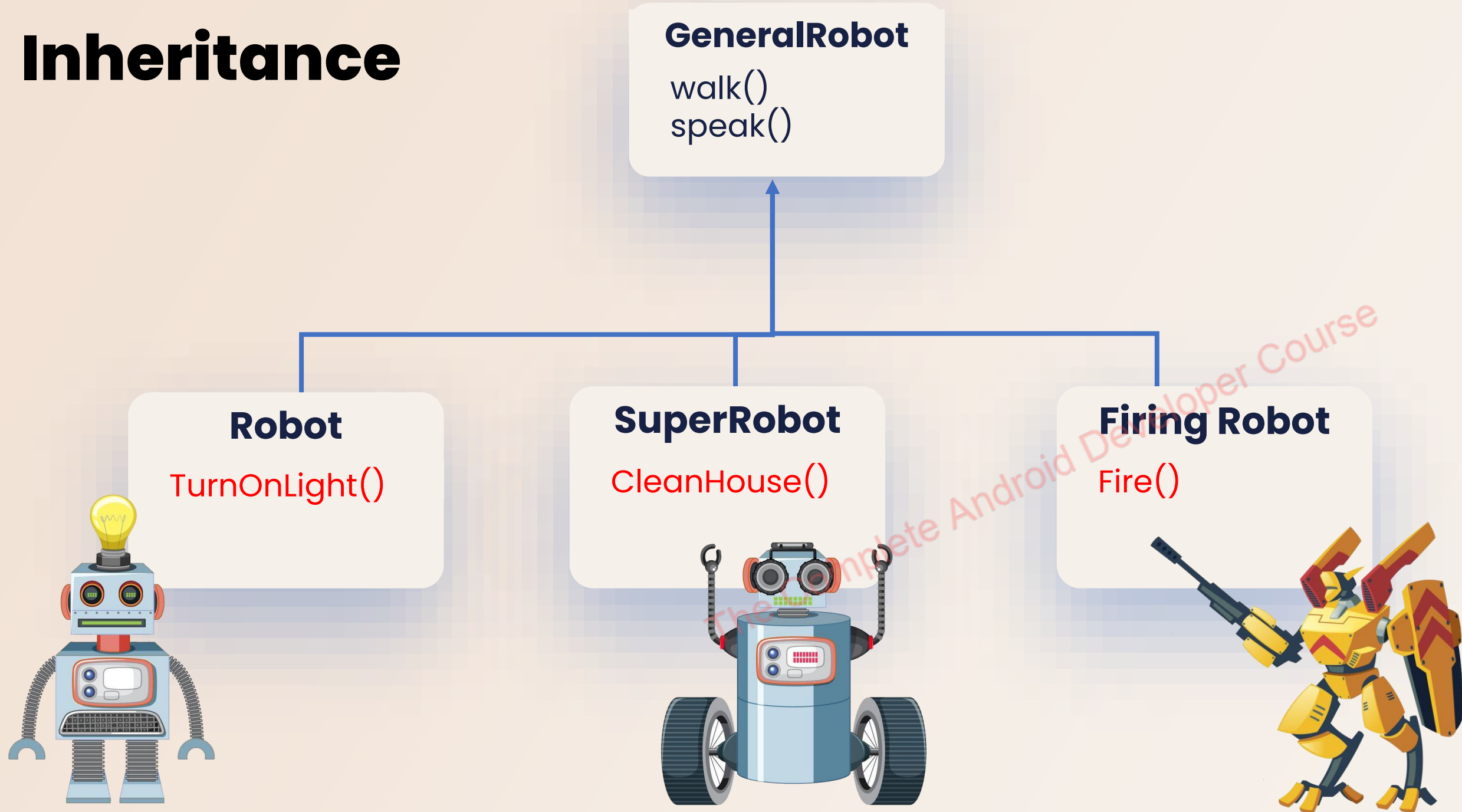


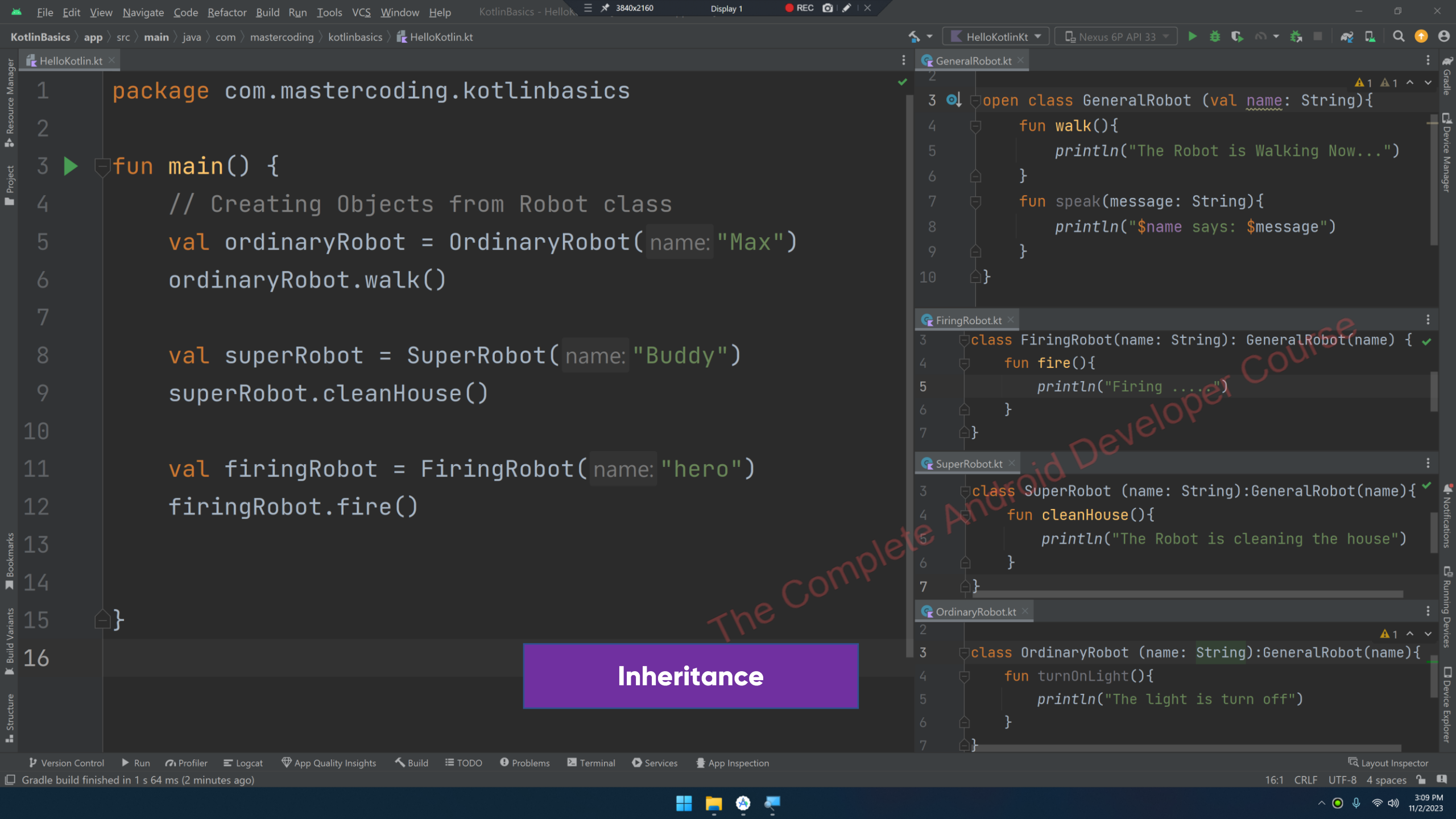
Firing Robot

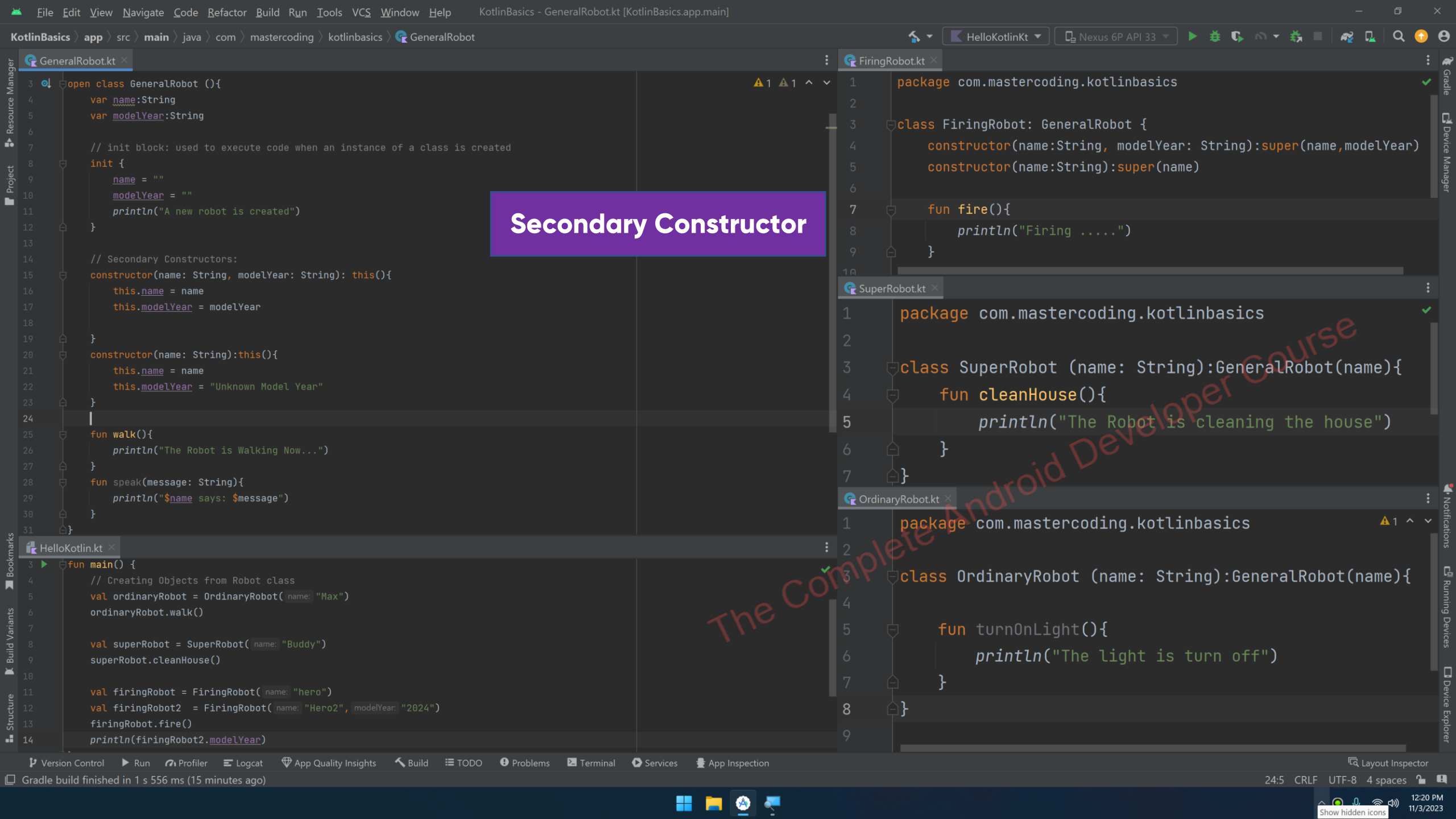
walk()
speak()
Fire()



Inheritance



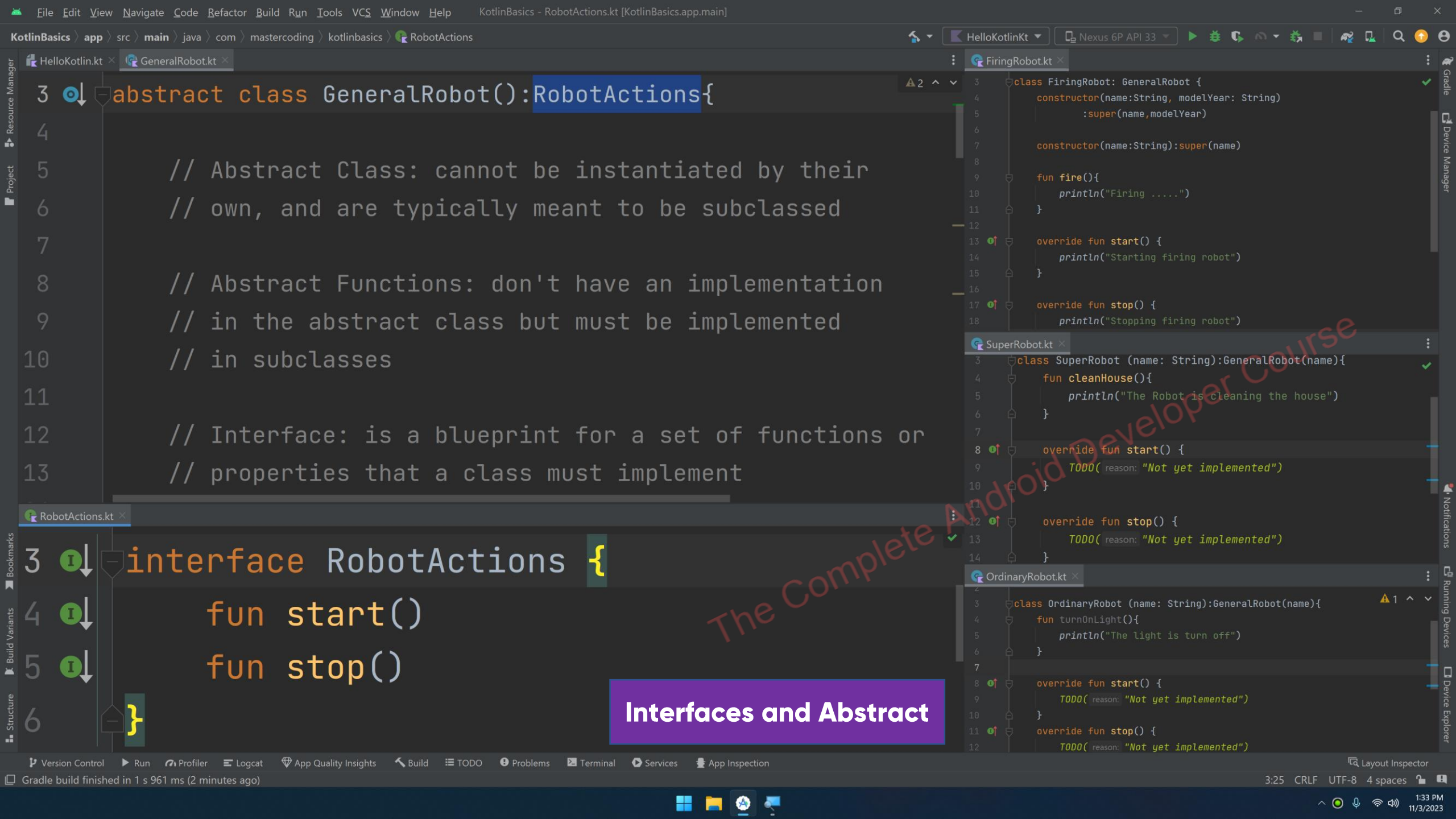




Secondary Constructor

Getters and Setters

Visibility Modifiers



```
3 abstract class GeneralRobot():RobotActions{
4
5     // Abstract Class: cannot be instantiated by their
6     // own, and are typically meant to be subclassed
7
8     // Abstract Functions: don't have an implementation
9     // in the abstract class but must be implemented
10    // in subclasses
11
12    // Interface: is a blueprint for a set of functions or
13    // properties that a class must implement
```

```
3 interface RobotActions {
4     fun start()
5     fun stop()
6 }
```

```
3 class FiringRobot: GeneralRobot {
4     constructor(name:String, modelYear: String)
5         :super(name,modelYear)
6
7     constructor(name:String):super(name)
8
9     fun fire(){
10        println("Firing .....")
11    }
12
13    override fun start() {
14        println("Starting firing robot")
15    }
16
17    override fun stop() {
18        println("Stopping firing robot")
19    }
```

```
3 class SuperRobot (name: String):GeneralRobot(name){
4     fun cleanHouse(){
5         println("The Robot is cleaning the house")
6     }
7
8     override fun start() {
9         TODO( reason: "Not yet implemented")
10    }
11
12    override fun stop() {
13        TODO( reason: "Not yet implemented")
14    }
```

```
3 class OrdinaryRobot (name: String):GeneralRobot(name){
4     fun turnOnLight(){
5         println("The light is turn off")
6     }
7
8     override fun start() {
9         TODO( reason: "Not yet implemented")
10    }
11    override fun stop() {
12        TODO( reason: "Not yet implemented")
13    }
```

Interfaces and Abstract

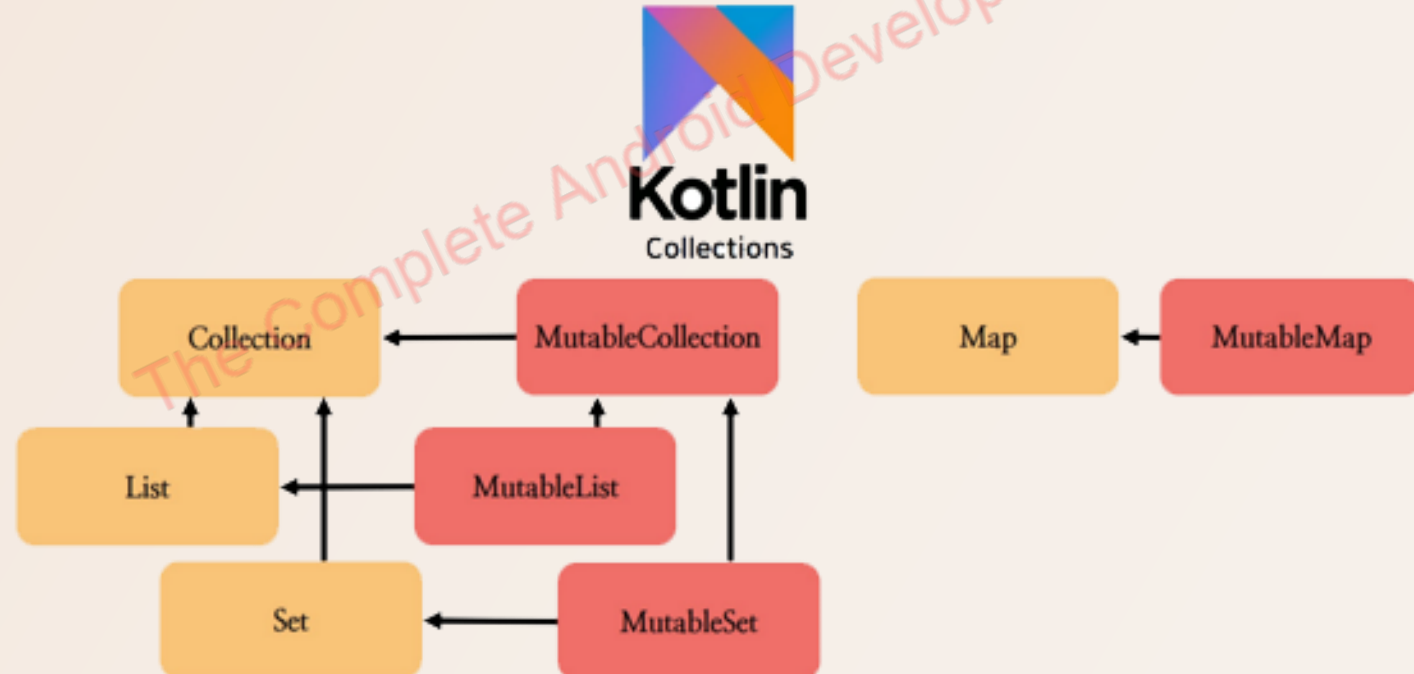
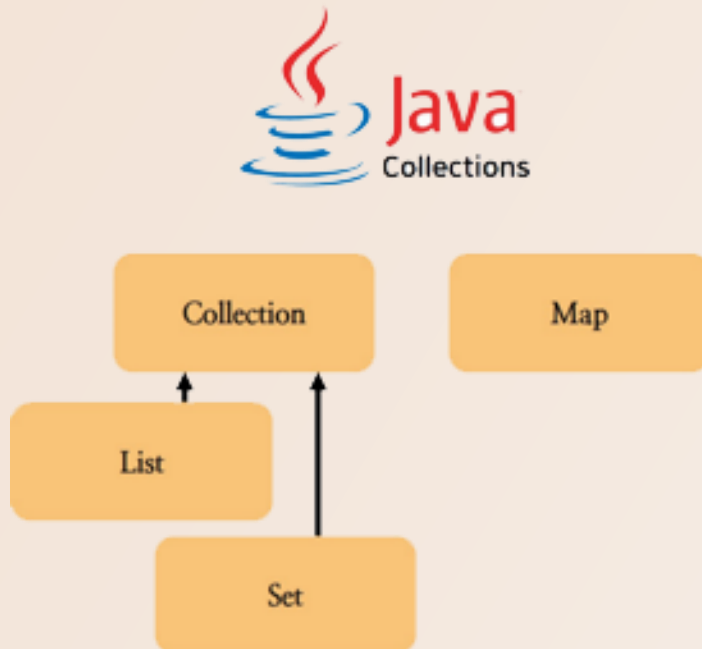
Kotlin Collections

A collection usually contains a number of objects of the same type and these objects in the collection are called elements or items.

Kotlin Standard Library provides a rich set of tools for managing collections.

Types of Collections

- Immutable Collection
- Mutable Collection

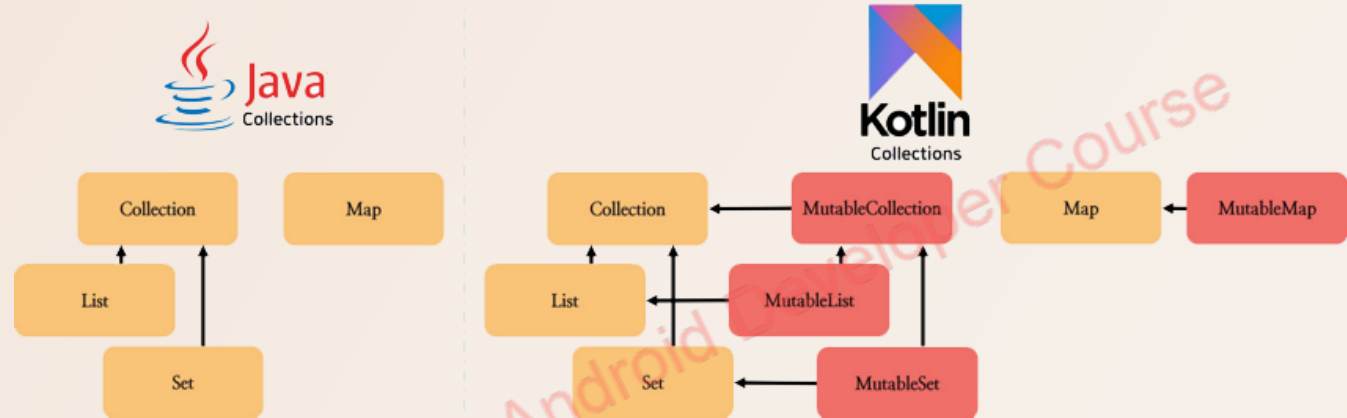


Mutable vs Immutable

Immutable Collection

It means that it supports only read-only functionalities and can not be modified its elements. Immutable Collections and their corresponding methods are:

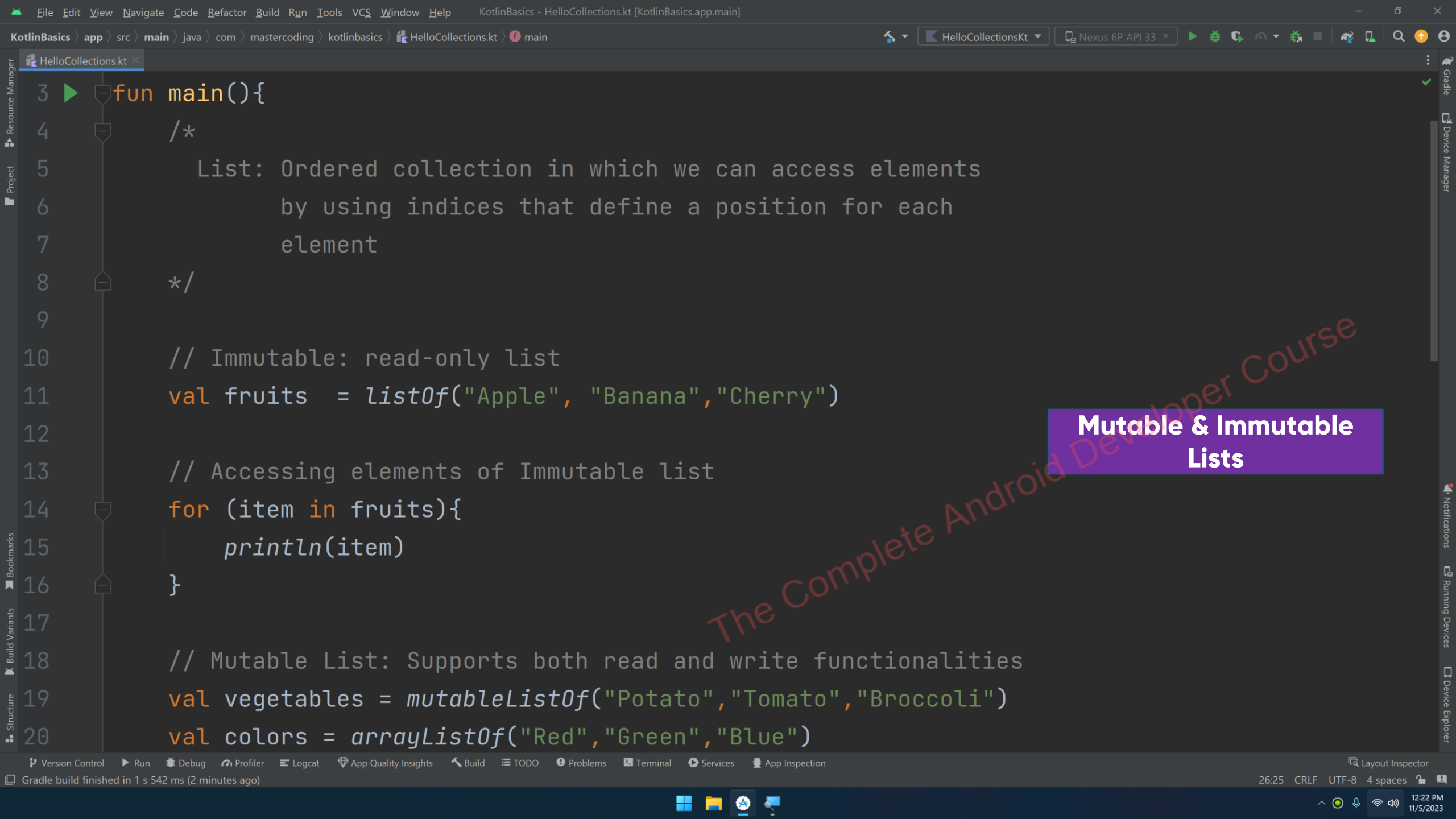
- List – `listOf()` and `listOf<T>()`
- Set – `setOf()`
- Map – `mapOf()`



Mutable Collection

It supports both read and write functionalities. Mutable collections and their corresponding methods are:

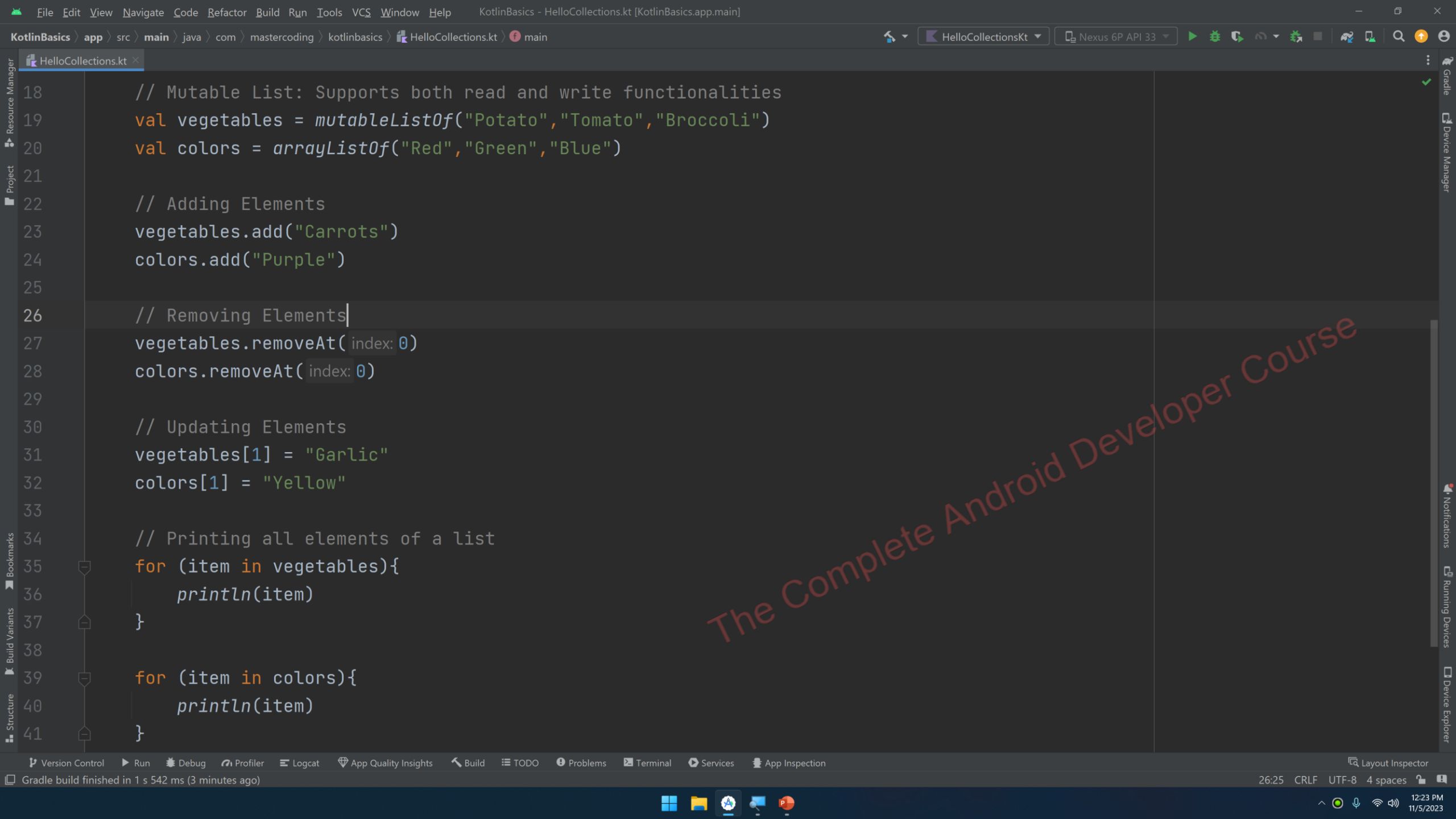
- List – `mutableListOf()`, `arrayListOf()` and `ArrayList`
- Set – `mutableSetOf()`, `hashSetOf()`
- Map – `mutableMapOf()`, `hashMapOf()` and `HashMap`

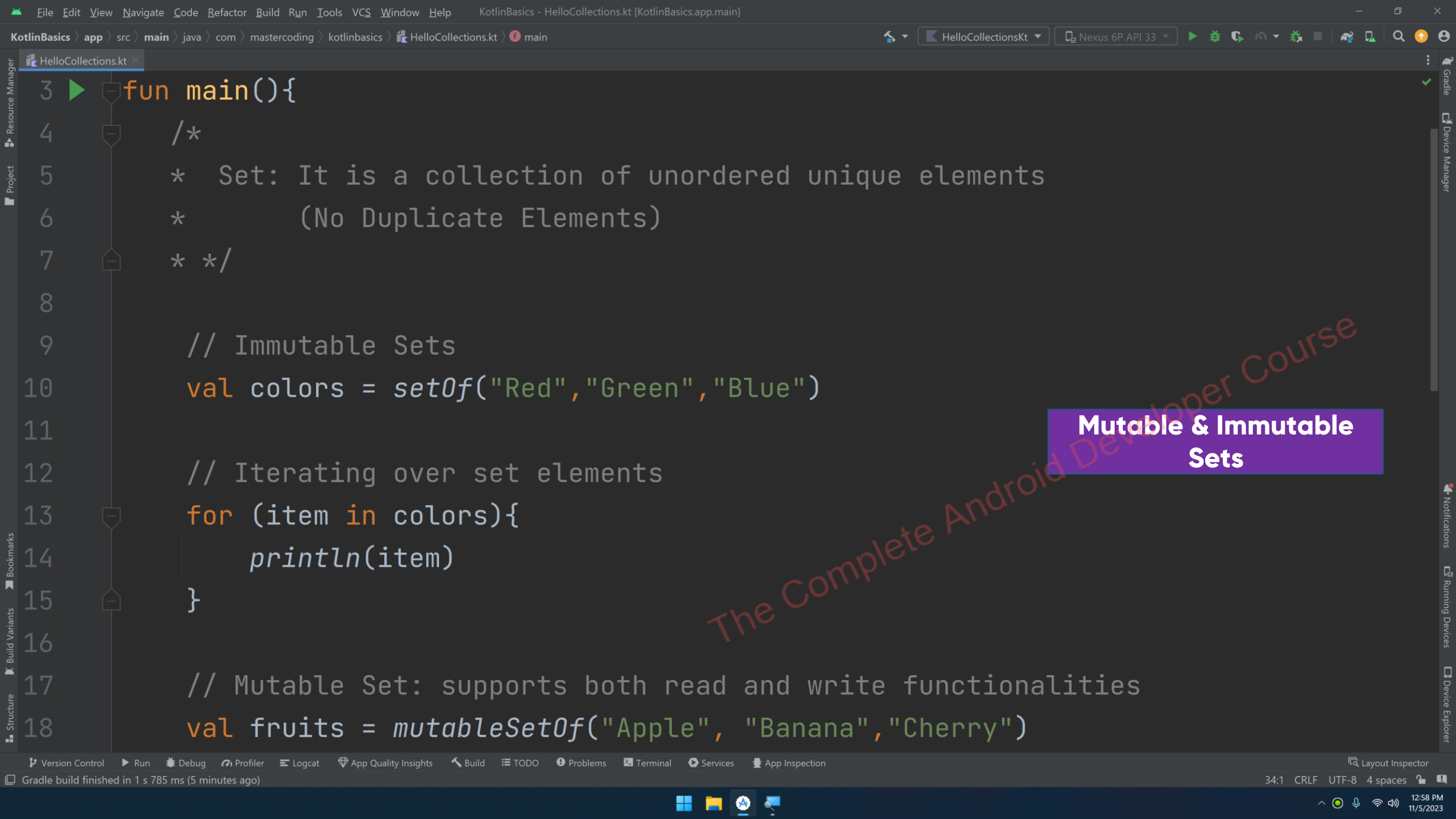


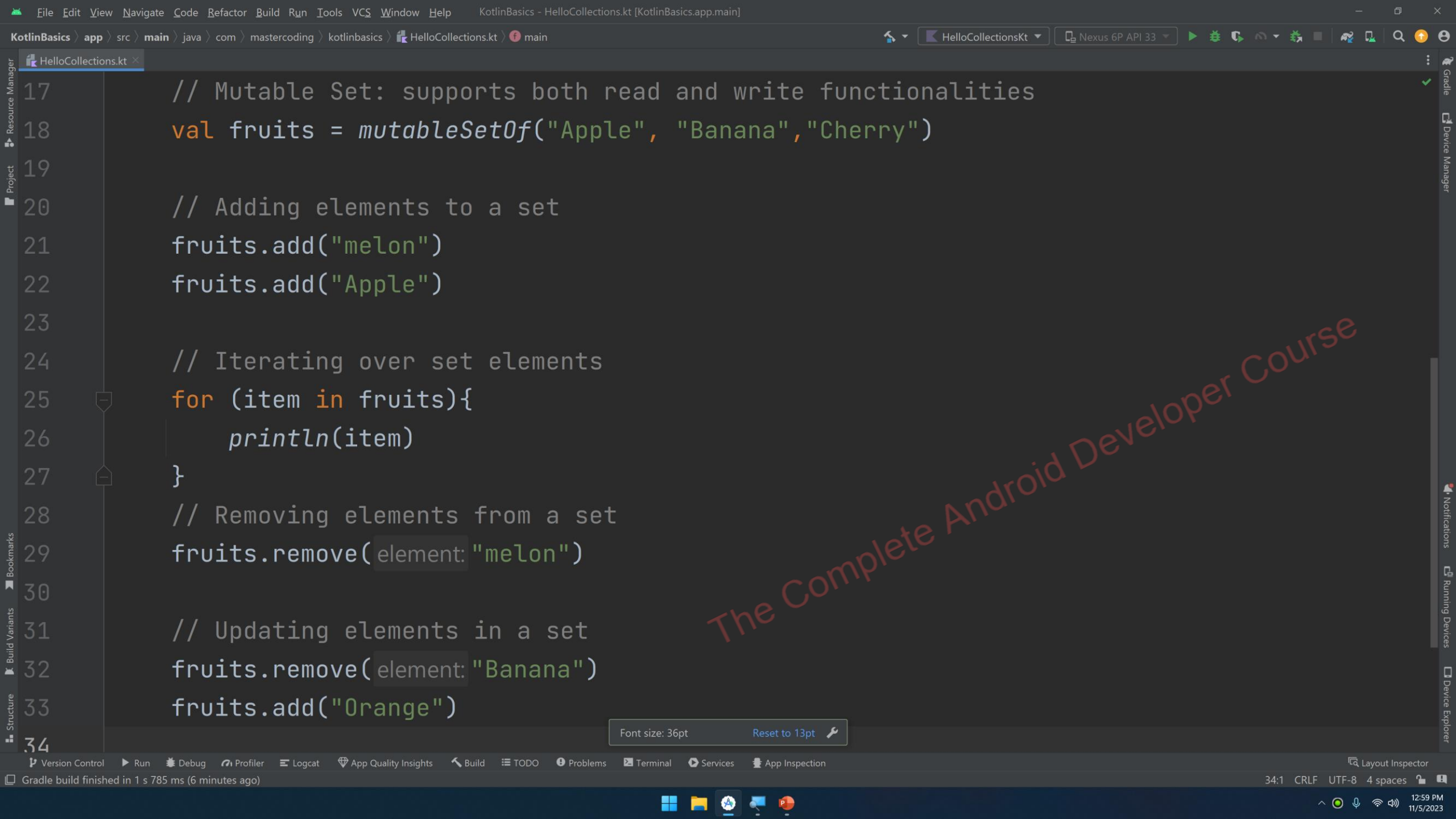
```
3 fun main(){
4     /*
5         List: Ordered collection in which we can access elements
6             by using indices that define a position for each
7             element
8     */
9
10    // Immutable: read-only list
11    val fruits = listOf("Apple", "Banana", "Cherry")
12
13    // Accessing elements of Immutable list
14    for (item in fruits){
15        println(item)
16    }
17
18    // Mutable List: Supports both read and write functionalities
19    val vegetables = mutableListOf("Potato", "Tomato", "Broccoli")
20    val colors = arrayListOf("Red", "Green", "Blue")
```

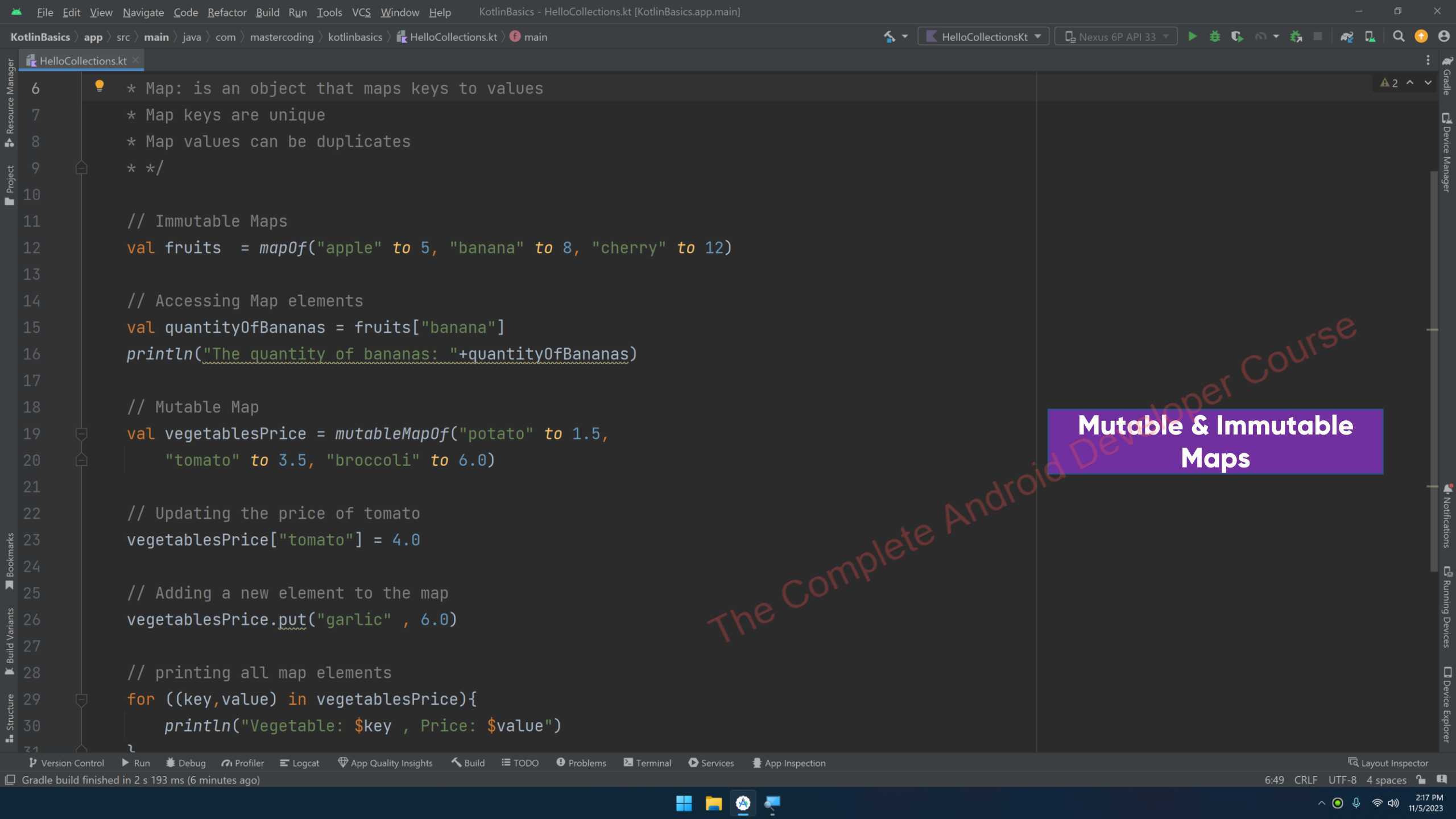
Mutable & Immutable Lists

The Complete Android Developer Course









```
6  * Map: is an object that maps keys to values
7  * Map keys are unique
8  * Map values can be duplicates
9  * */
10
11 // Immutable Maps
12 val fruits = mapOf("apple" to 5, "banana" to 8, "cherry" to 12)
13
14 // Accessing Map elements
15 val quantityOfBananas = fruits["banana"]
16 println("The quantity of bananas: "+quantityOfBananas)
17
18 // Mutable Map
19 val vegetablesPrice = mutableMapOf("potato" to 1.5,
20     "tomato" to 3.5, "broccoli" to 6.0)
21
22 // Updating the price of tomato
23 vegetablesPrice["tomato"] = 4.0
24
25 // Adding a new element to the map
26 vegetablesPrice.put("garlic" , 6.0)
27
28 // printing all map elements
29 for ((key,value) in vegetablesPrice){
30     println("Vegetable: $key , Price: $value")
31 }
```

Mutable & Immutable Maps

The Complete Android Developer Course

Kotlin Literals

In programming, a literal is a constant value assigned to a variable.

```
val x = 10
```

```
var double1 = 13.8
```

```
val str = "Welcome to Our Course"
```

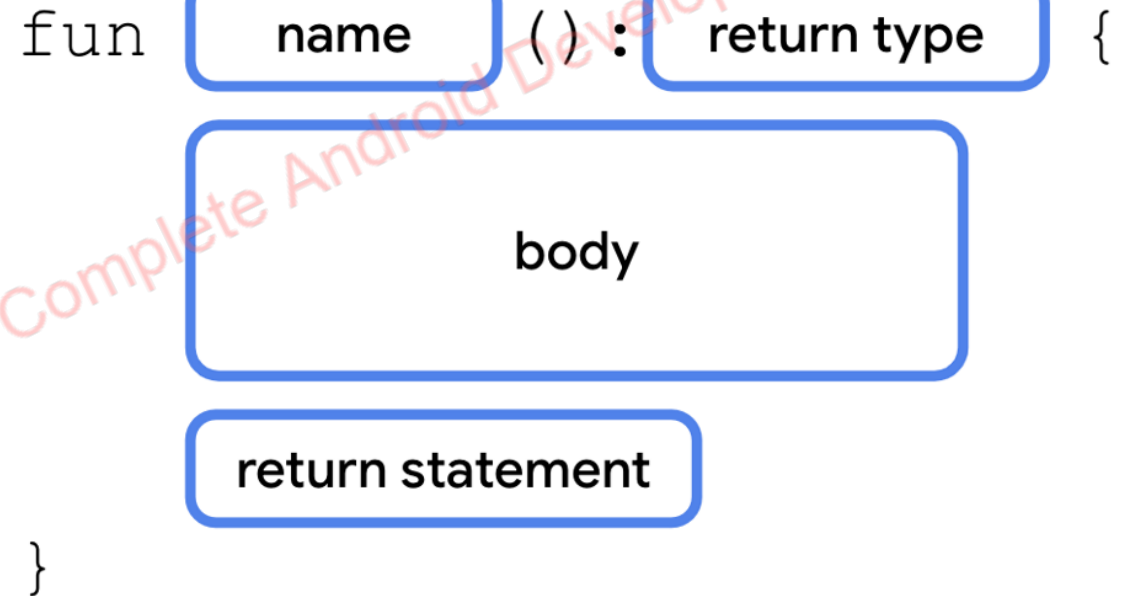
The Complete Android Developer Course

Function Literals

When we assign a function to a variable, it becomes a function literal.

Kotlin provides 2 types of function literals. They are:

- Lambda Expression
- Anonymous Function

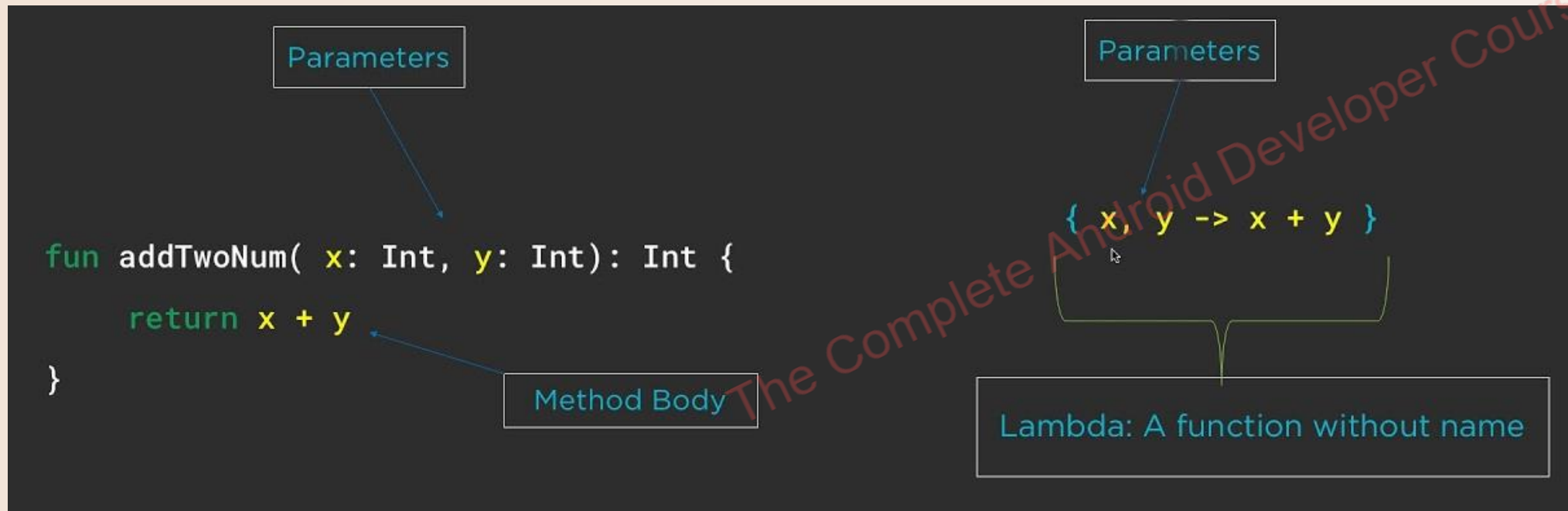


The diagram illustrates the syntax of a Kotlin function. It shows the following components in order: the keyword `fun`, a box labeled `name`, a pair of parentheses `()`, a colon `:`, a box labeled `return type`, an opening curly brace `{`, a large box labeled `body`, a box labeled `return statement`, and a closing curly brace `}`. A diagonal watermark text "The Complete Android Developer Course" is visible across the diagram.

```
fun name () : return type {  
    body  
    return statement  
}
```

Lambda Expression

Lambda expression is a short way to create a function. To define a lambda expression, enclose the method body inside braces.



Lambda Expression

There are two ways to execute the lambda:

- Add parentheses after the variable name
- Call `invoke()` method

```
val printWelcome = { println("Welcome to our Course") }  
printWelcome()  
printWelcome.invoke()
```

The Complete Android Developer Course

Syntax of a Lambda Expression

```
val lambda: (DataType1, DataType2) → ReturnType = { variable1: DataType1, variable2:DataType2 → methodBody }
```

**Variable
Name**

**Function
Type**

Lambda

```
val add : (Int, Int) -> Int = { a: Int, b: Int -> a+ b }
```

Shorter Syntax

```
val lambda: (DataType1, DataType2) → ReturnType = { variable1: DataType1, variable2:DataType2 → methodBody }
```

**Variable
Name**

**Function
Type**

Lambda

```
val add : (Int, Int) -> Int = {a: Int, b: Int -> a+ b}
```

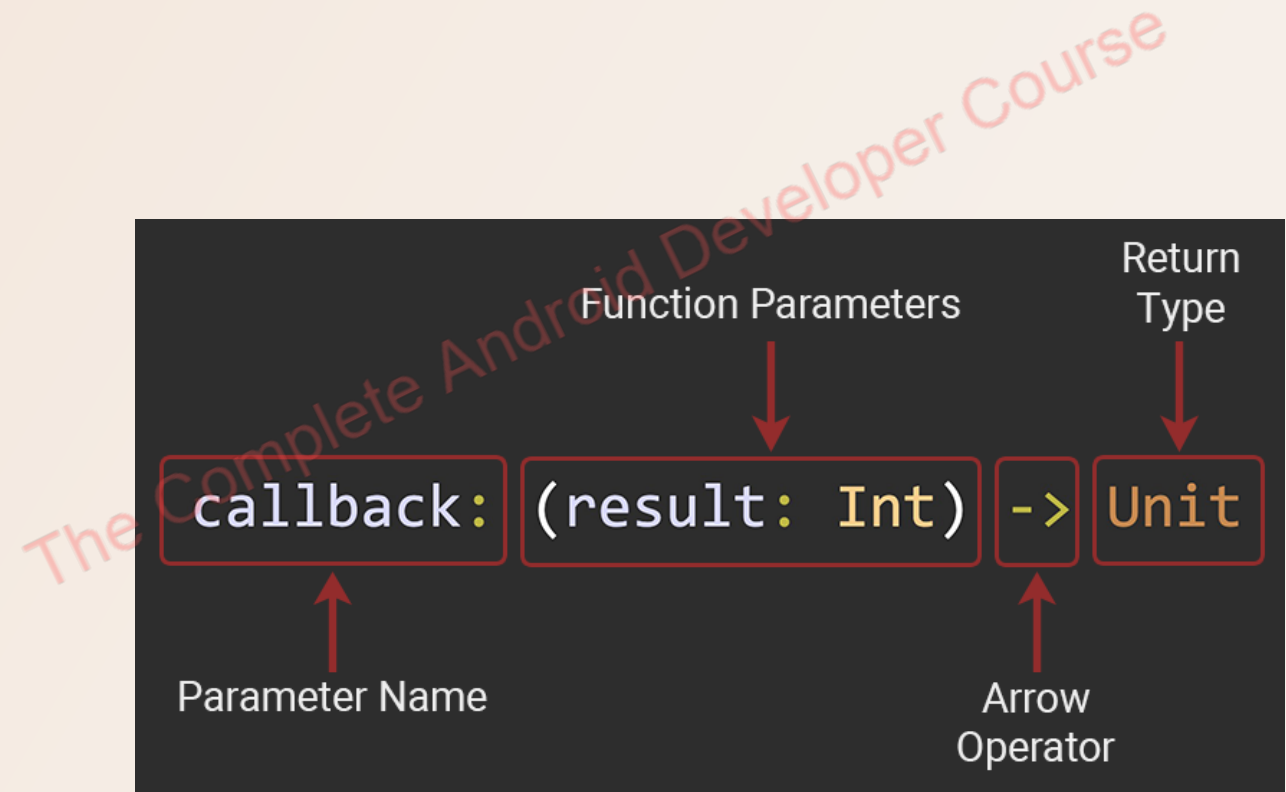
```
val add = {a: Int, b: Int -> a+ b}
```

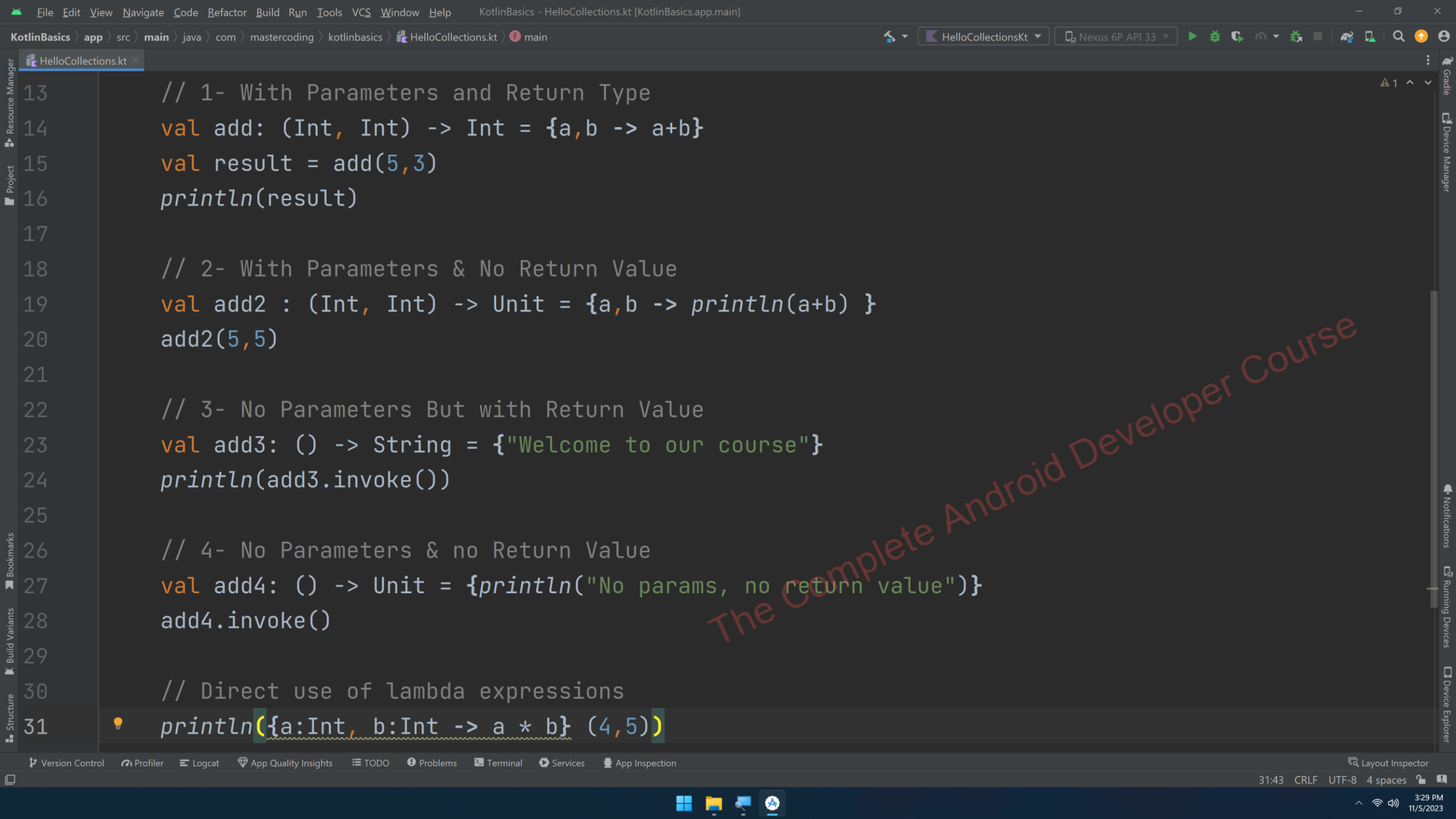
```
val add: (Int, Int) -> Int = {a, b -> a+b }
```

Lambda Expression Types

There are 4 types depending on the parameters and return type:

- With Parameters and No Return Value
- With Parameters and Return Value
- No Parameters and No Return Value
- No Parameters and Return Value





Anonymous Function

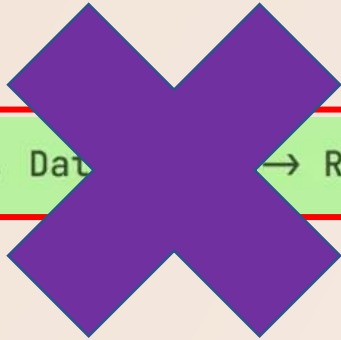
It is a function without a name.

```
val variableName: (DataType1, DataType2) → ReturnType = fun(parameter1, parameter2): ReturnType { methodBody }
```

```
val add: (Int, Int) -> Int = fun(a, b): Int { return a + b }
```

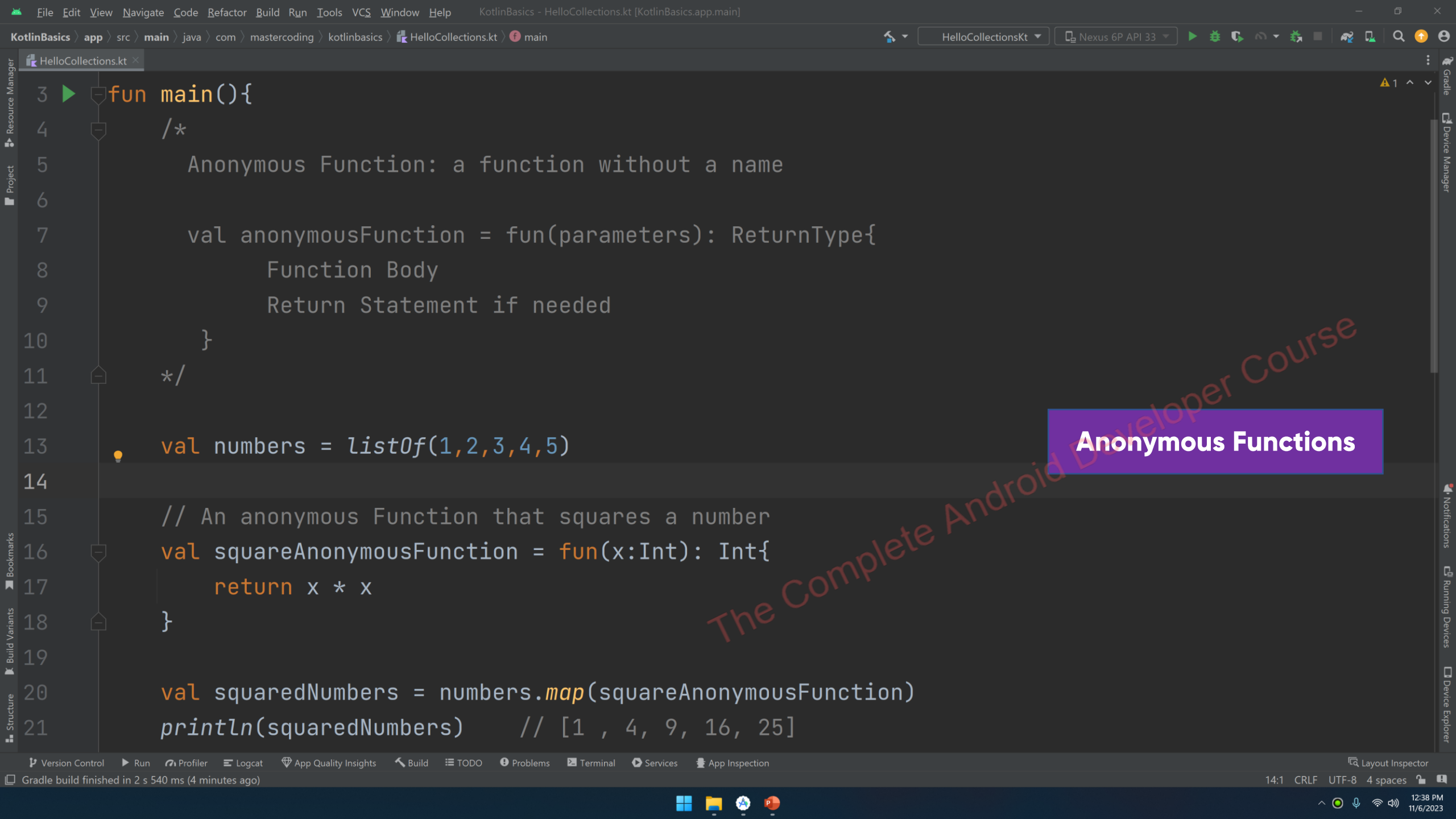
Shorter Syntax Anonymous Function

It is a function without a name.

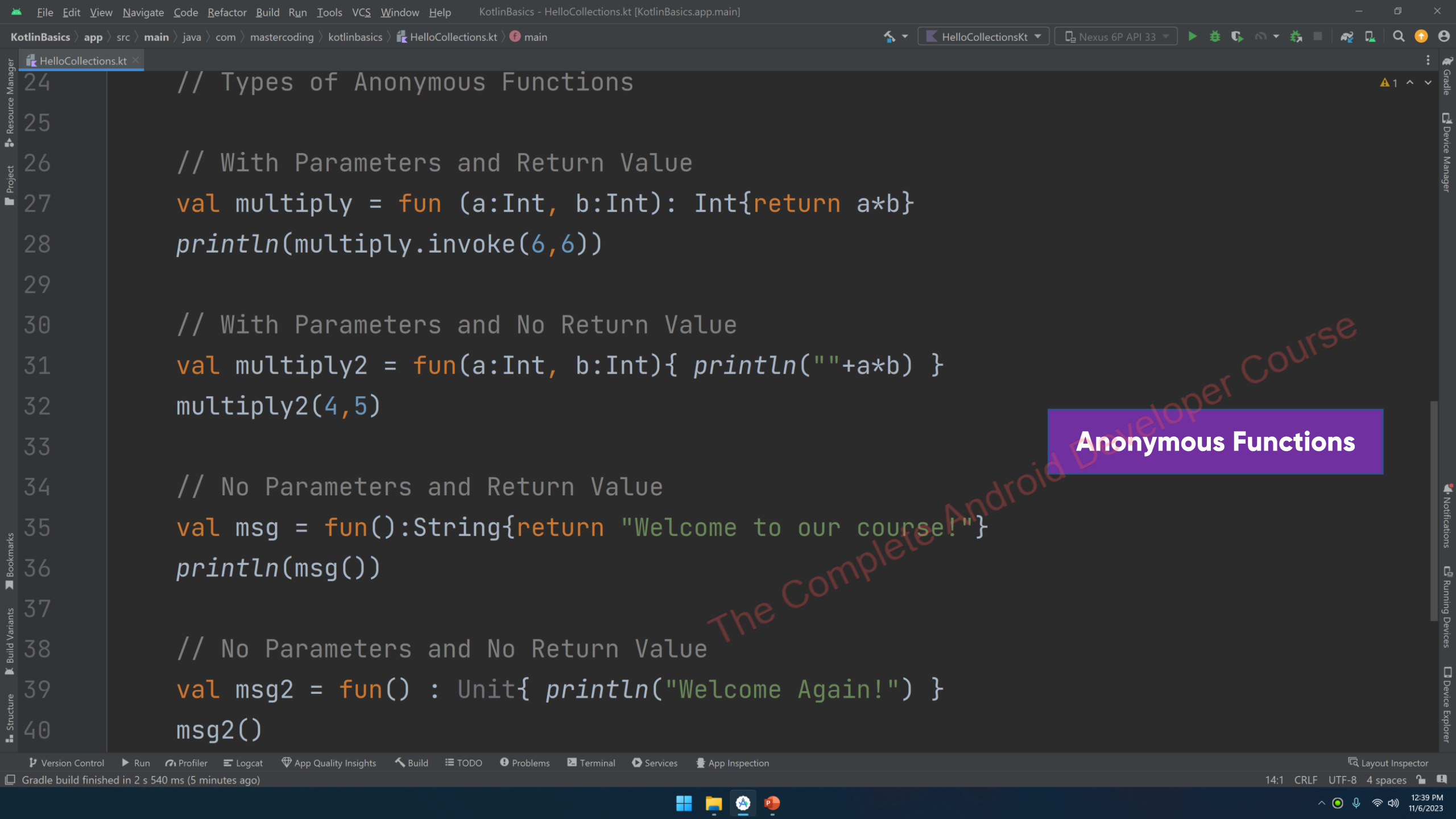


```
val variableName: (DataType1, DataType2) → ReturnType = fun(parameter1, parameter2): ReturnType { methodBody }
```

```
val variableName = fun(DataType1, DataType2): ReturnType { methodBody }
```



Anonymous Functions

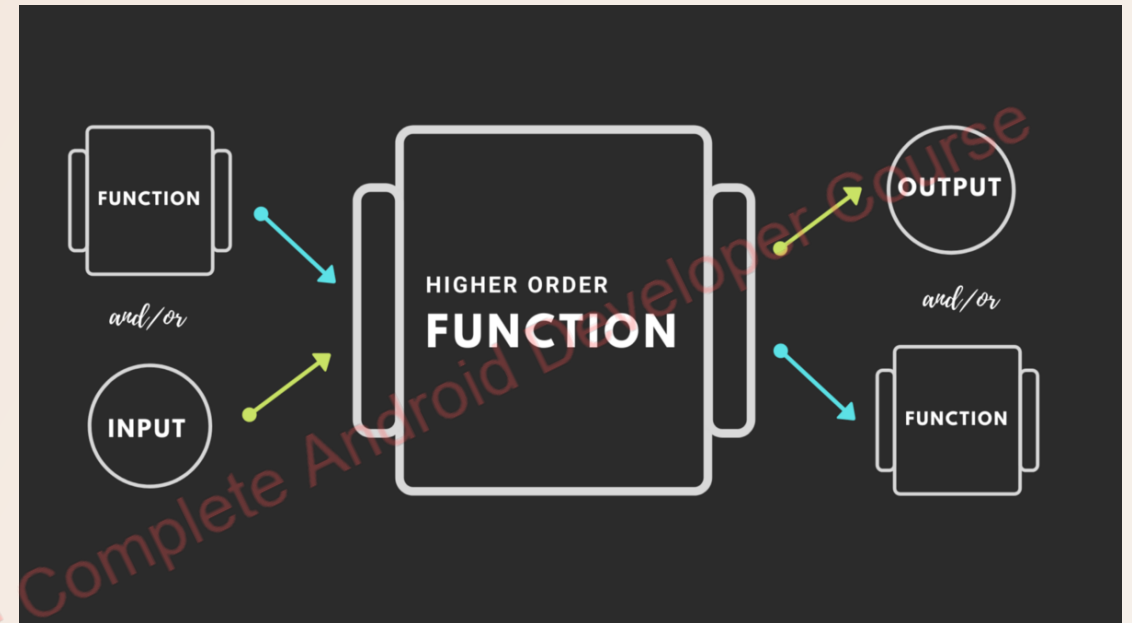


Anonymous Functions

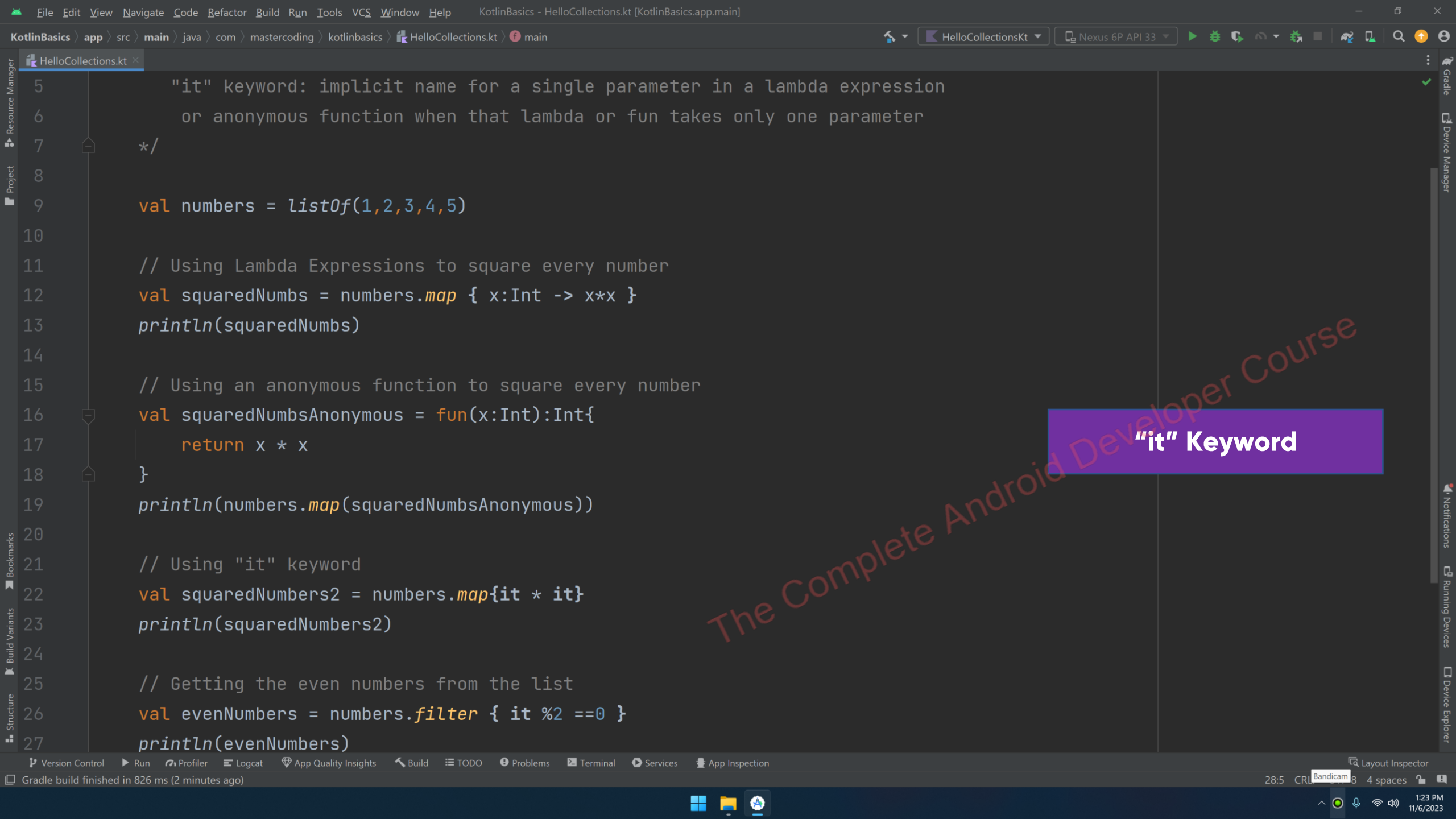
Higher Order Function

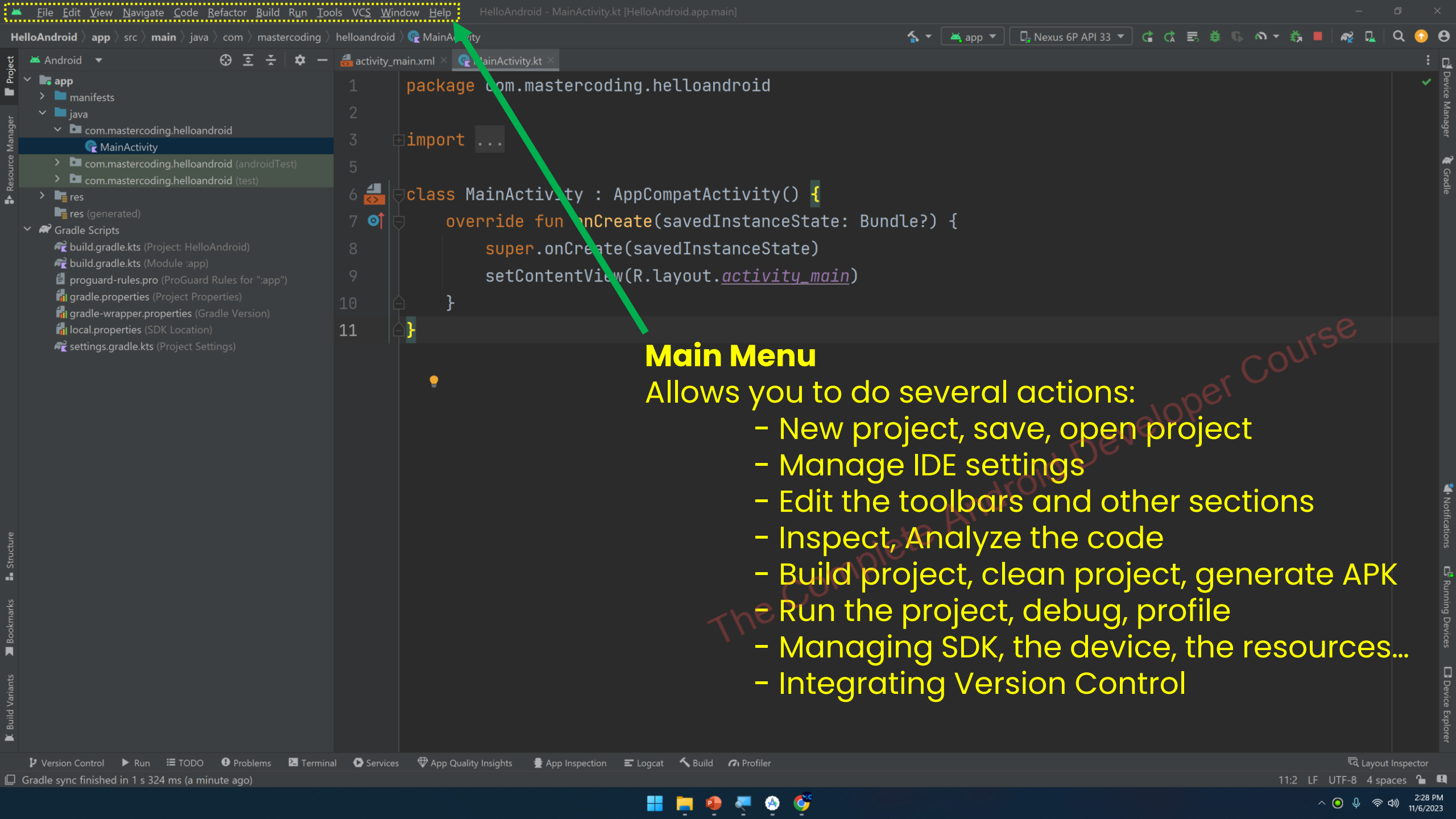
It is a function that takes a function as a parameter or returns a function or both.

In general, lambda expressions are passed as an argument to a higher-order function or returned from it. We can also use anonymous functions for the same.



Higher Order Functions

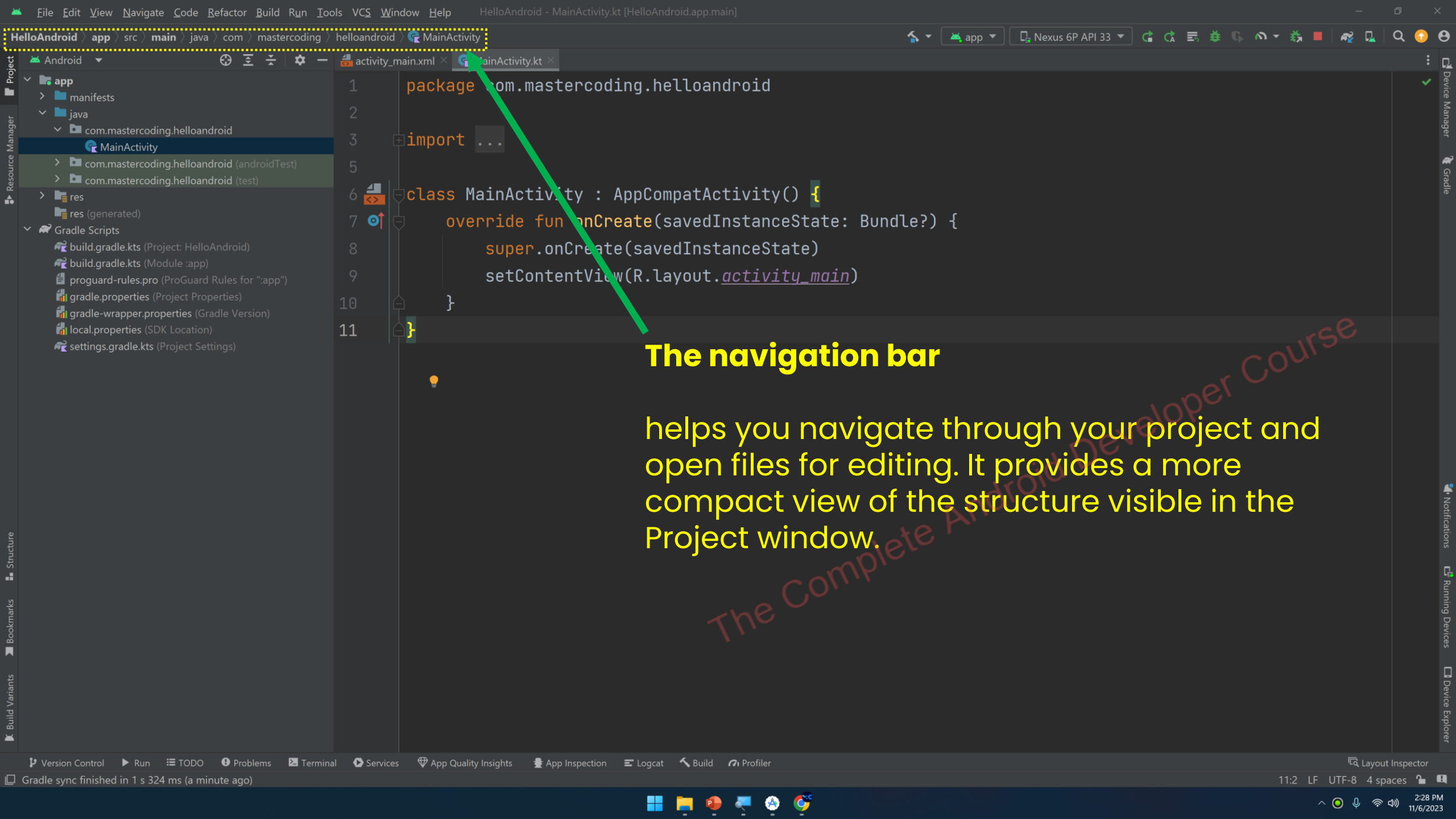




Main Menu

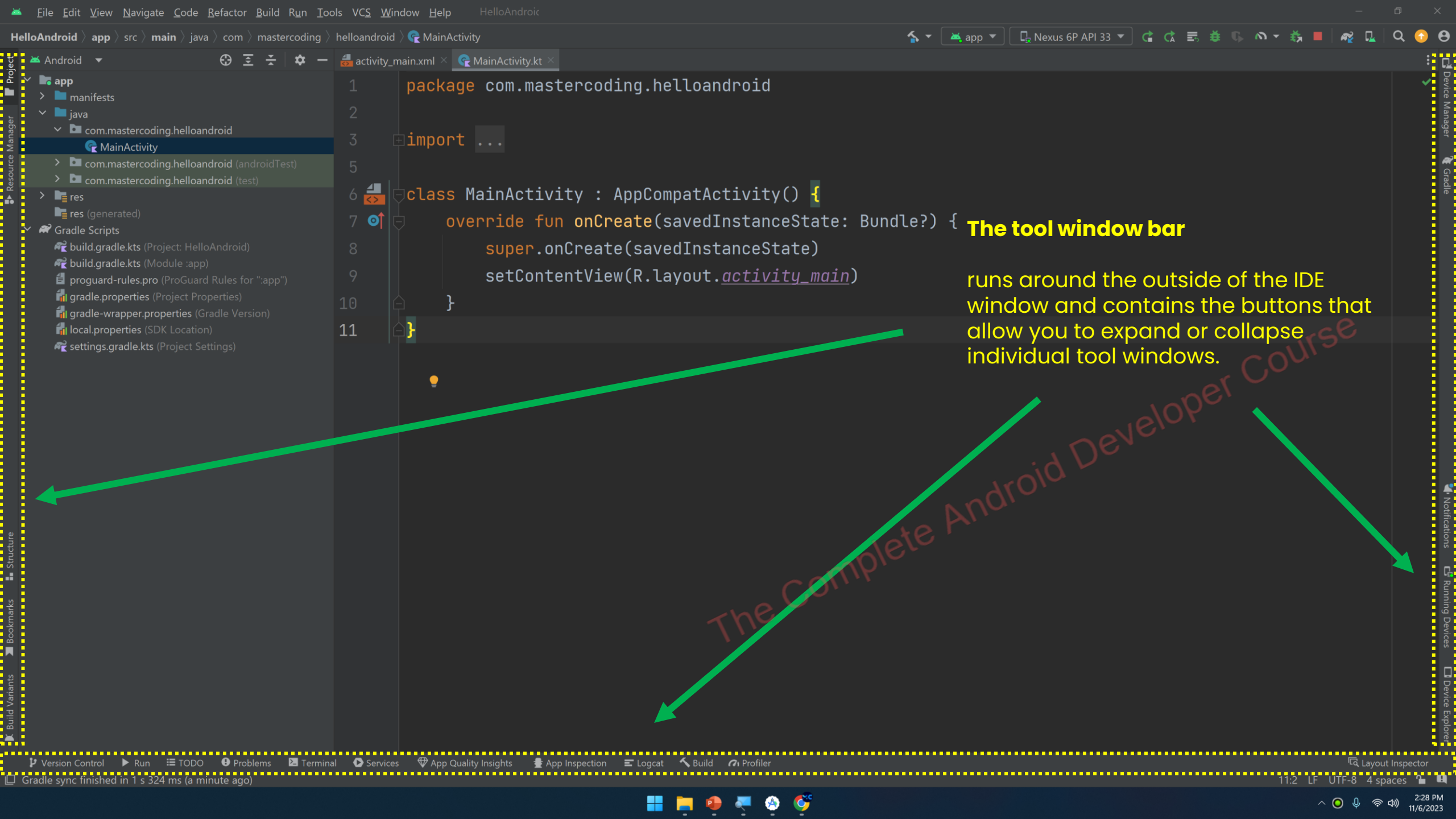
Allows you to do several actions:

- New project, save, open project
- Manage IDE settings
- Edit the toolbars and other sections
- Inspect, Analyze the code
- Build project, clean project, generate APK
- Run the project, debug, profile
- Managing SDK, the device, the resources...
- Integrating Version Control



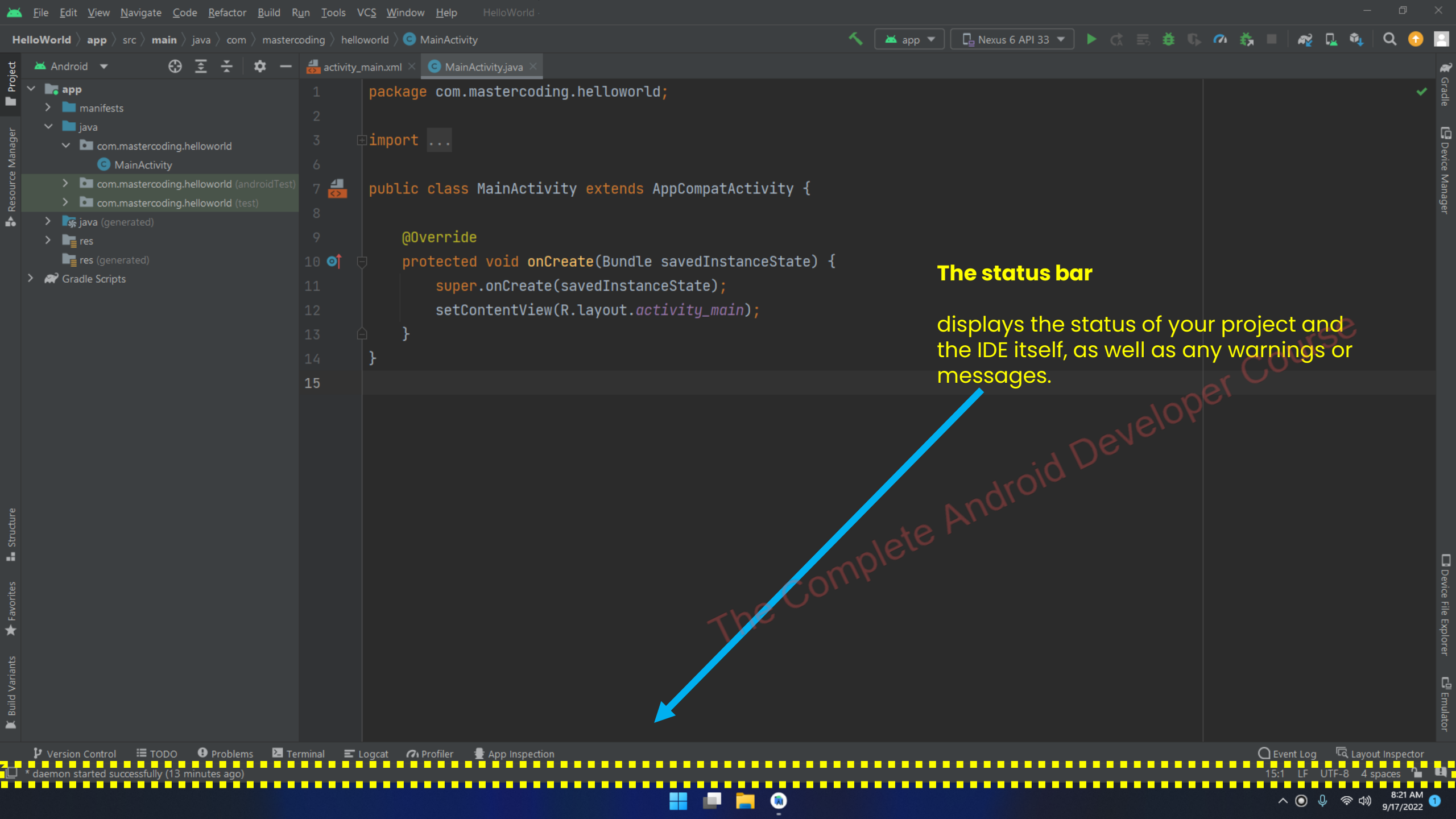
The navigation bar

helps you navigate through your project and open files for editing. It provides a more compact view of the structure visible in the Project window.



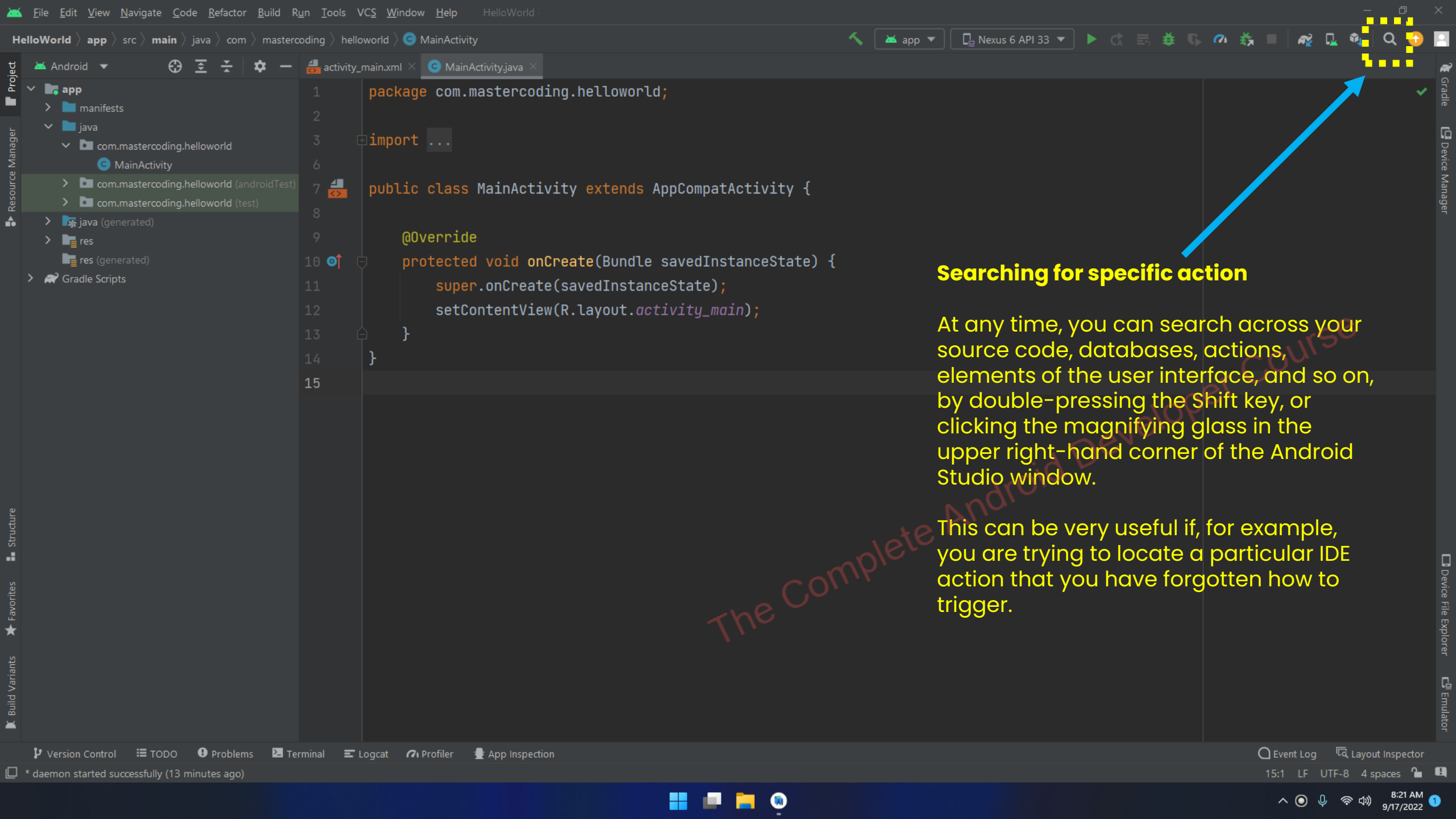
The tool window bar

runs around the outside of the IDE window and contains the buttons that allow you to expand or collapse individual tool windows.



The status bar

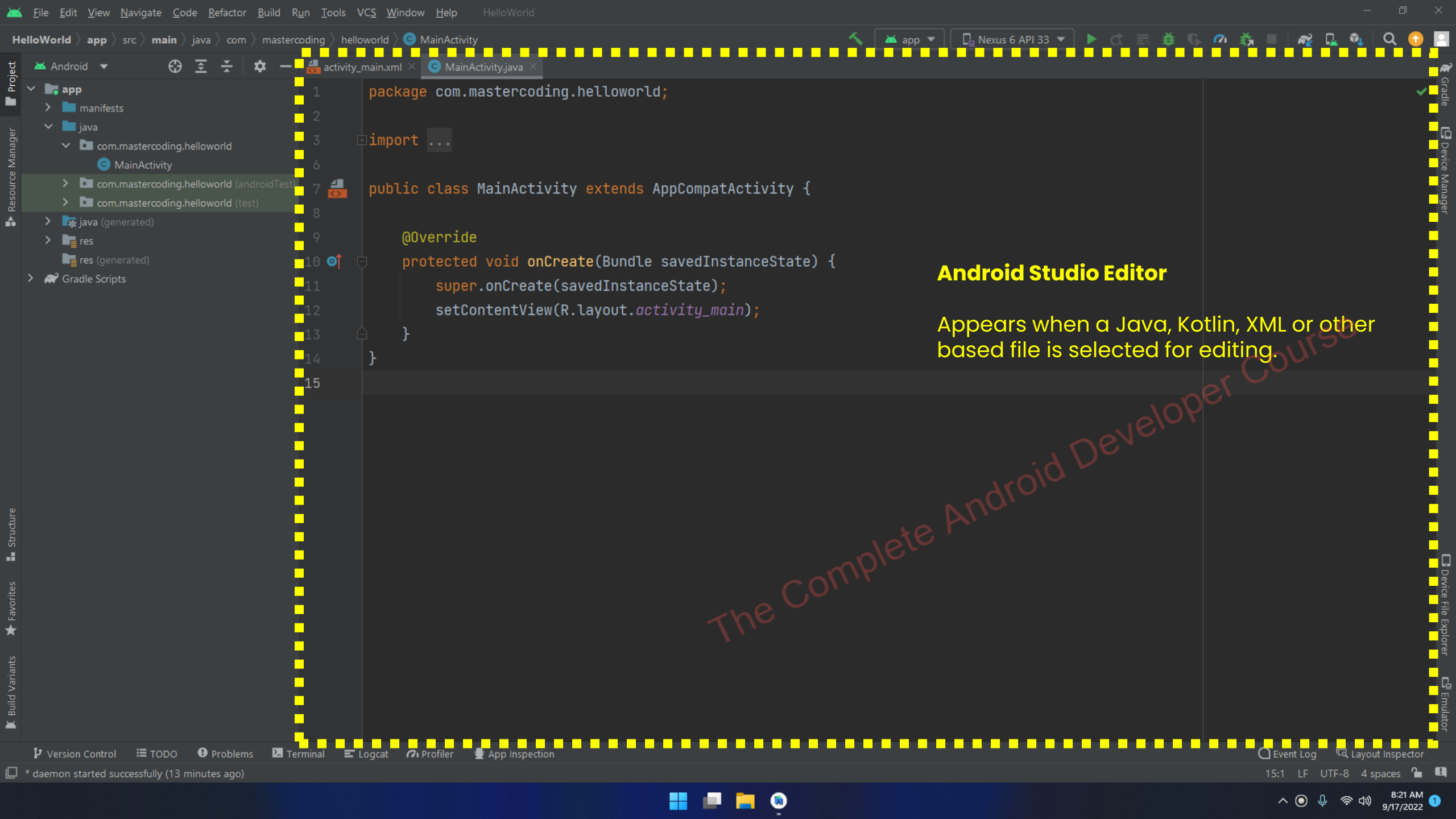
displays the status of your project and the IDE itself, as well as any warnings or messages.



Searching for specific action

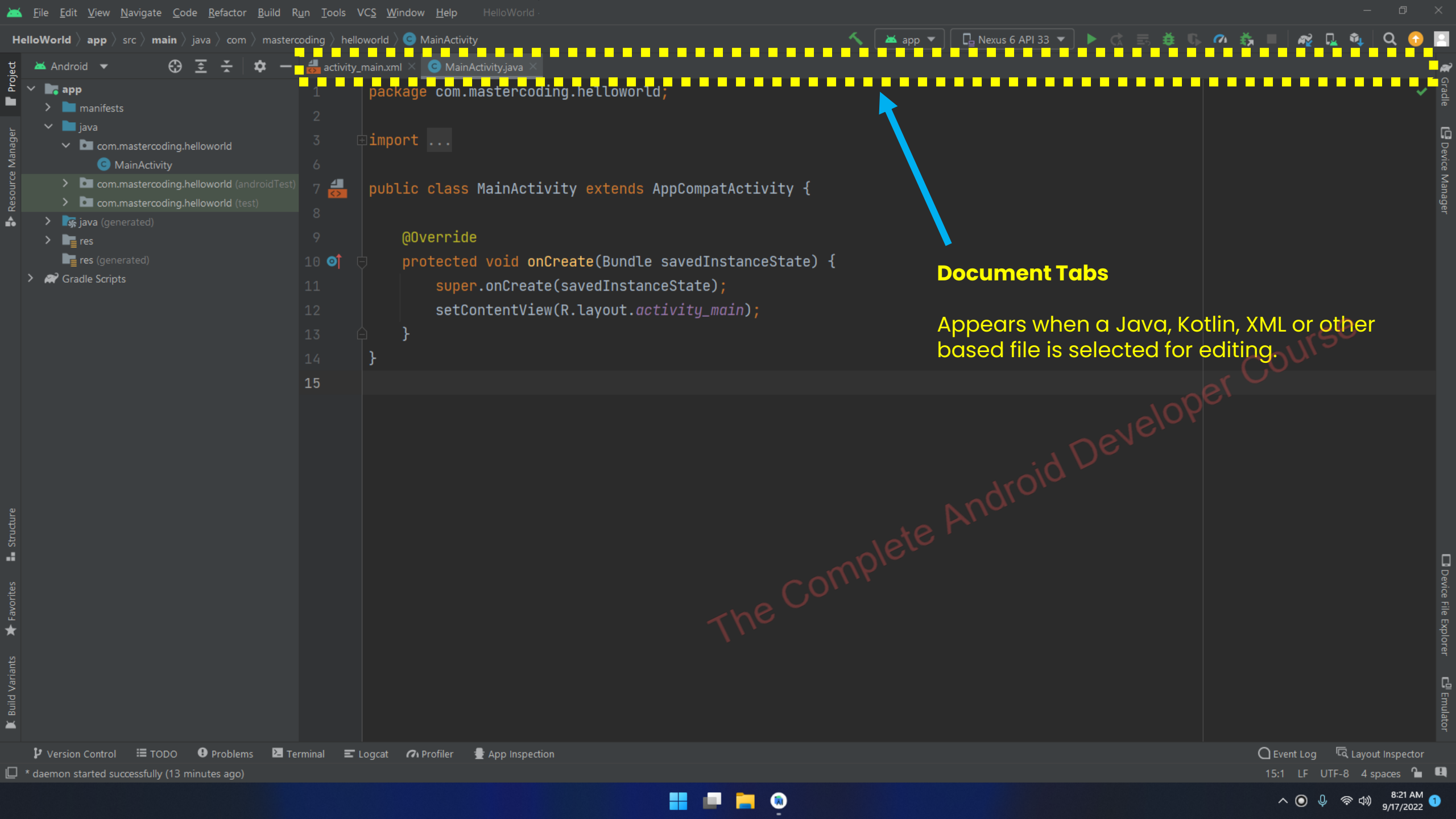
At any time, you can search across your source code, databases, actions, elements of the user interface, and so on, by double-pressing the Shift key, or clicking the magnifying glass in the upper right-hand corner of the Android Studio window.

This can be very useful if, for example, you are trying to locate a particular IDE action that you have forgotten how to trigger.



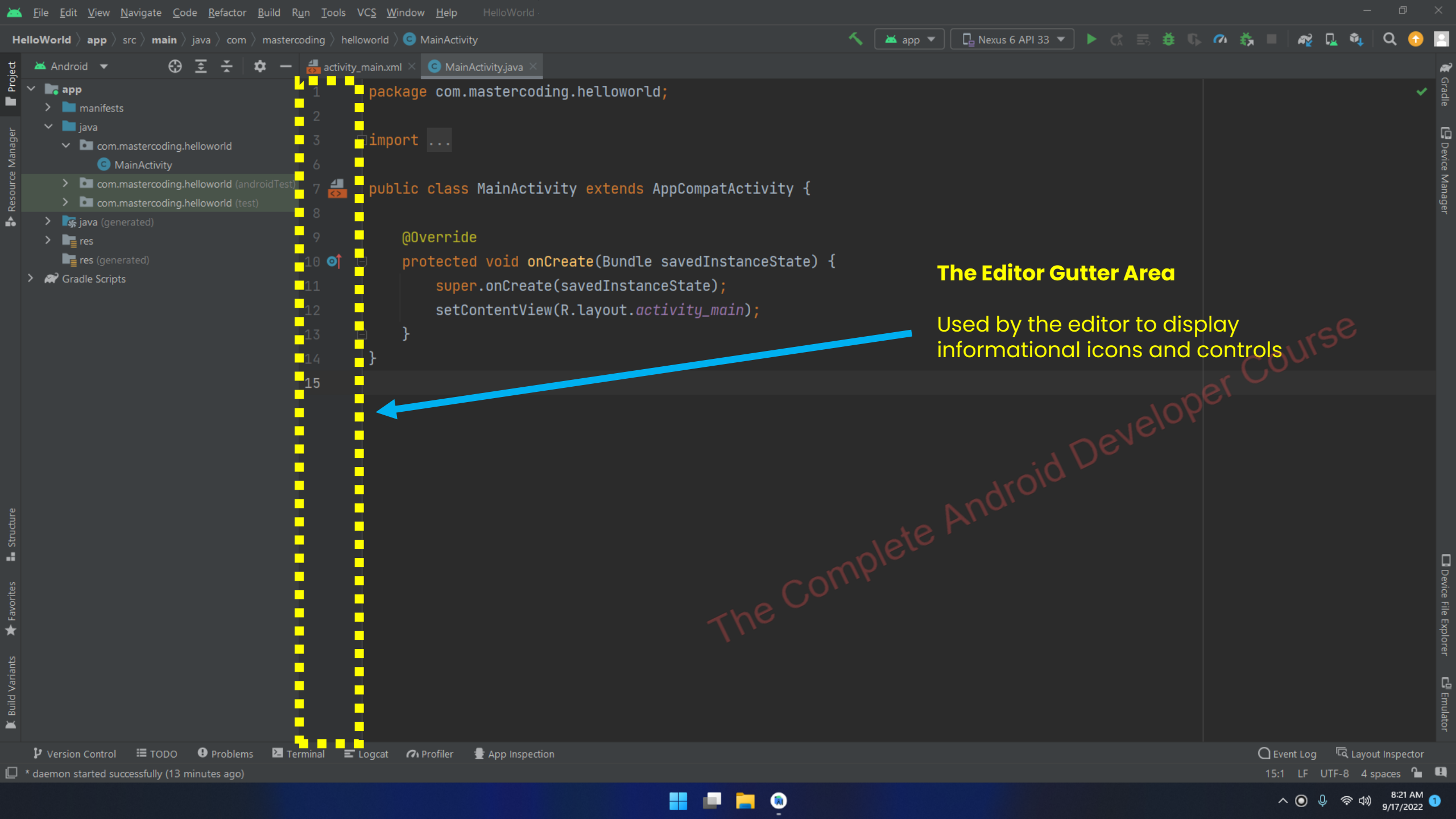
Android Studio Editor

Appears when a Java, Kotlin, XML or other based file is selected for editing.



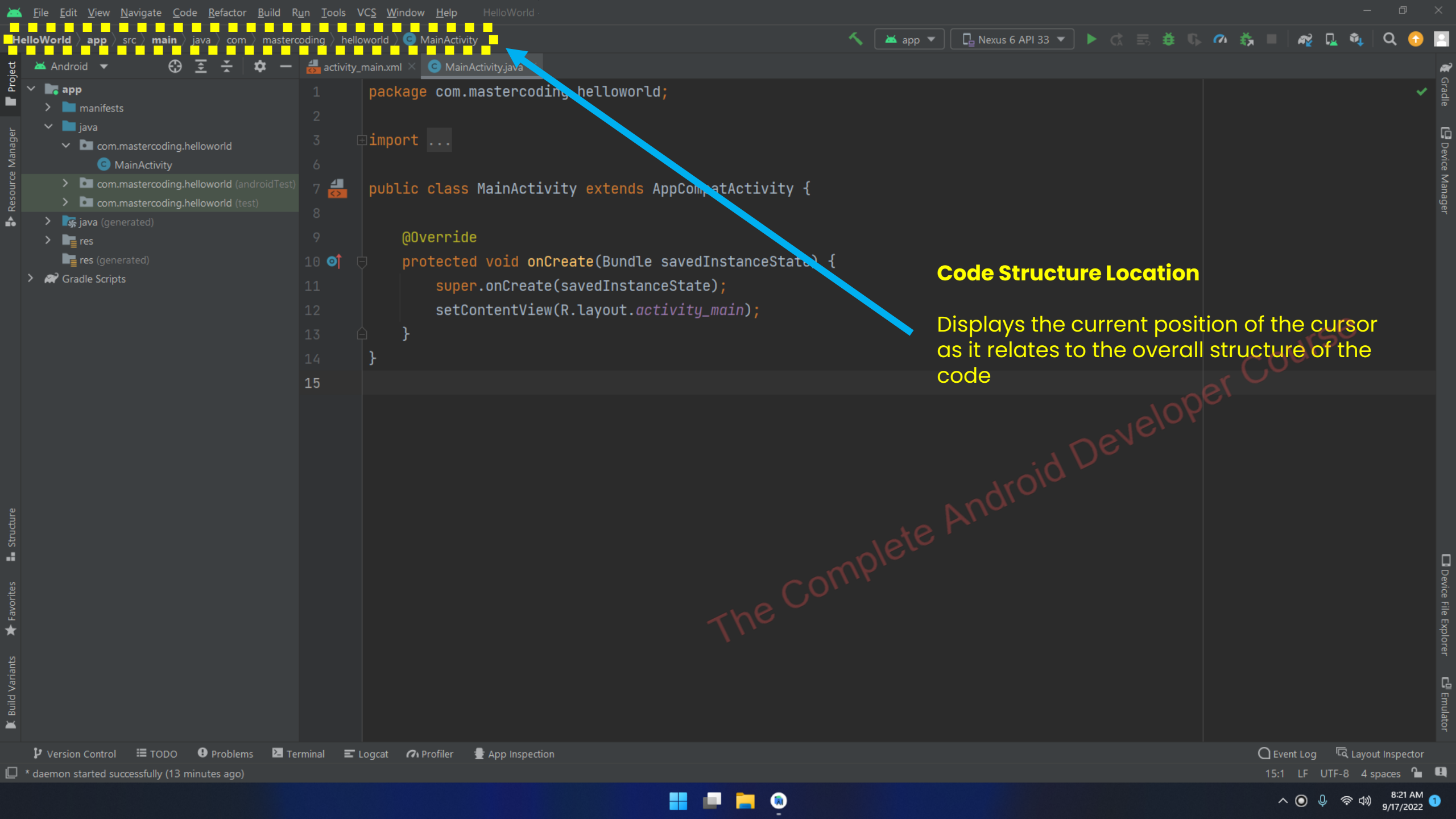
Document Tabs

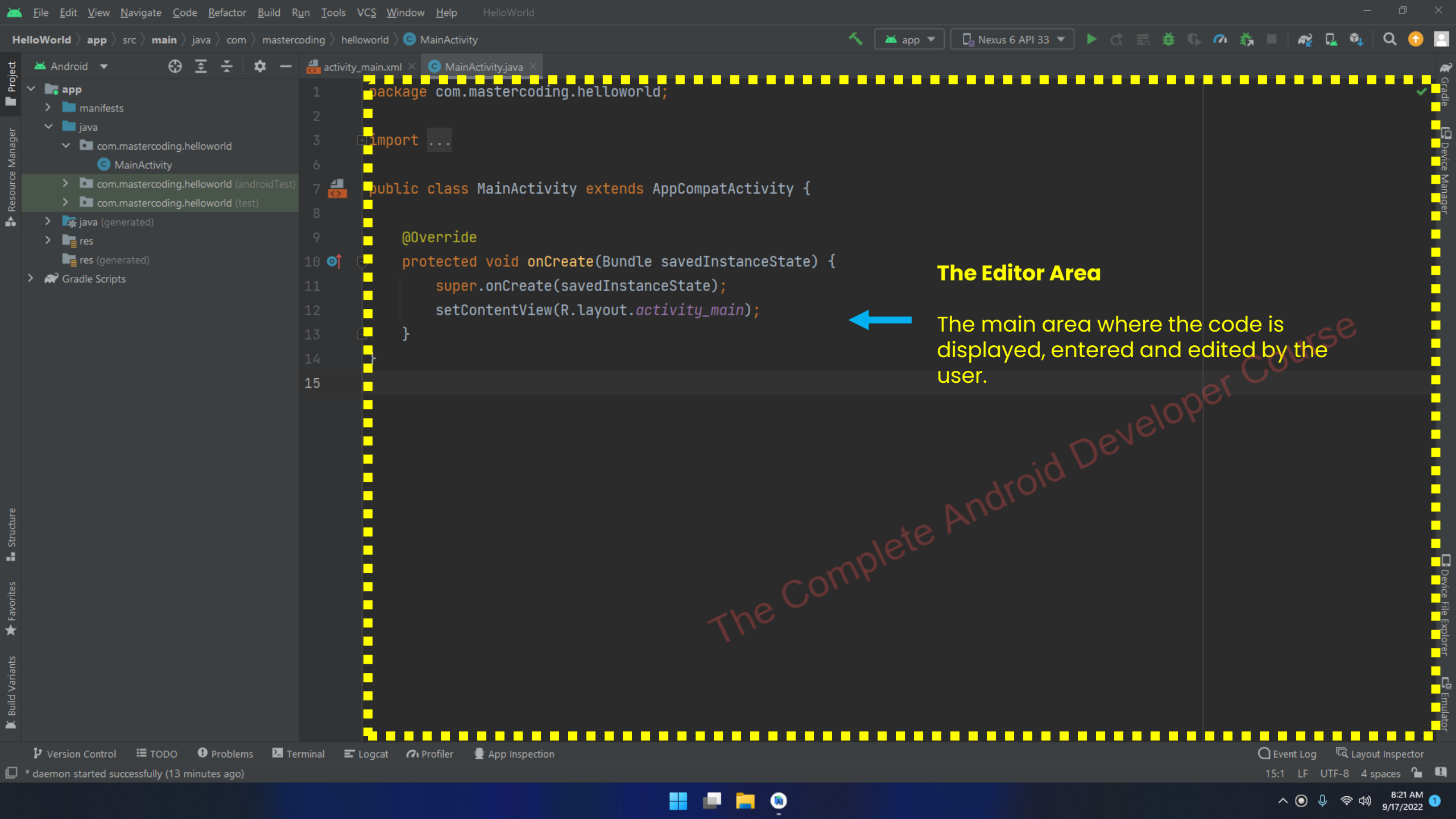
Appears when a Java, Kotlin, XML or other based file is selected for editing.



The Editor Gutter Area

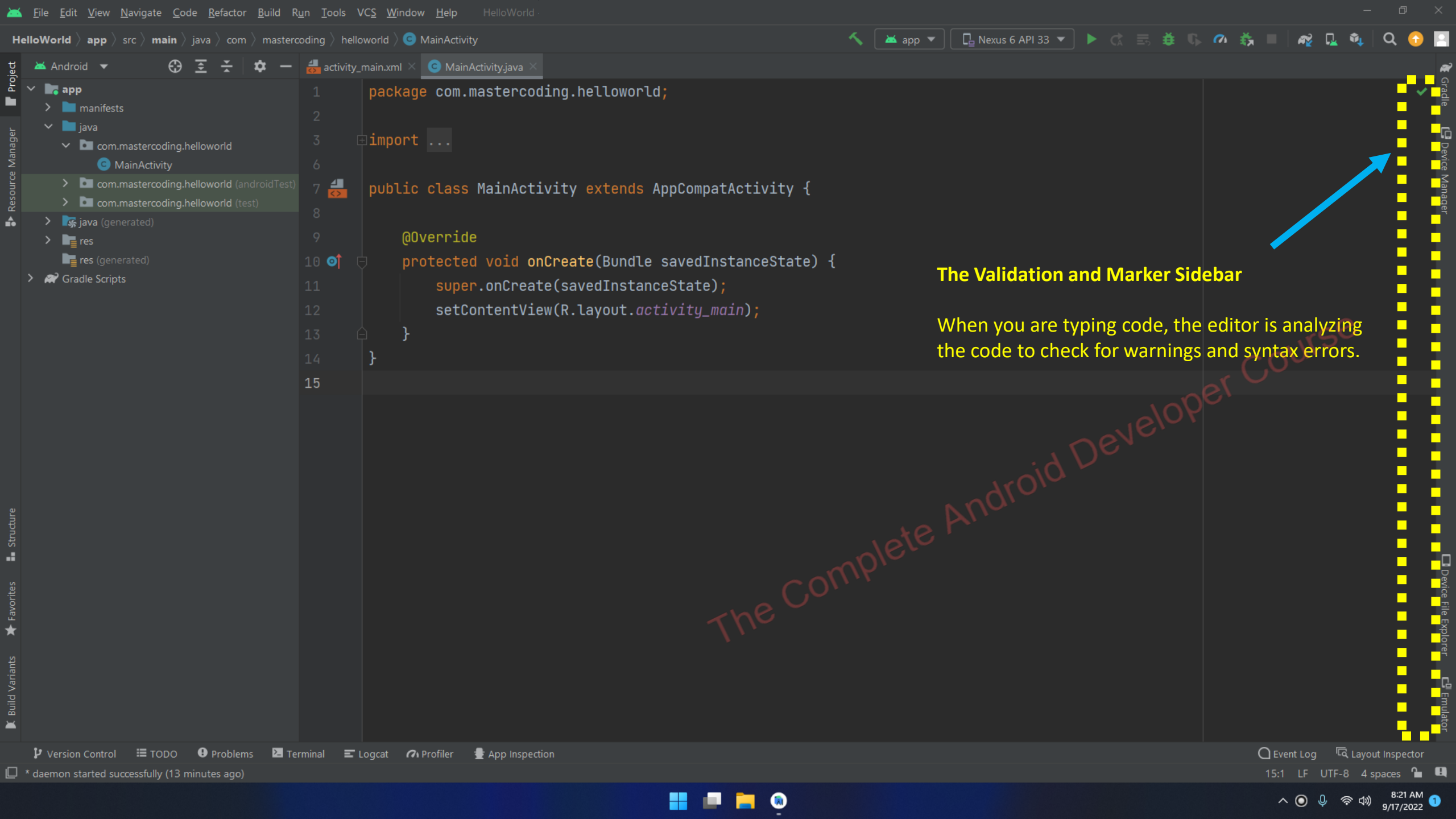
Used by the editor to display informational icons and controls





The Editor Area

The main area where the code is displayed, entered and edited by the user.

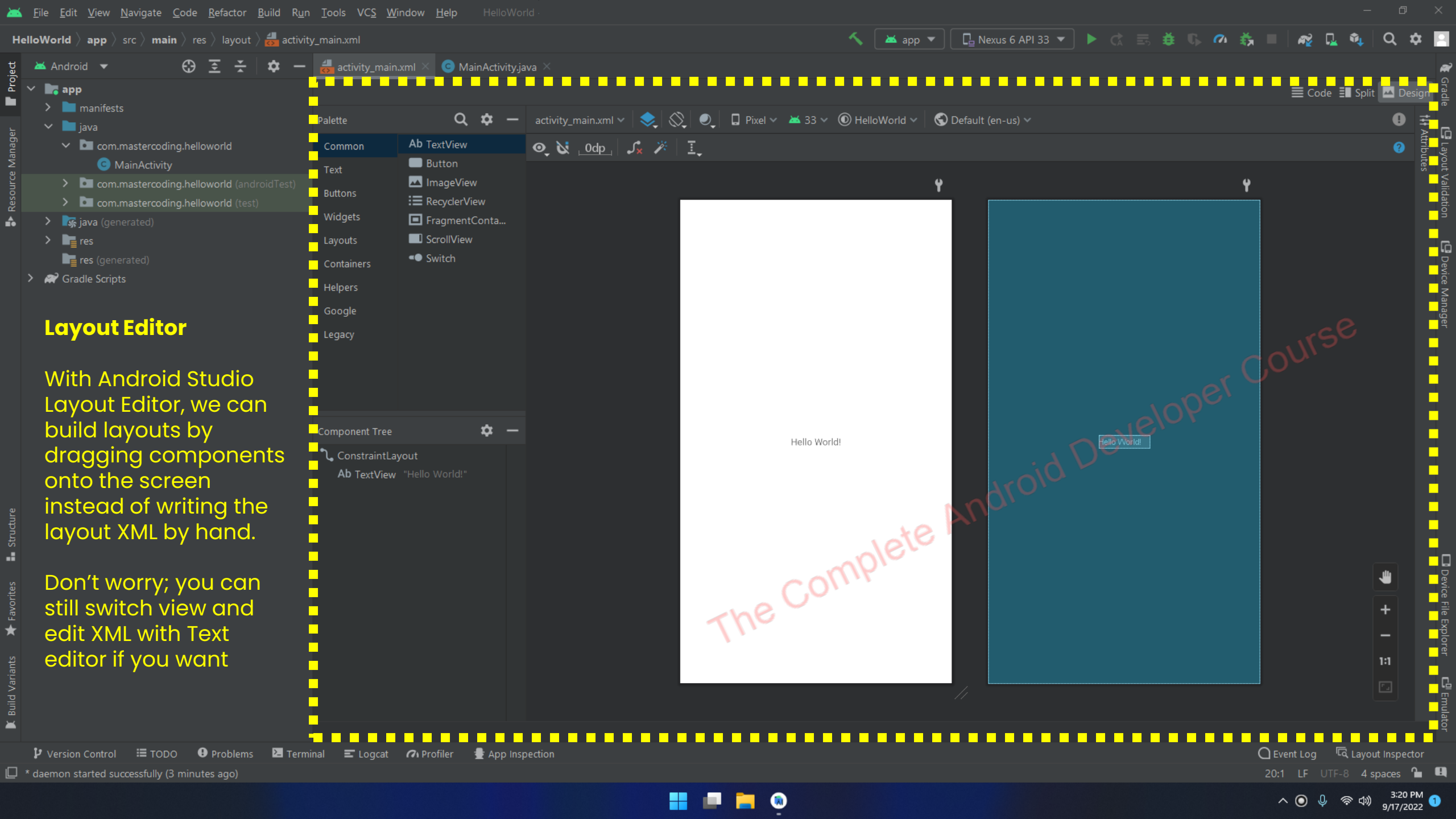


```
1 package com.mastercoding.helloworld;
2
3 import ...
6
7 public class MainActivity extends AppCompatActivity {
8
9     @Override
10    protected void onCreate(Bundle savedInstanceState) {
11        super.onCreate(savedInstanceState);
12        setContentView(R.layout.activity_main);
13    }
14 }
15
```

The Validation and Marker Sidebar

When you are typing code, the editor is analyzing the code to check for warnings and syntax errors.

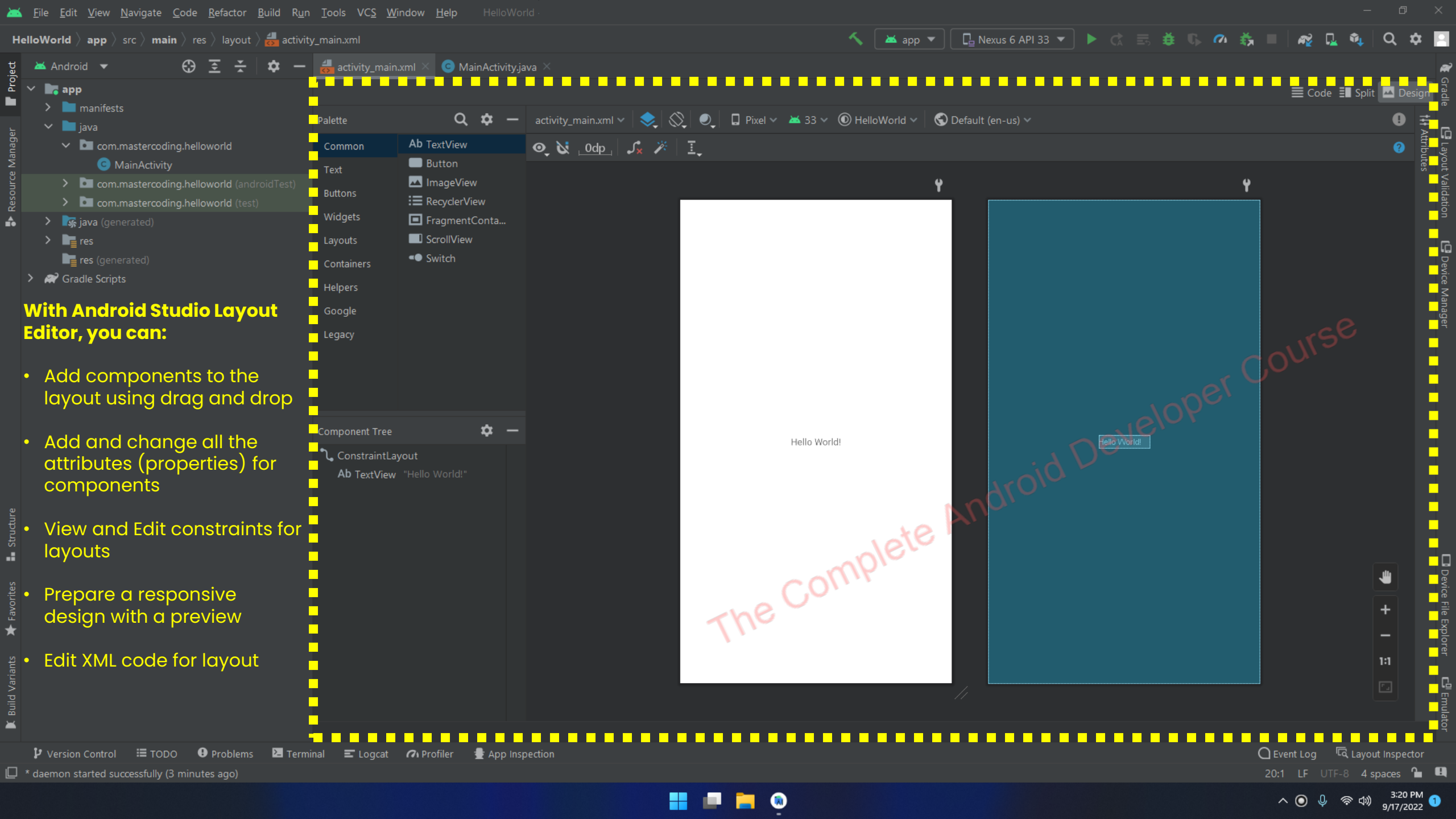
The Complete Android Developer Course



Layout Editor

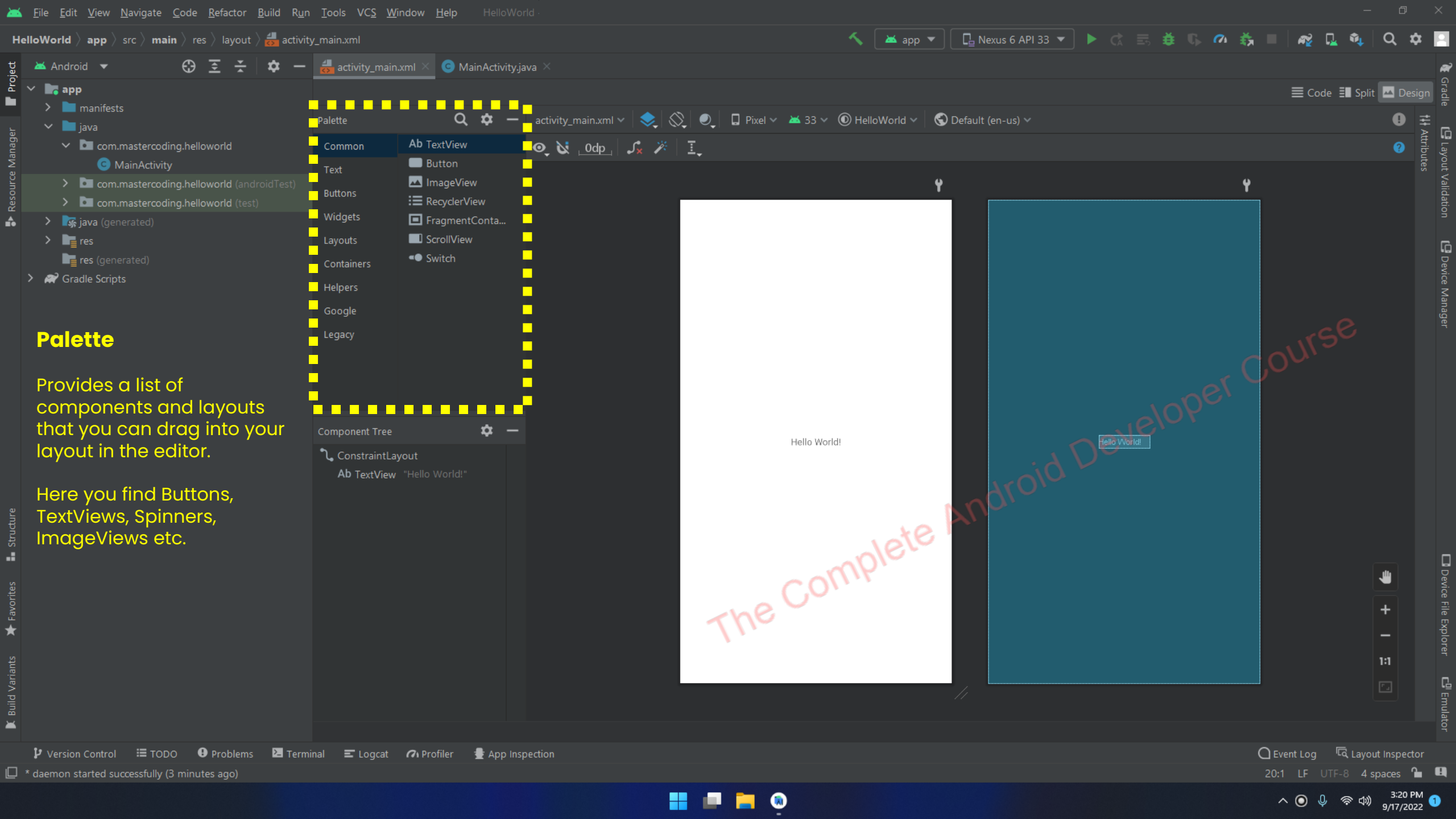
With Android Studio Layout Editor, we can build layouts by dragging components onto the screen instead of writing the layout XML by hand.

Don't worry; you can still switch view and edit XML with Text editor if you want



With Android Studio Layout Editor, you can:

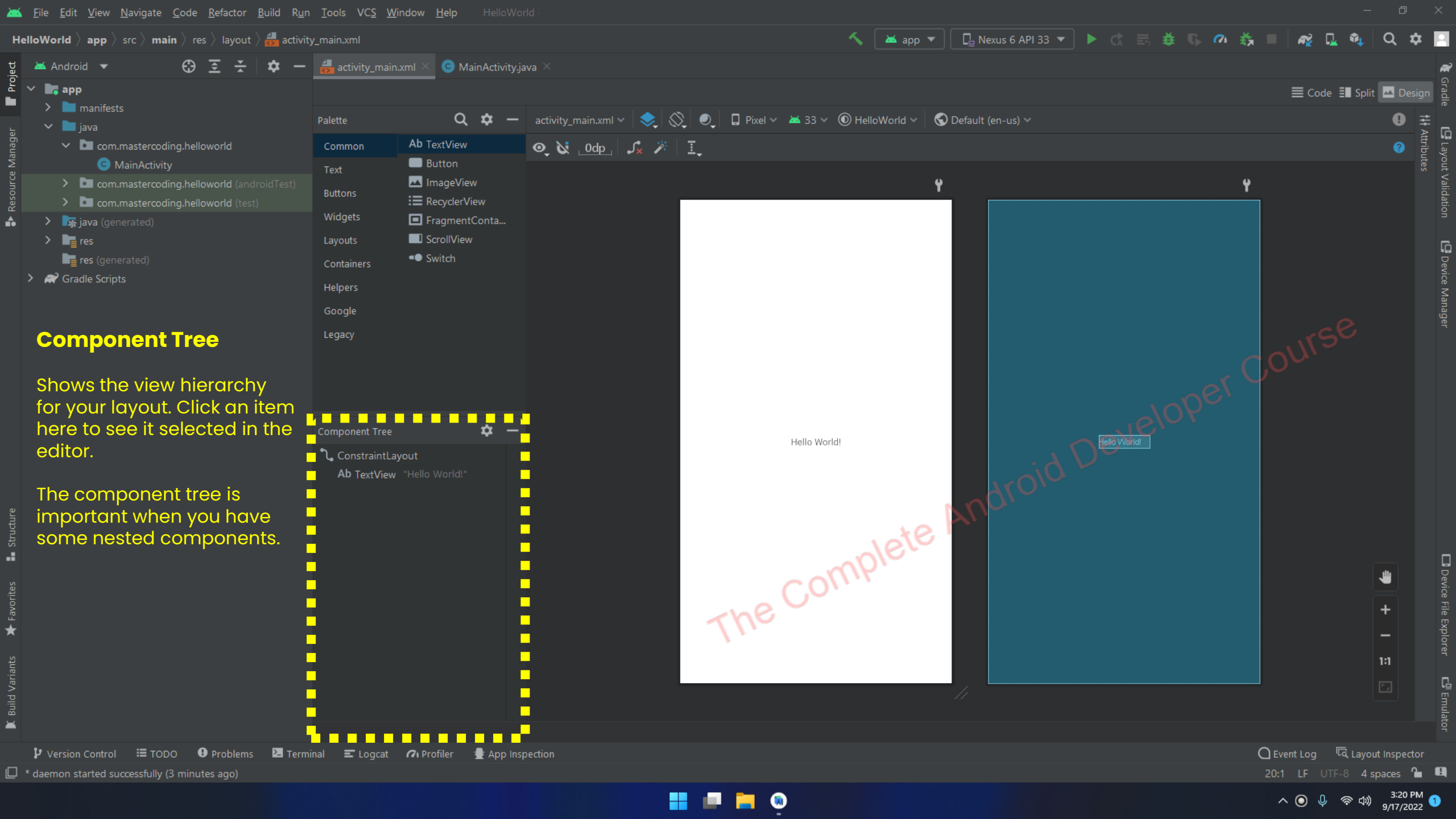
- Add components to the layout using drag and drop
- Add and change all the attributes (properties) for components
- View and Edit constraints for layouts
- Prepare a responsive design with a preview
- Edit XML code for layout



Palette

Provides a list of components and layouts that you can drag into your layout in the editor.

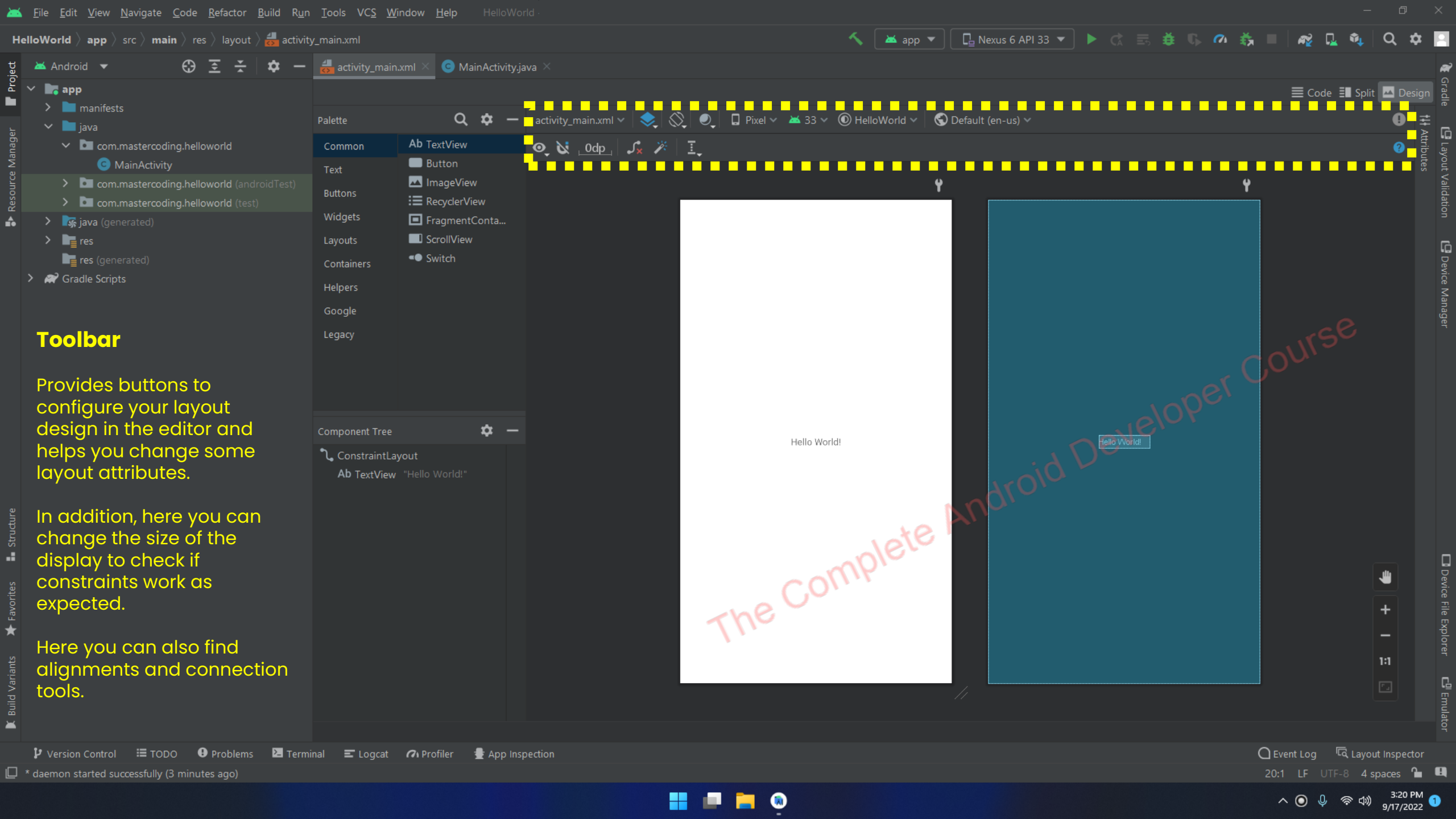
Here you find Buttons, TextViews, Spinners, ImageViews etc.



Component Tree

Shows the view hierarchy for your layout. Click an item here to see it selected in the editor.

The component tree is important when you have some nested components.

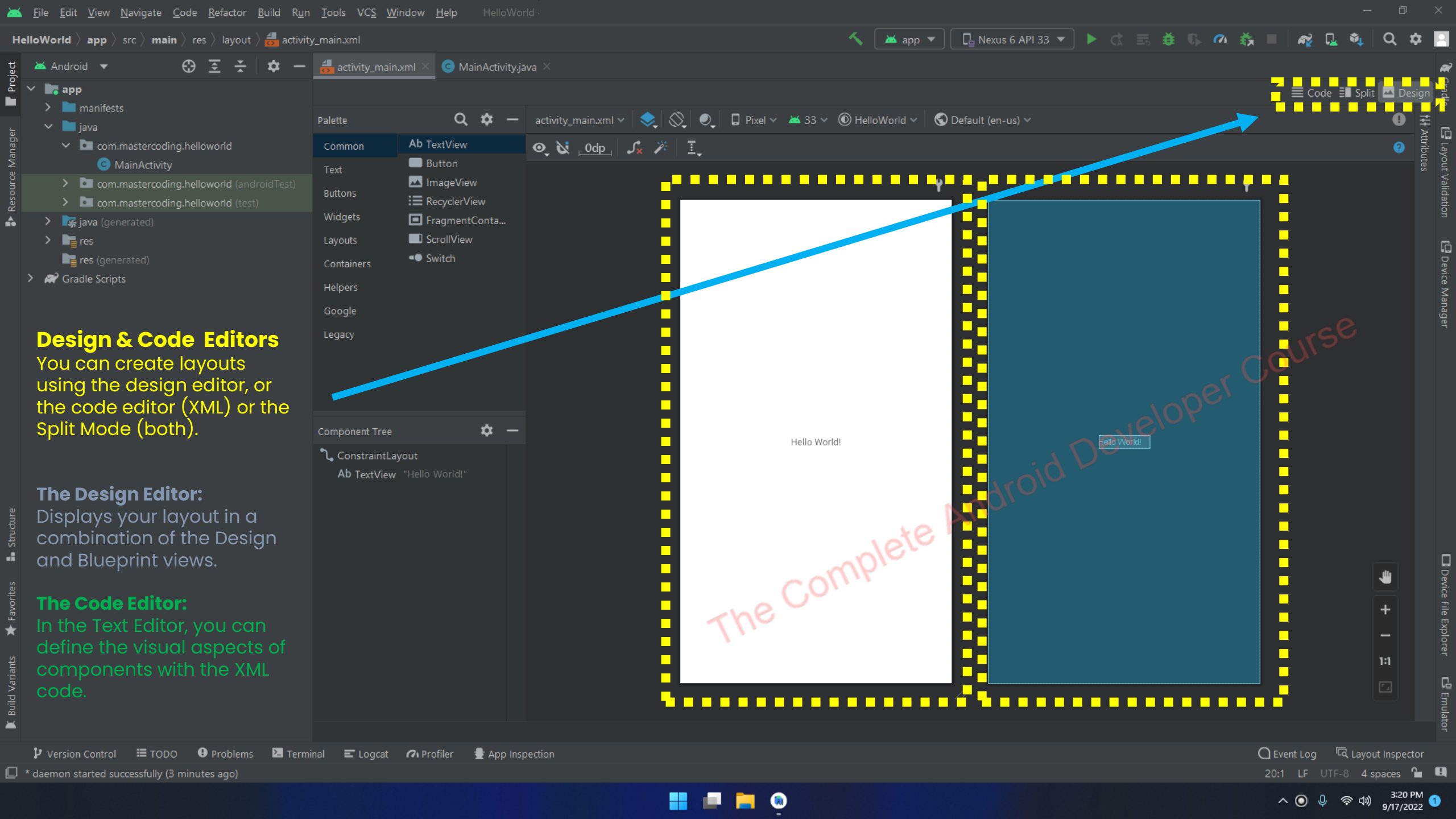


Toolbar

Provides buttons to configure your layout design in the editor and helps you change some layout attributes.

In addition, here you can change the size of the display to check if constraints work as expected.

Here you can also find alignments and connection tools.



Design & Code Editors

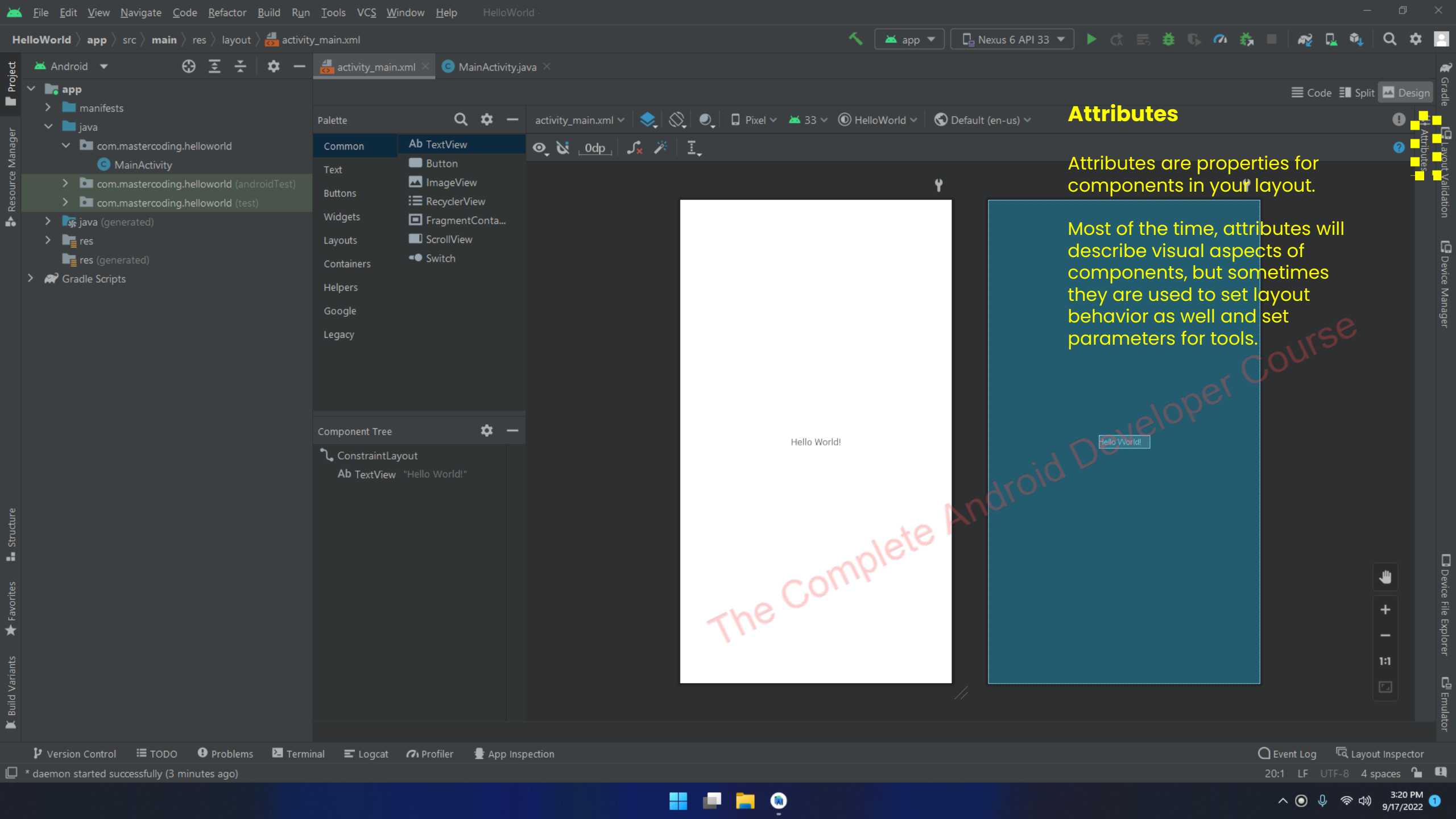
You can create layouts using the design editor, or the code editor (XML) or the Split Mode (both).

The Design Editor:

Displays your layout in a combination of the Design and Blueprint views.

The Code Editor:

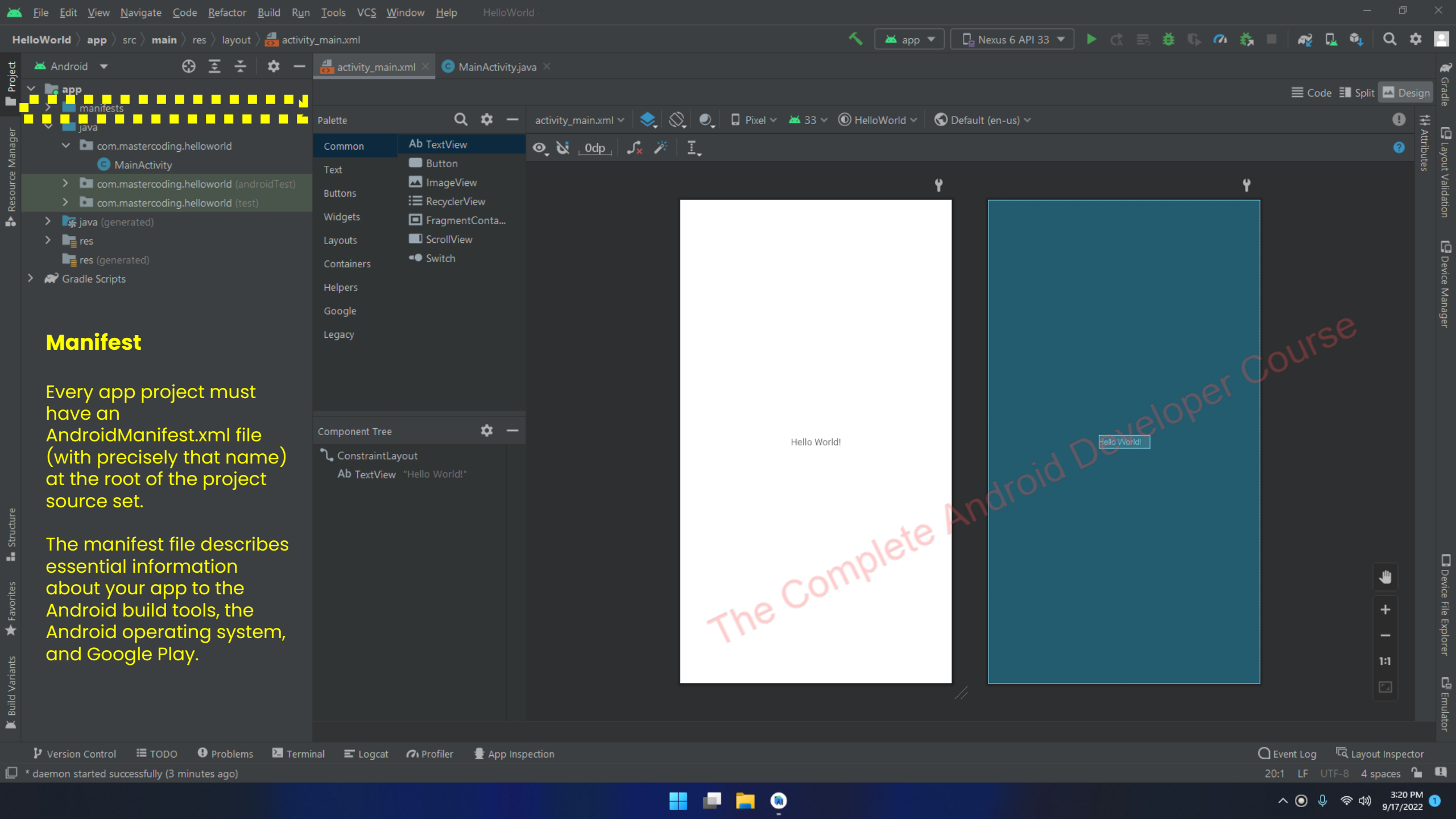
In the Text Editor, you can define the visual aspects of components with the XML code.



Attributes

Attributes are properties for components in your layout.

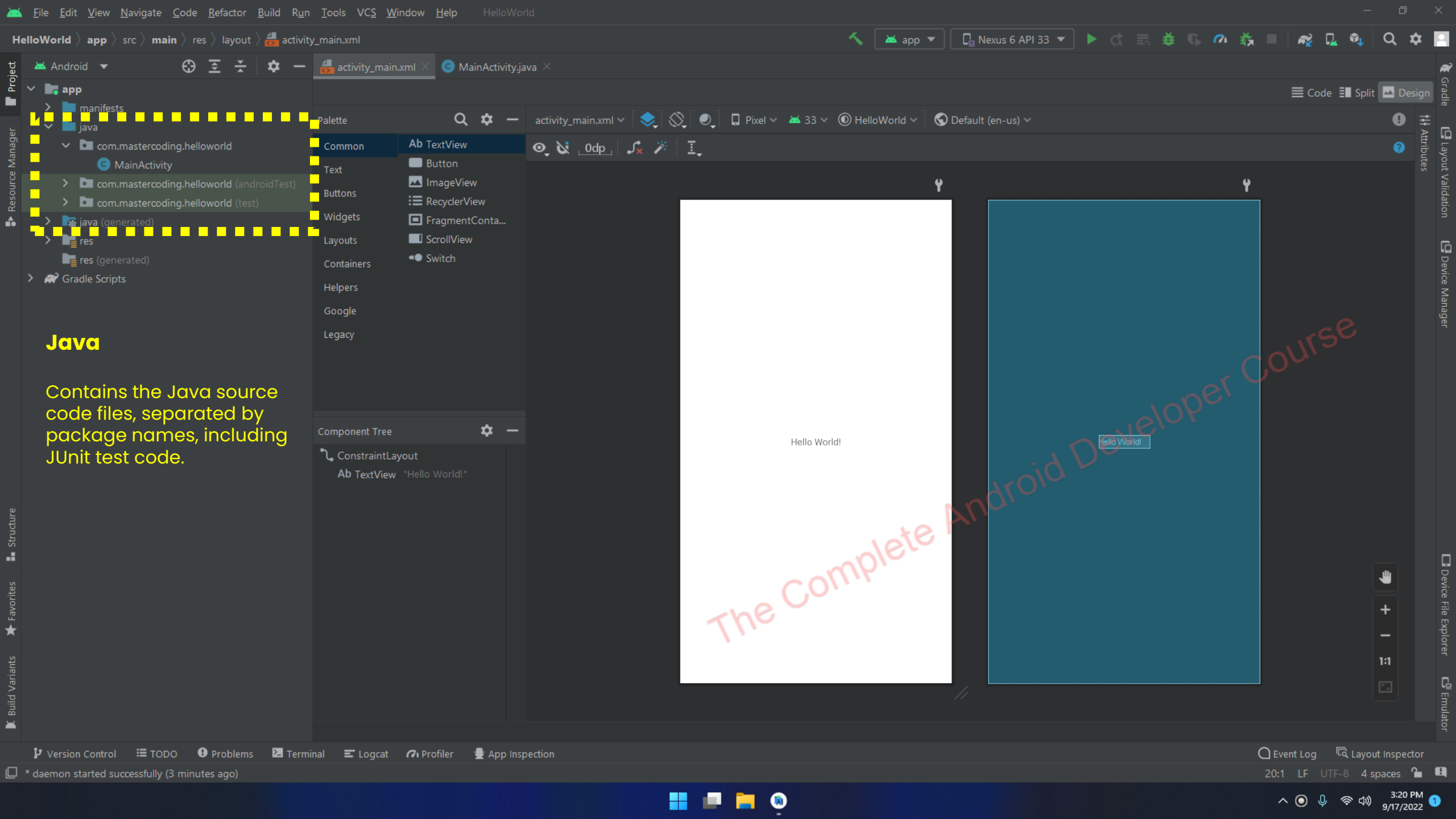
Most of the time, attributes will describe visual aspects of components, but sometimes they are used to set layout behavior as well and set parameters for tools.



Manifest

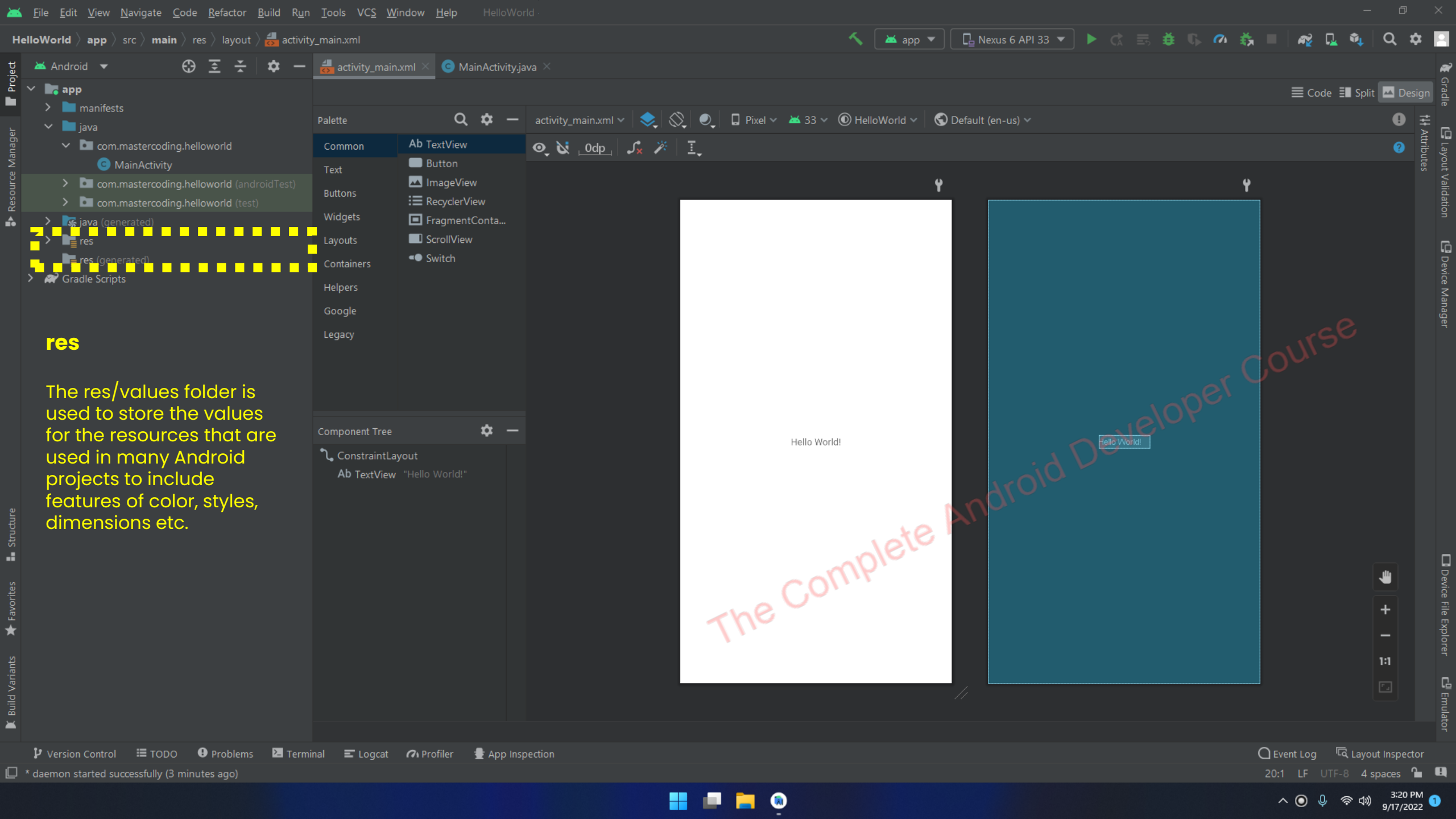
Every app project must have an **AndroidManifest.xml** file (with precisely that name) at the root of the project source set.

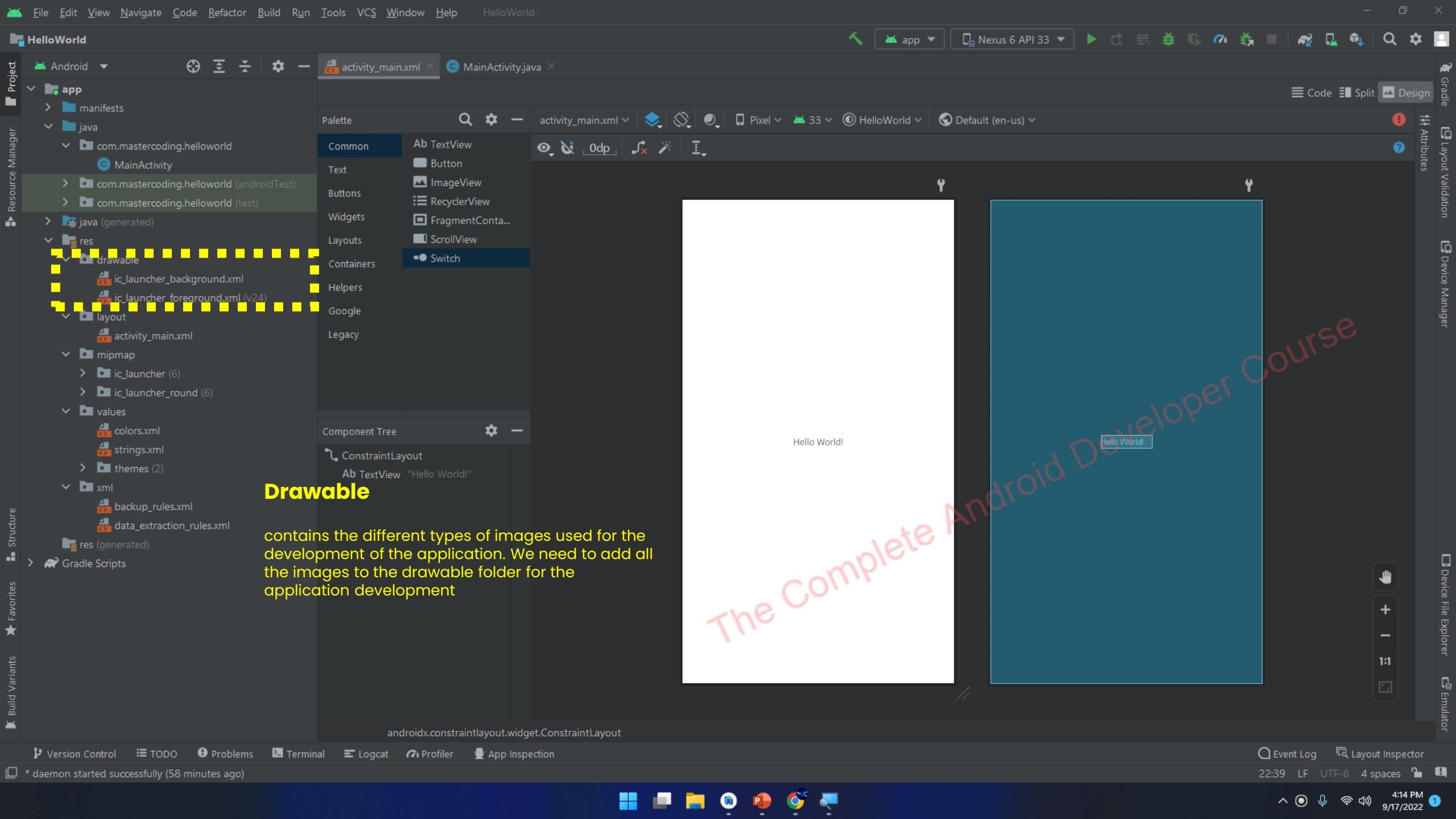
The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.



Java

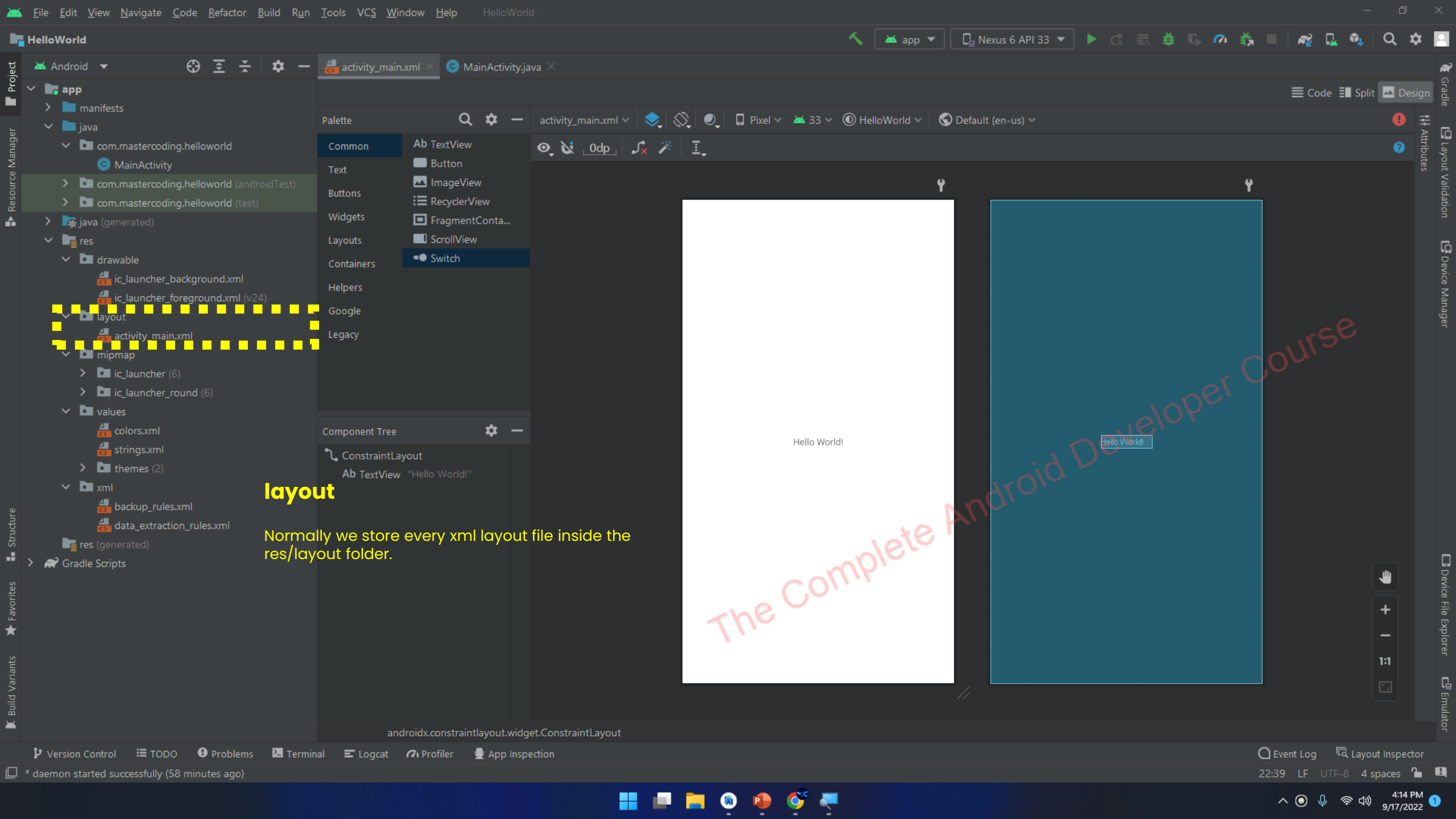
Contains the Java source code files, separated by package names, including JUnit test code.





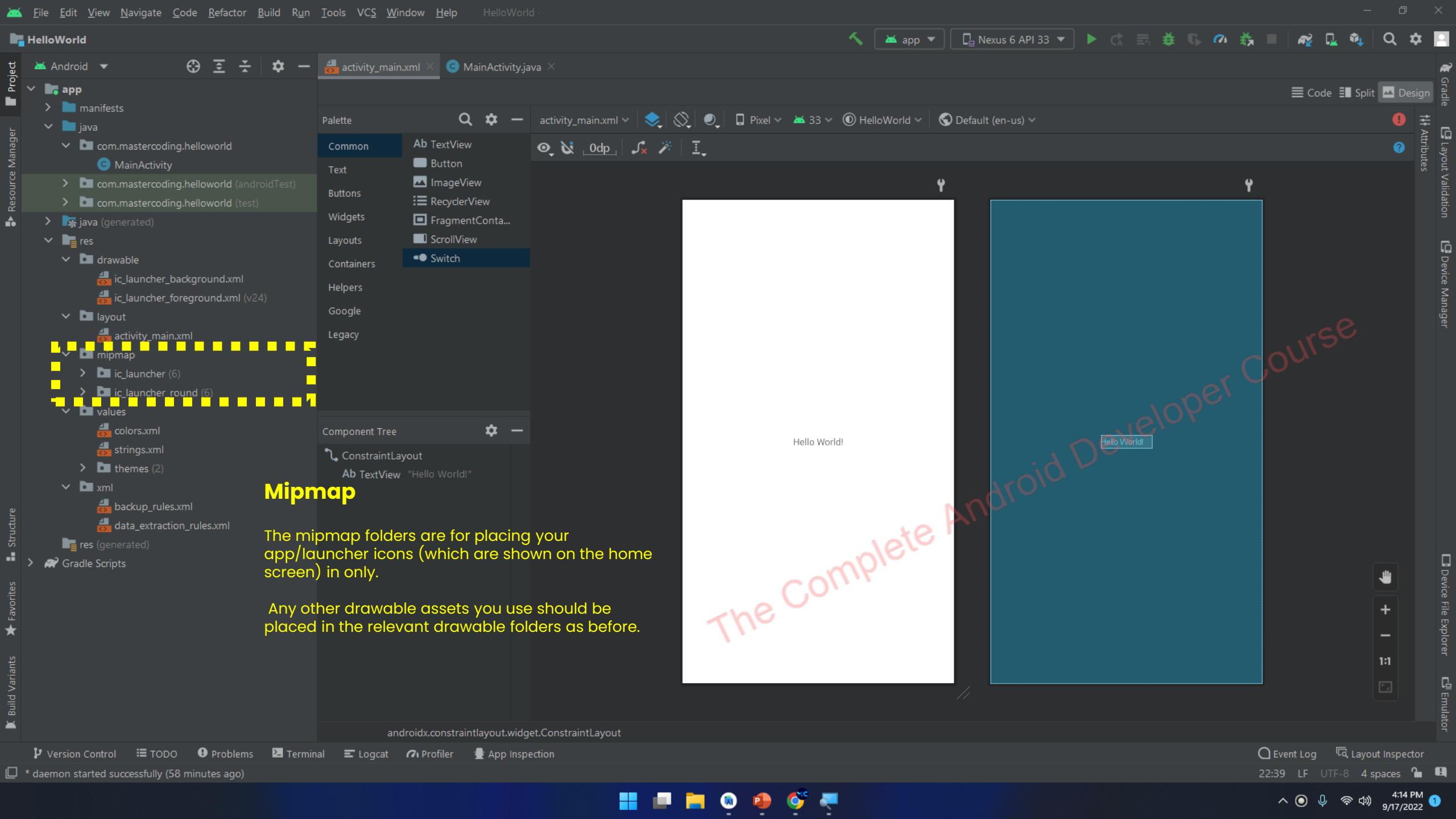
Drawable

contains the different types of images used for the development of the application. We need to add all the images to the drawable folder for the application development



layout

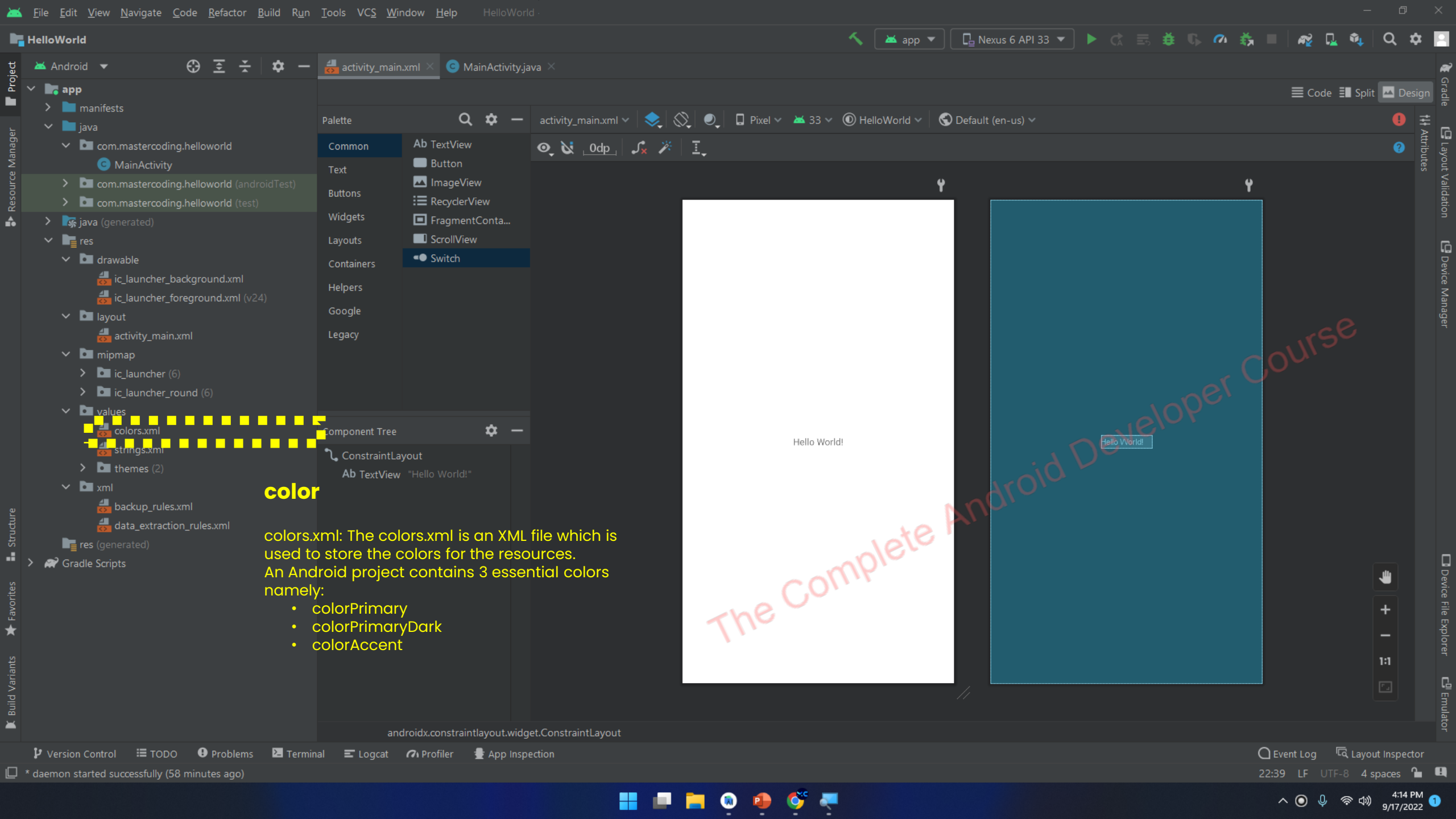
Normally we store every xml layout file inside the res/layout folder.



Mipmap

The mipmap folders are for placing your app/launcher icons (which are shown on the home screen) in only.

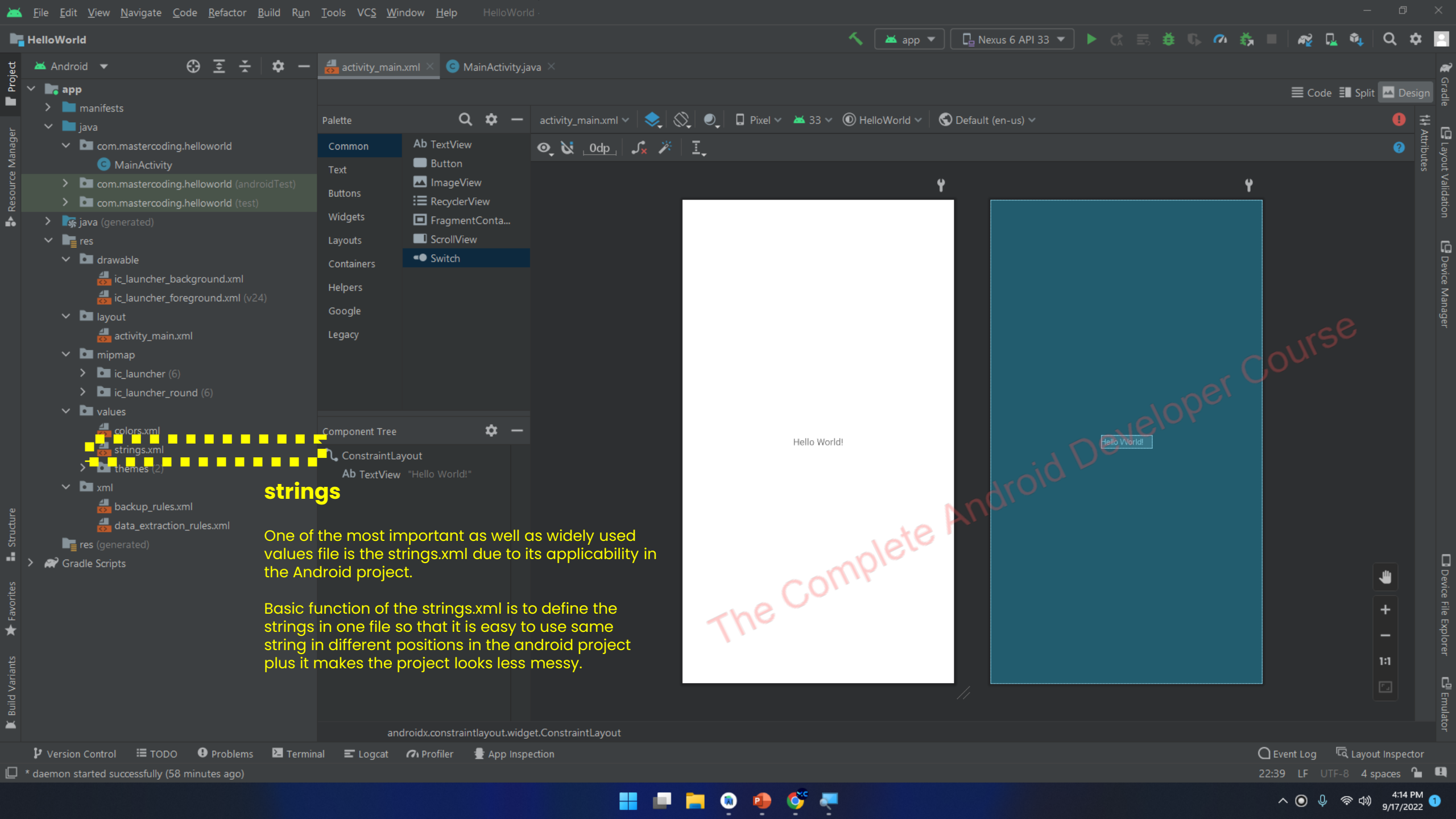
Any other drawable assets you use should be placed in the relevant drawable folders as before.



color

colors.xml: The colors.xml is an XML file which is used to store the colors for the resources. An Android project contains 3 essential colors namely:

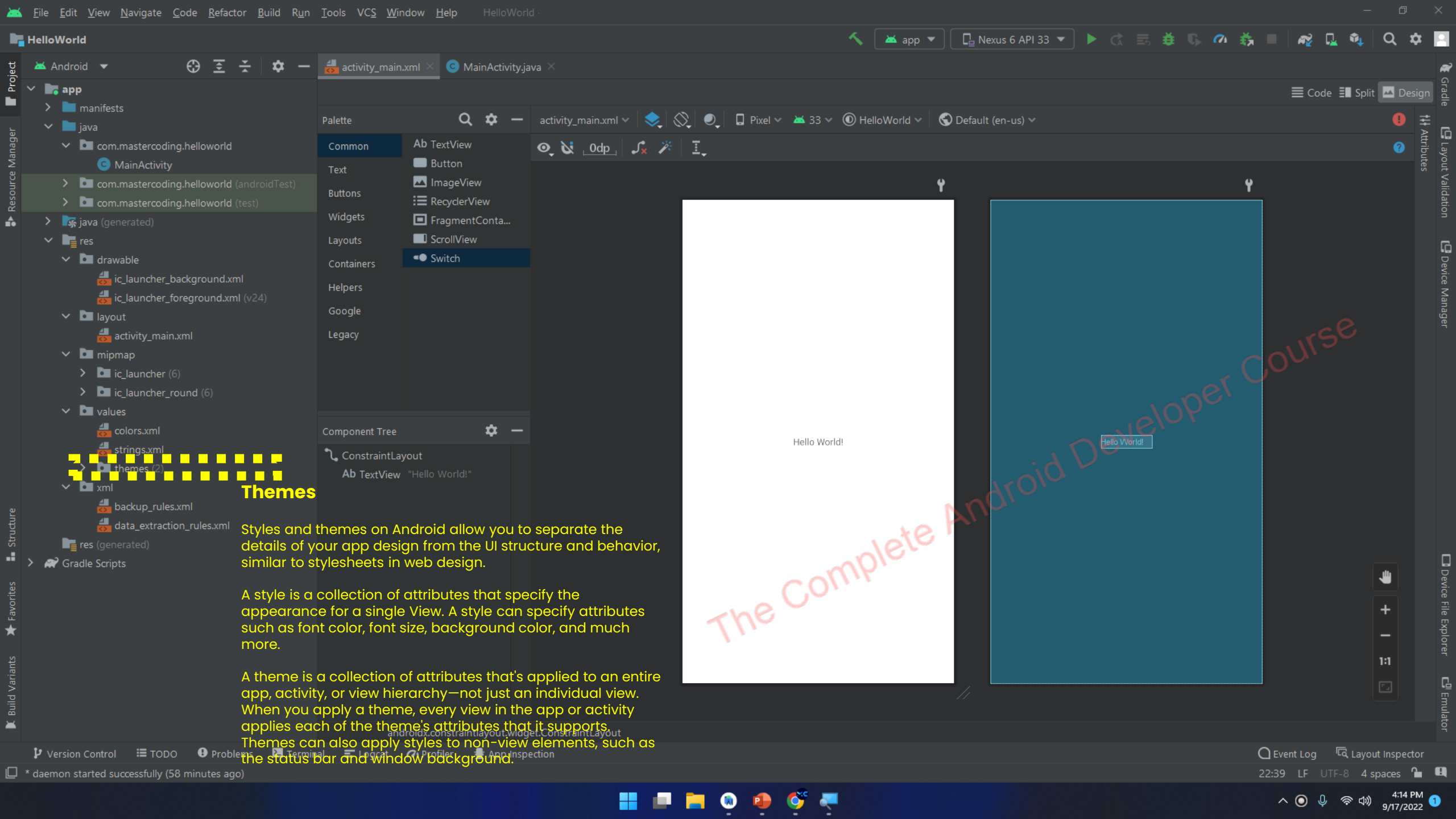
- colorPrimary
- colorPrimaryDark
- colorAccent



strings

One of the most important as well as widely used values file is the strings.xml due to its applicability in the Android project.

Basic function of the strings.xml is to define the strings in one file so that it is easy to use same string in different positions in the android project plus it makes the project looks less messy.

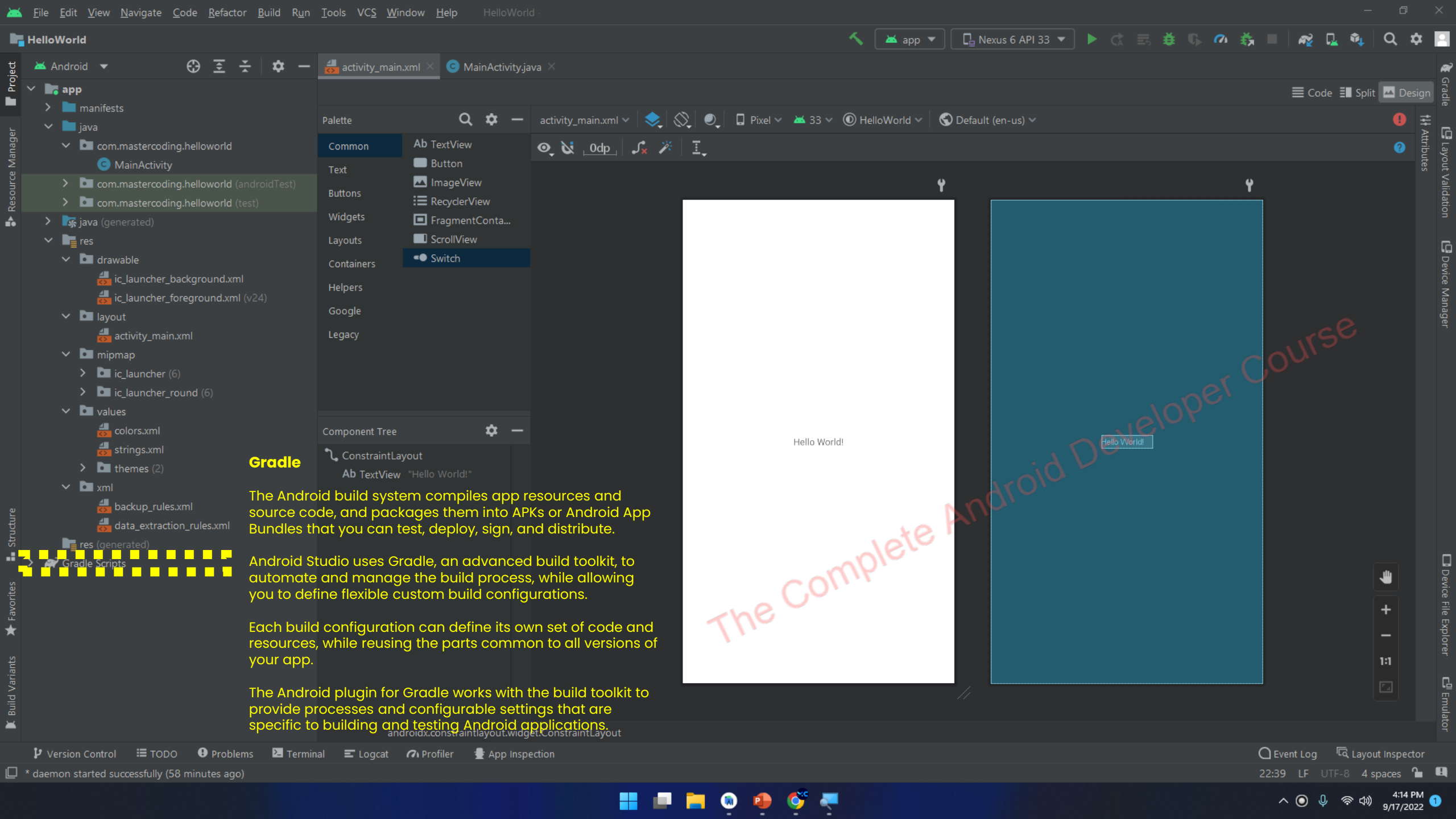


Themes

Styles and themes on Android allow you to separate the details of your app design from the UI structure and behavior, similar to stylesheets in web design.

A style is a collection of attributes that specify the appearance for a single View. A style can specify attributes such as font color, font size, background color, and much more.

A theme is a collection of attributes that's applied to an entire app, activity, or view hierarchy—not just an individual view. When you apply a theme, every view in the app or activity applies each of the theme's attributes that it supports. Themes can also apply styles to non-view elements, such as the status bar and window background.



Gradle

The Android build system compiles app resources and source code, and packages them into APKs or Android App Bundles that you can test, deploy, sign, and distribute.

Android Studio uses Gradle, an advanced build toolkit, to automate and manage the build process, while allowing you to define flexible custom build configurations.

Each build configuration can define its own set of code and resources, while reusing the parts common to all versions of your app.

The Android plugin for Gradle works with the build toolkit to provide processes and configurable settings that are specific to building and testing Android applications.

The Views & ViewGroups

ViewGroup

A ViewGroup is a special subclass of the View class that serves as a container for multiple Views.

There are several useful widgets that extend the ViewGroup Class.

Most are actually subclasses of another ViewGroup known as a Layout.

Views

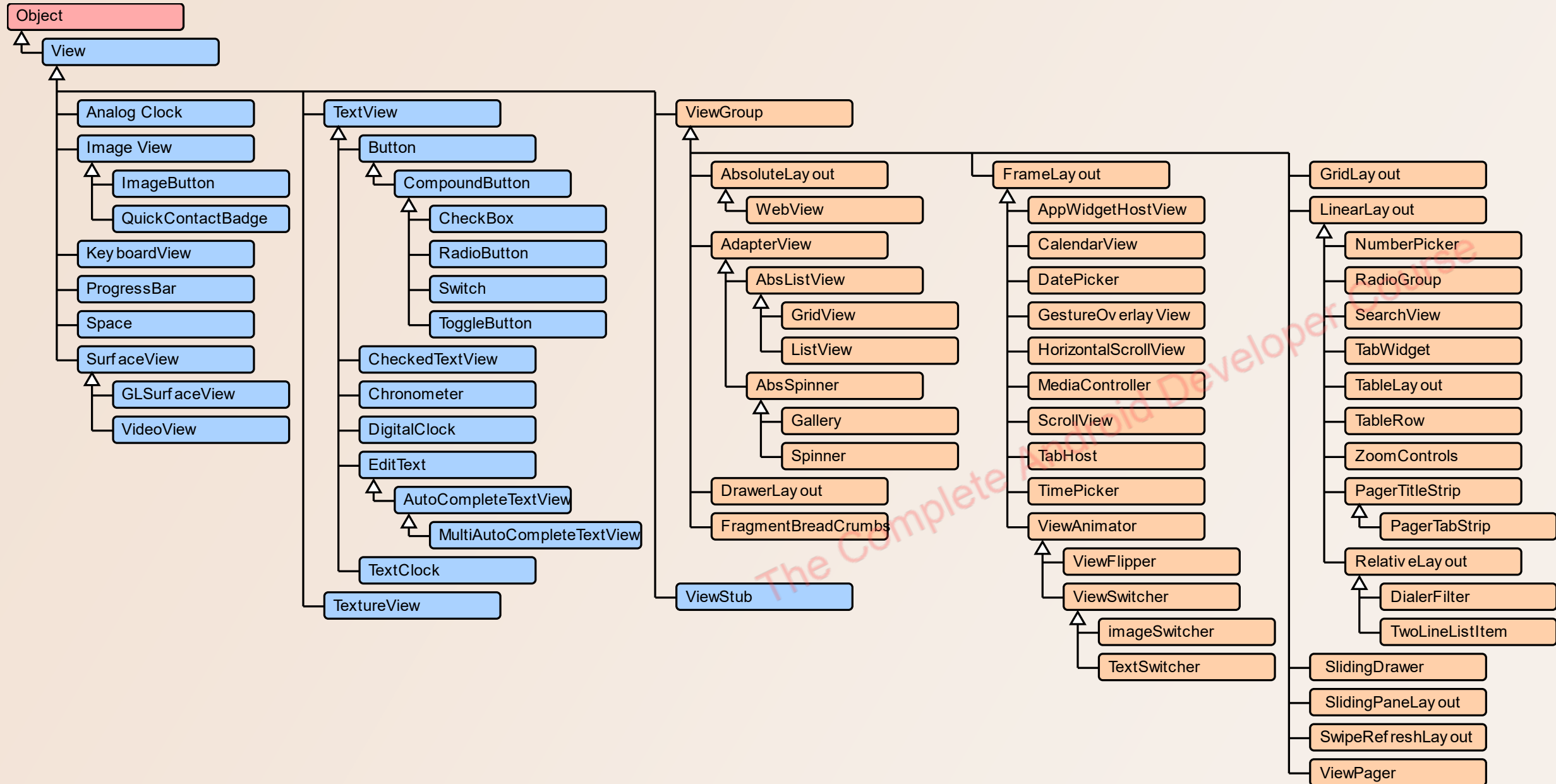
A View is a simple building block of a user interface.

It is a small rectangular box that can be TextView, EditText, or even a button.

It occupies the area on the screen in a rectangular area and is responsible for drawing and event handling.

The Complete Android Developer Course

The View Class



App Creation Steps

- 1- The Required Dependencies and Libraries
- 2- Adding Permissions
- 3- Designing the Layout
- 4- Adding the Functionalities and Logic
- 5- Running and Testing

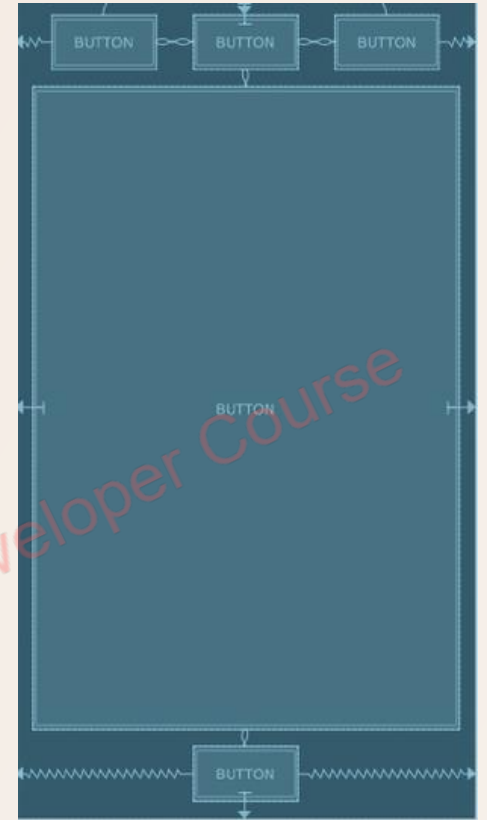
The Complete Android Developer Course

Constraint Layout

It is a View Group which solves problems like nesting and performance.

This not only helps developers to build more complex and large UIs, but also it comes with a flat hierarchy.

ConstraintLayout works based on the relationships between views and those are called as **Constraints**. The essential block of ConstraintLayout is developing constraints.



The Complete Android Developer Course

Constraint Layout

Views and widgets such as Buttons, Texts etc. are added inside the **«ConstraintLayout»** tag, which then becomes the parent to all the nested elements.

Anchor Point



Constraint Layout

Constraints' Attributes in XML:

app:layout_constraint<SOURCE_ANCHOR>_to<TARGET_ANCHOR>of="<TARGET_ID>"

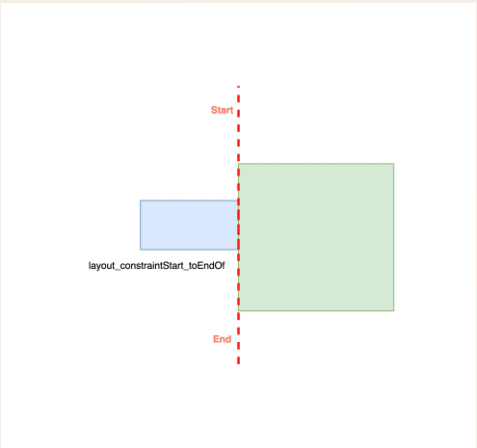
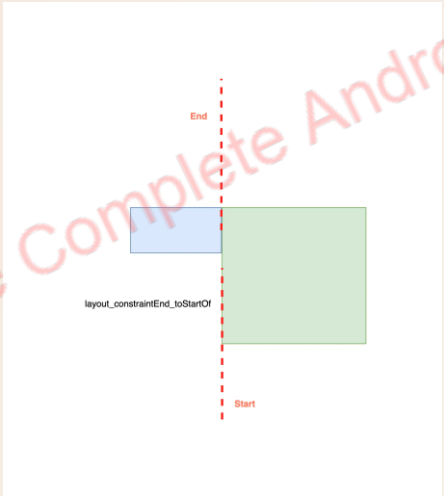
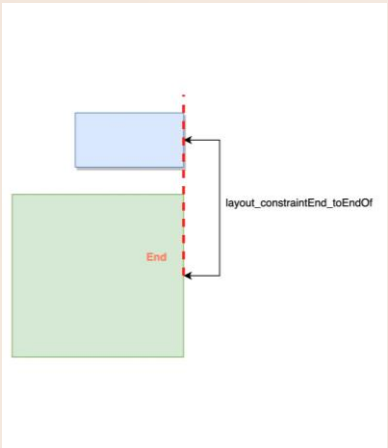
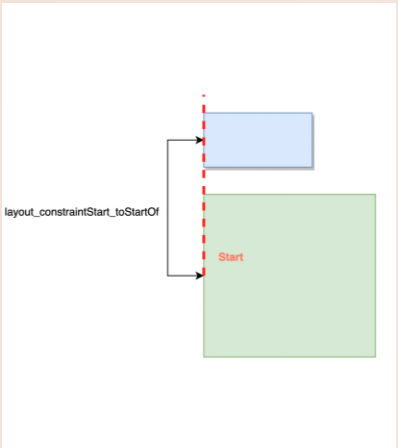
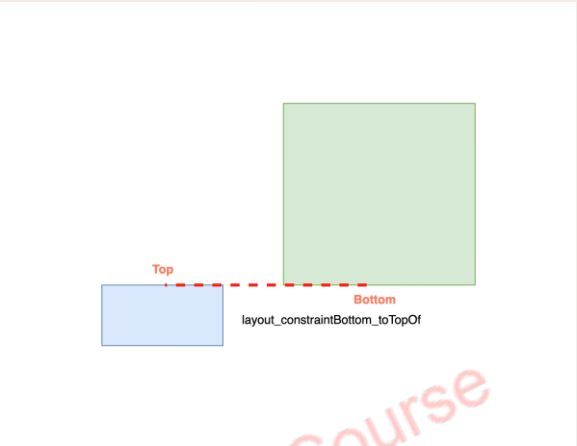
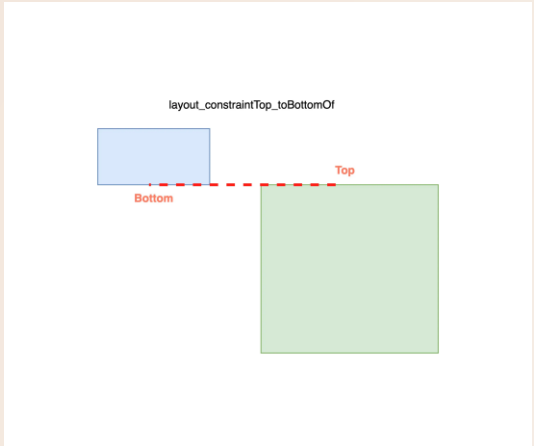
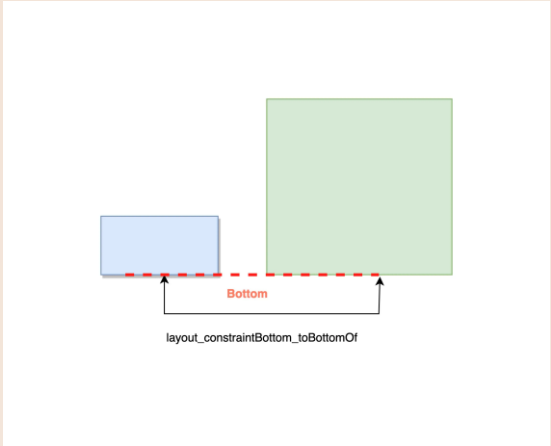
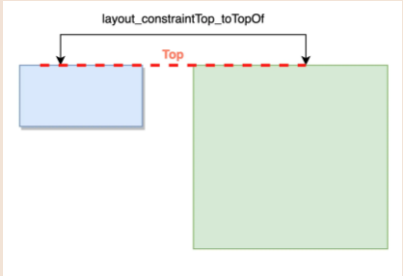
start
end
top
bottom

start
end
top
bottom

@+id/anyViewID
parent

The Complete Android Developer Course

Constraint Layout



The Complete Android Developer Course

View Size

You can use the corner handles to resize a view, but this hard codes the size so the view will not resize for different content or screen sizes. To select a different sizing mode, click a view and open the Attributes window on the right side of the editor.

When selecting a view, the Attributes window includes controls for **1** size ratio, **2** deleting constraints, **3** height/width mode, **4** margins, and **5** constraint bias. You can also highlight individual constraints in the Layout Editor by clicking on them in the **6** constraint list.

You can change the way the height and width are calculated by clicking the symbols indicated with callout 3



Fixed: You specify a specific dimension in the text box below or by resizing the view in the editor.

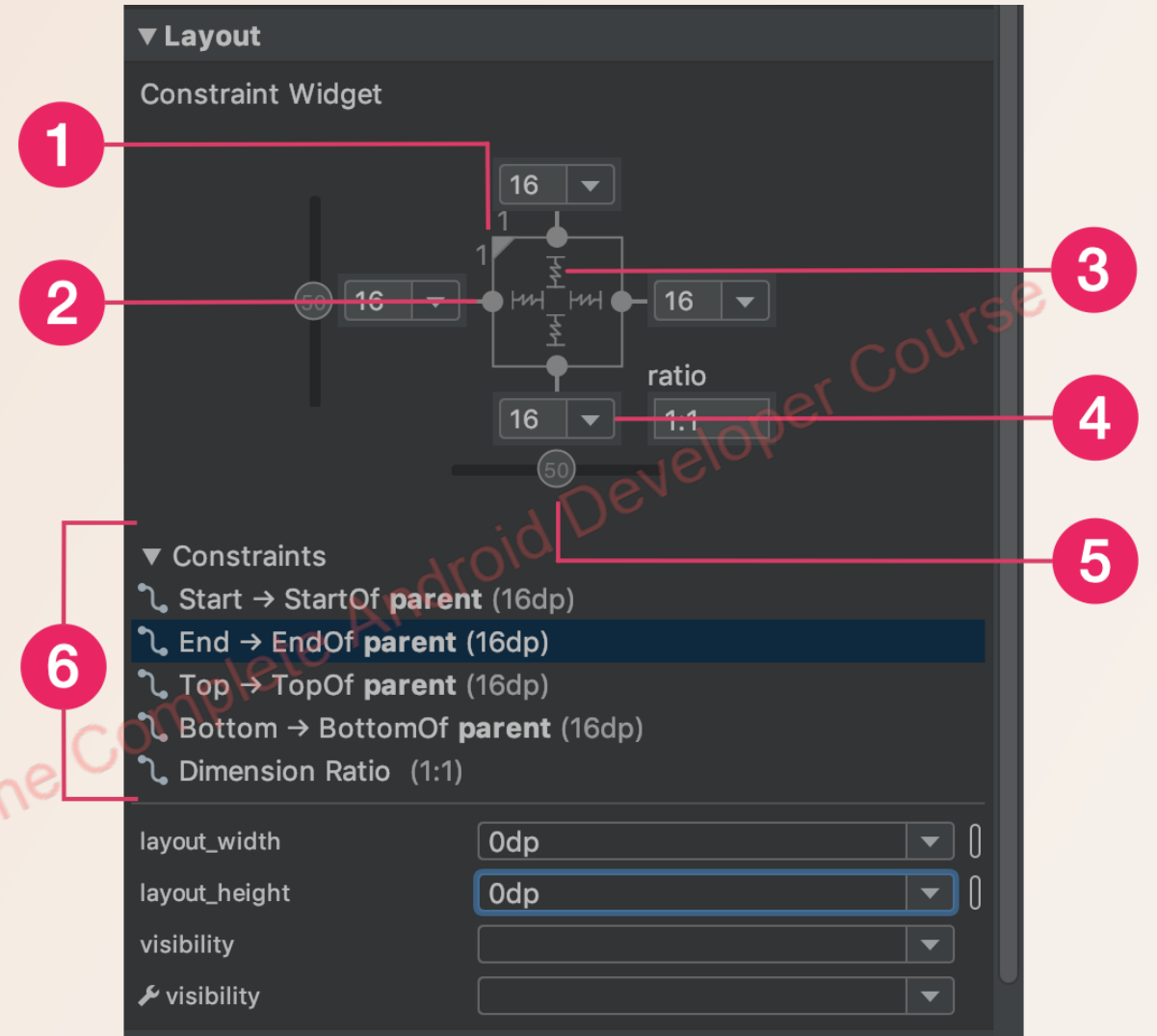


Wrap Content: The view expands only as much as needed to fit its contents

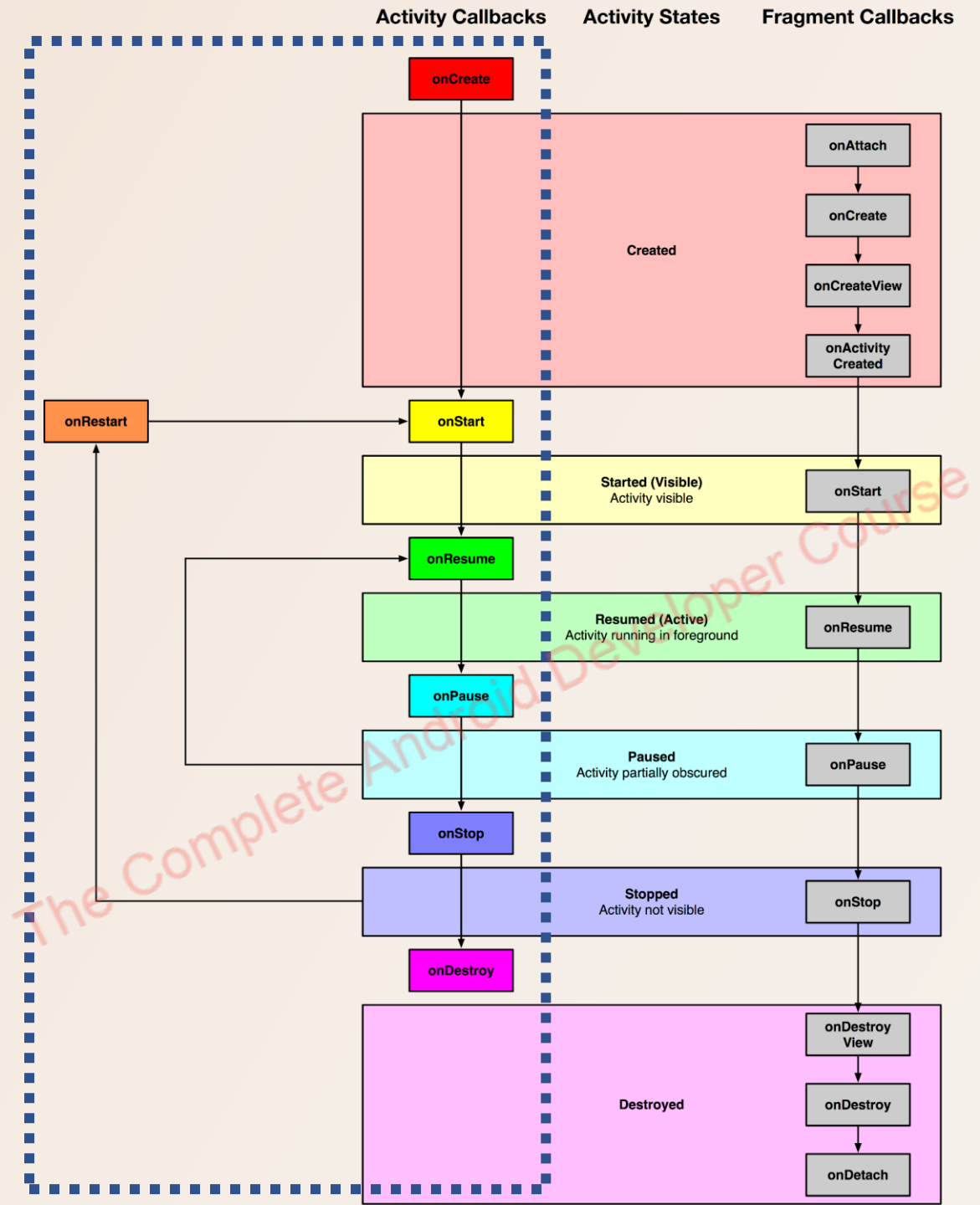


Match Constraints: The view expands as much as possible to meet the constraints on each side (after accounting for the view's margins).

However, you can modify that behavior with the following attributes and values



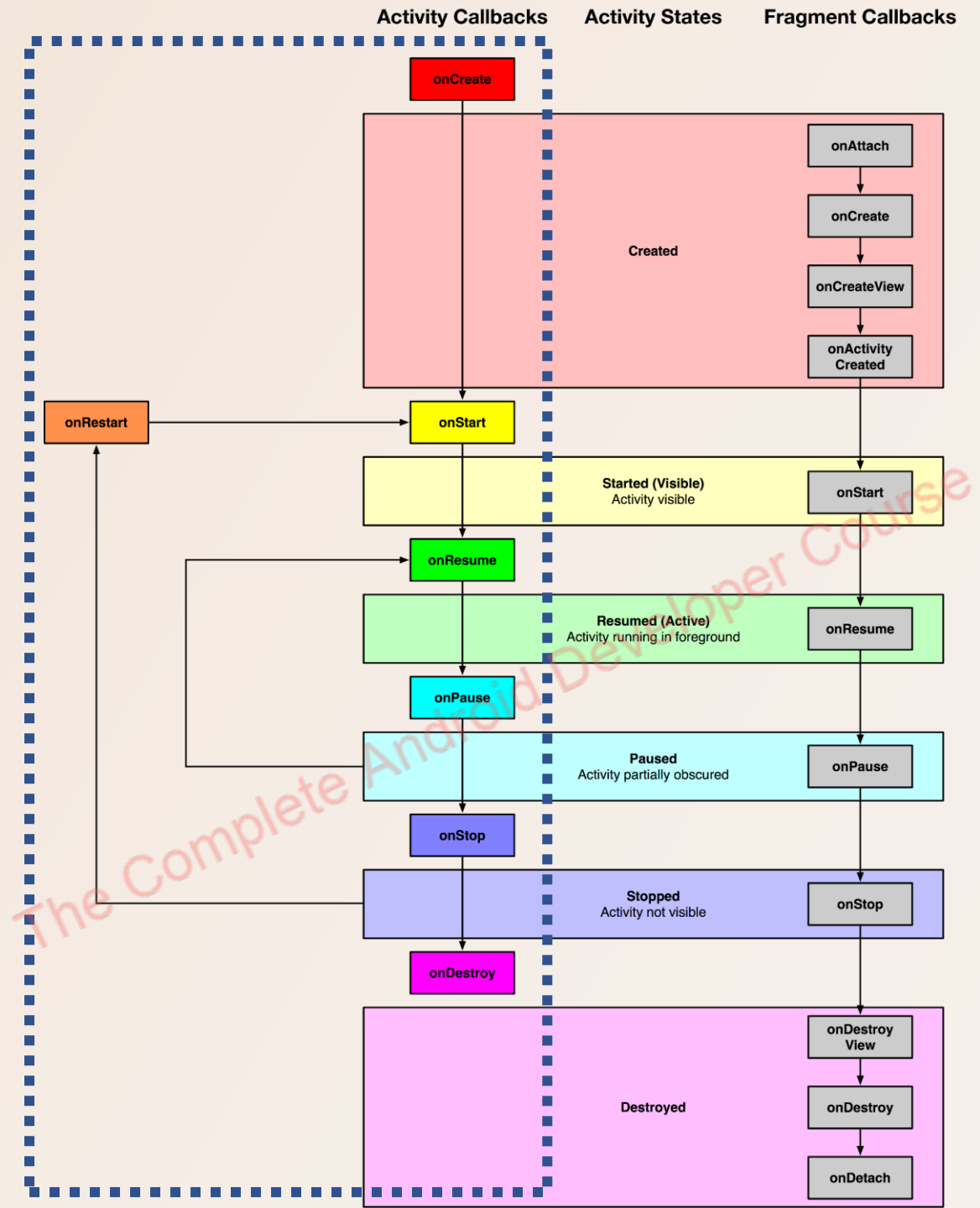
Activity Life Cycle



onCreate()

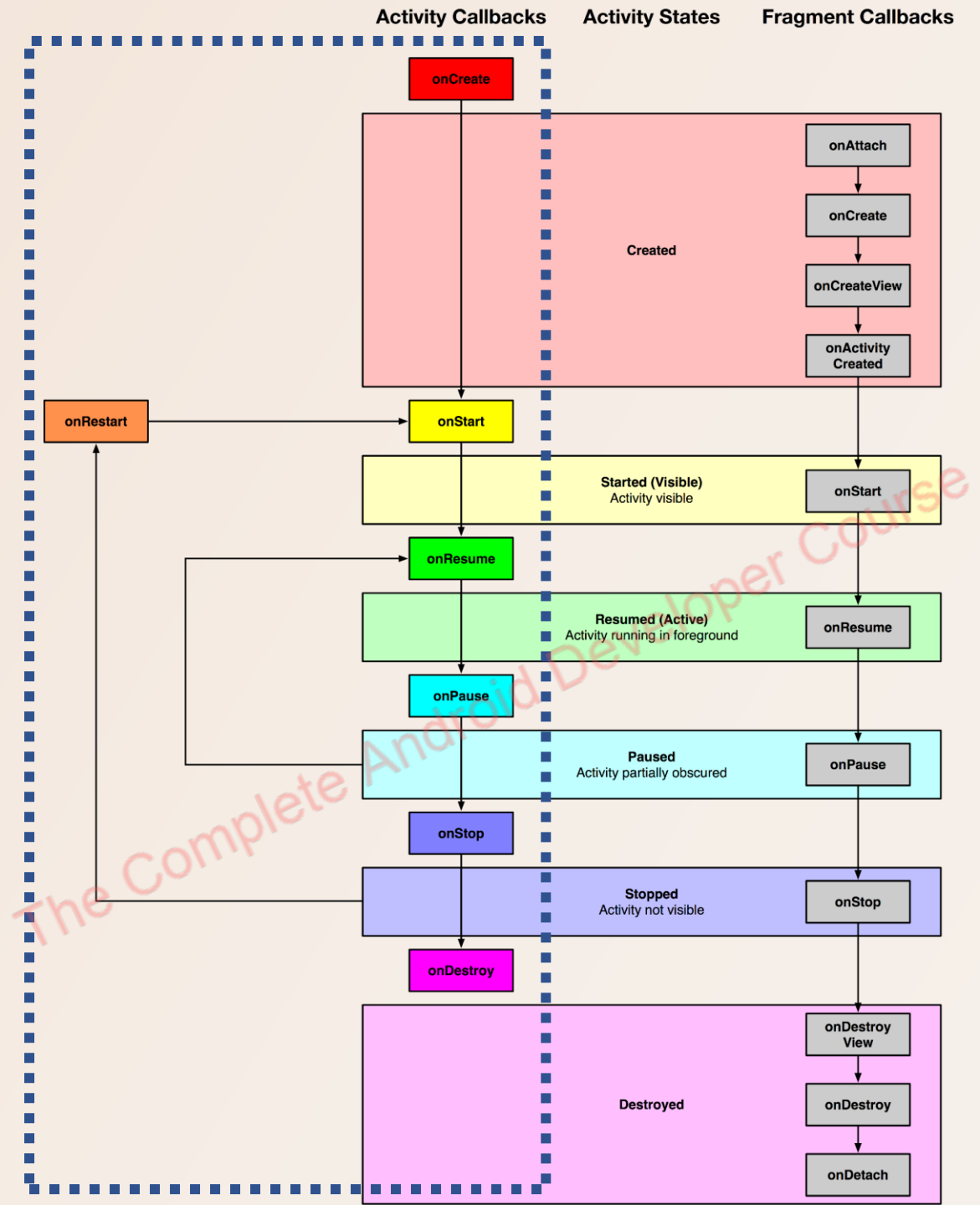
The method that is called when the activity is first created and the ideal location for most initialization tasks to be performed.

The method is passed an argument in the form of a *Bundle* object that may contain dynamic state information (typically relating to the state of the user interface) from a prior invocation of the activity.



onRestart()

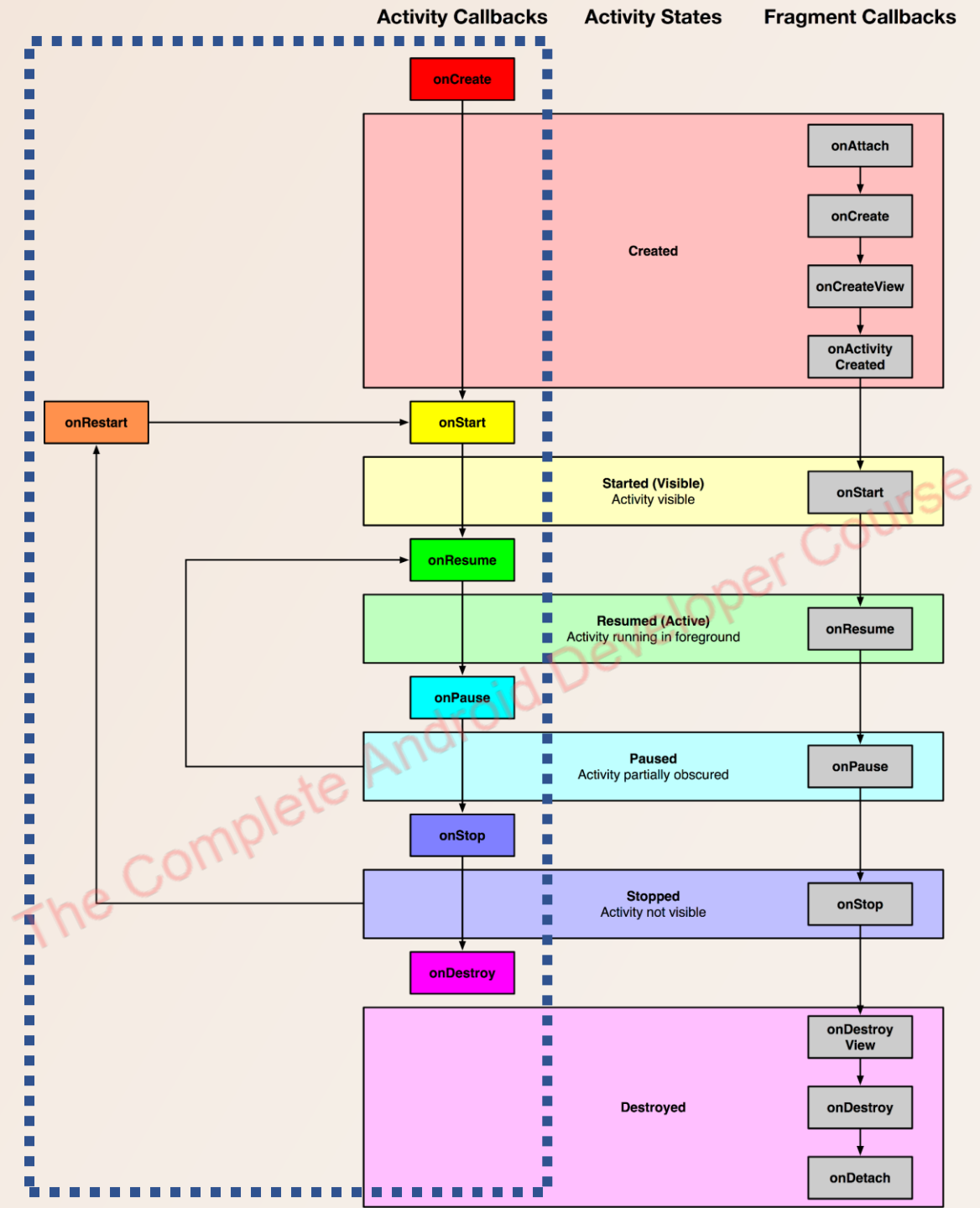
onRestart() – Called when the activity is about to restart after having previously been stopped by the runtime system.



onStart()

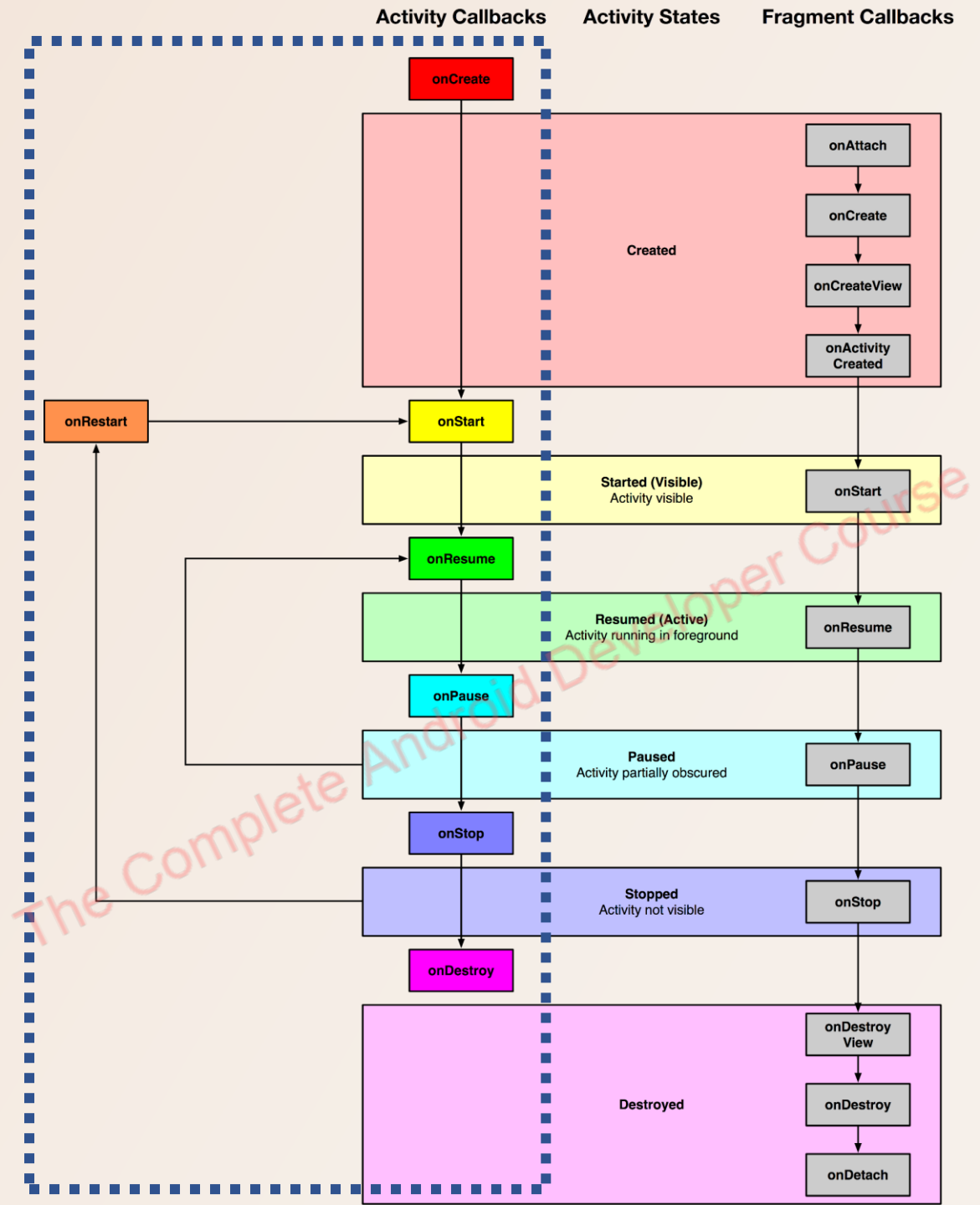
Always called immediately after the call to the `onCreate()` or `onRestart()` methods, this method indicates to the activity that it is about to become visible to the user.

This call will be followed by a call to `onResume()` if the activity moves to the top of the activity stack, or `onStop()` in the event that it is pushed down the stack by another activity.



onResume()

Indicates that the activity is now at the top of the activity stack and is the activity with which the user is currently interacting.



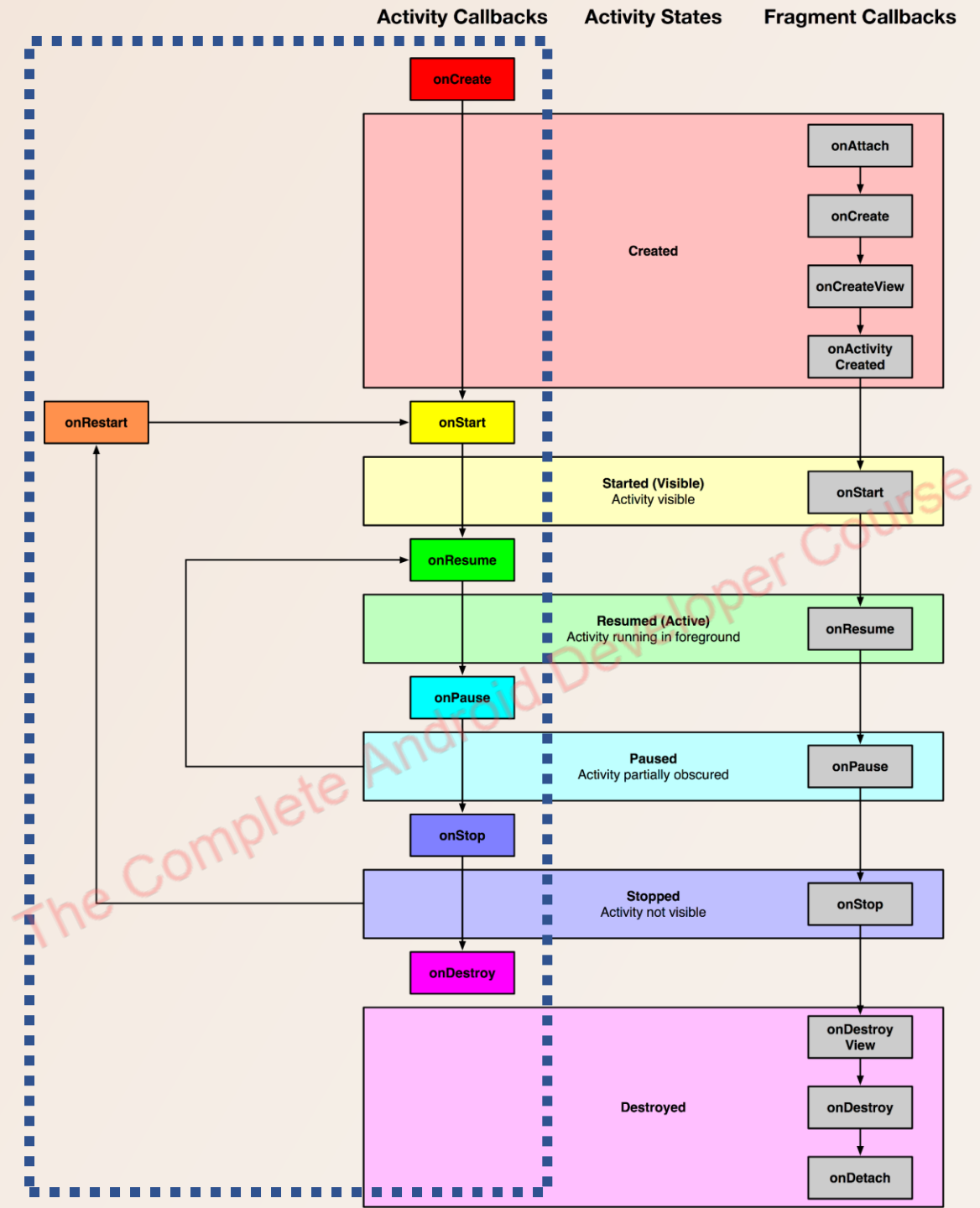
onPause()

Indicates that a previous activity is about to become the foreground activity. This call will be followed by a call to either the `onResume()` or `onStop()` method depending on whether the activity moves back to the foreground or becomes invisible to the user.

Steps may be taken within this method to store persistent state information not yet saved by the app.

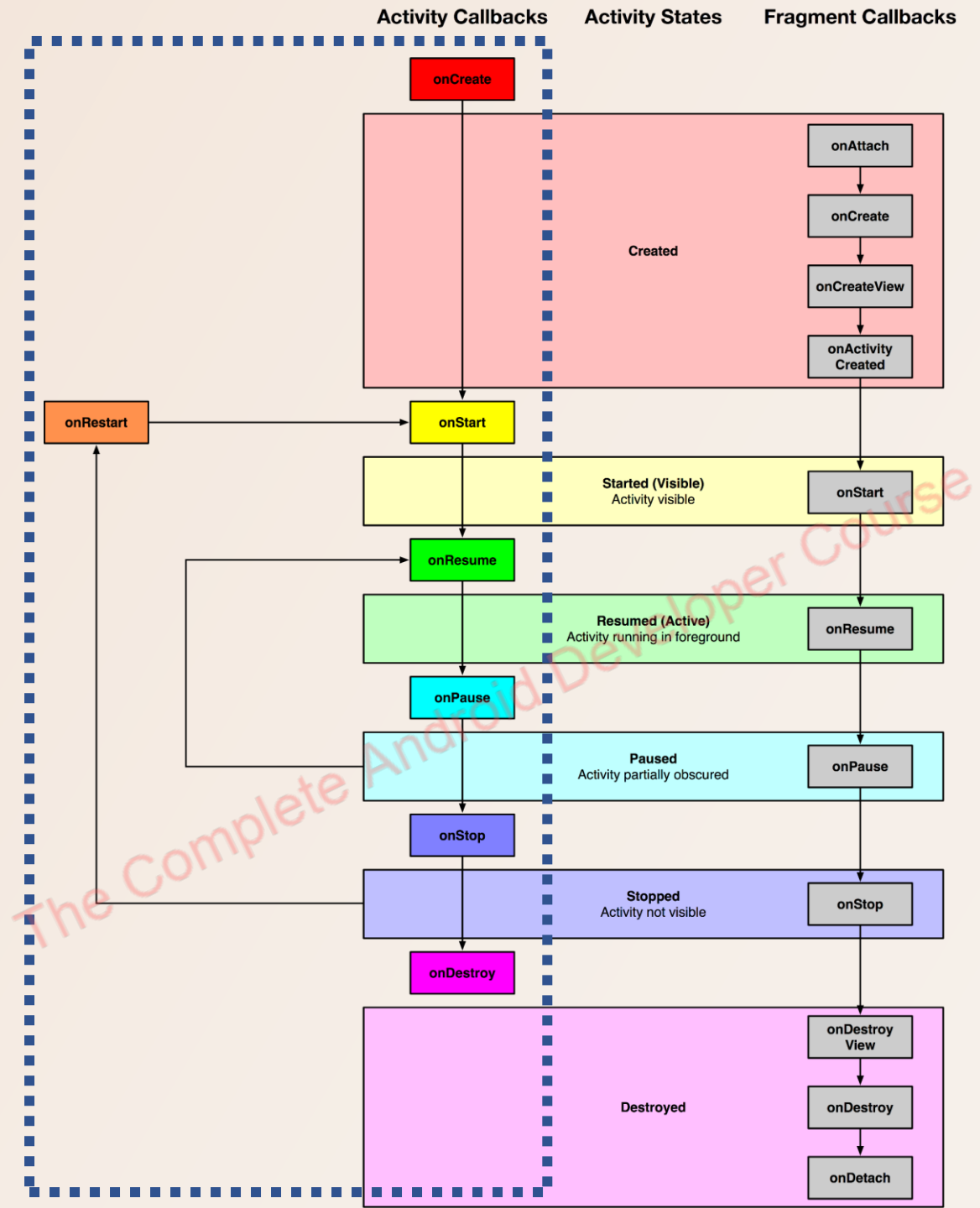
To avoid delays in switching between activities, time consuming operations such as storing data to a database or performing network operations should be avoided within this method.

This method should also ensure that any CPU intensive tasks such as animation are stopped.



onStop()

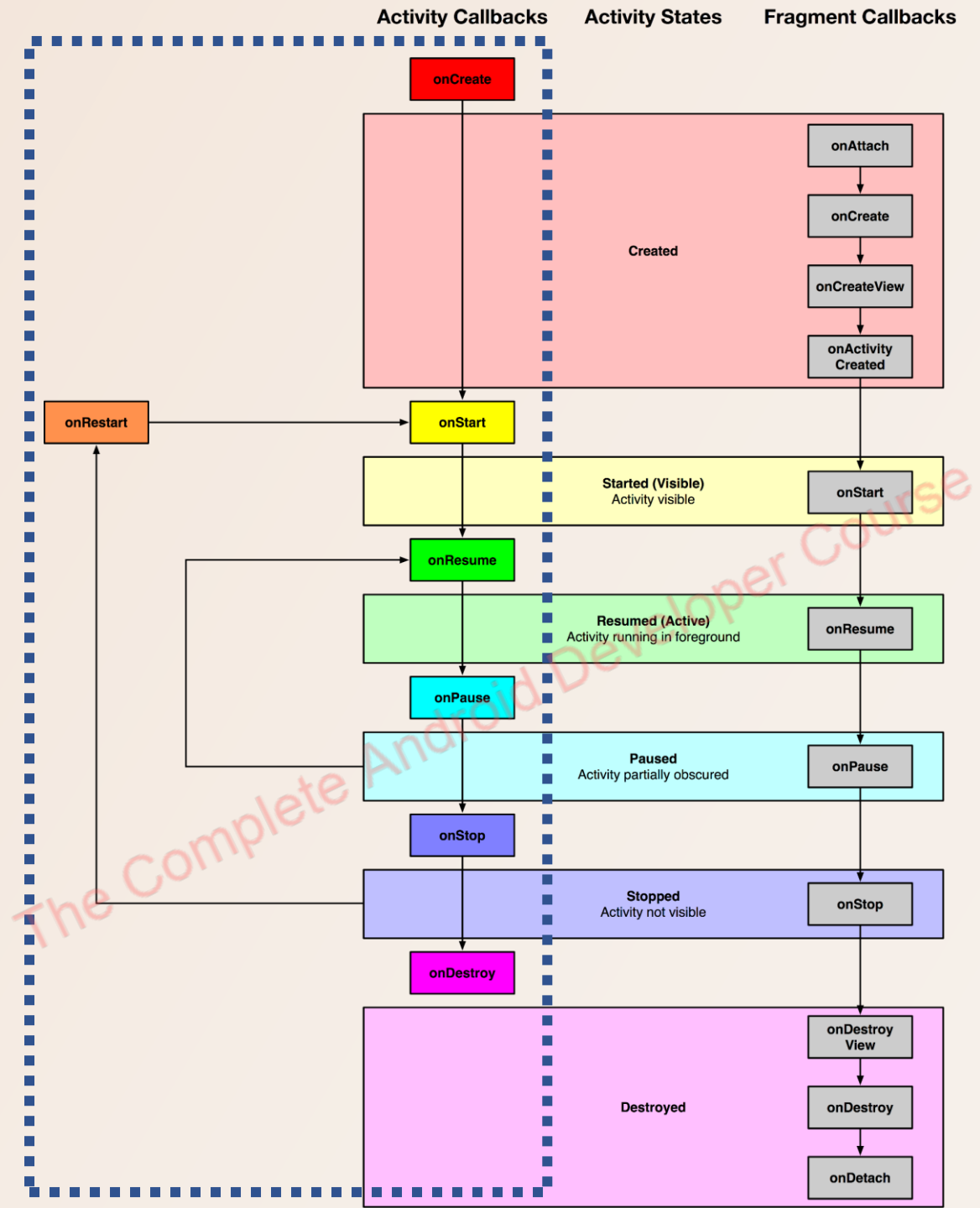
The activity is now no longer visible to the user. The two possible scenarios that may follow this call are a call to `onRestart()` in the event that the activity moves to the foreground again, or `onDestroy()` if the activity is being terminated.

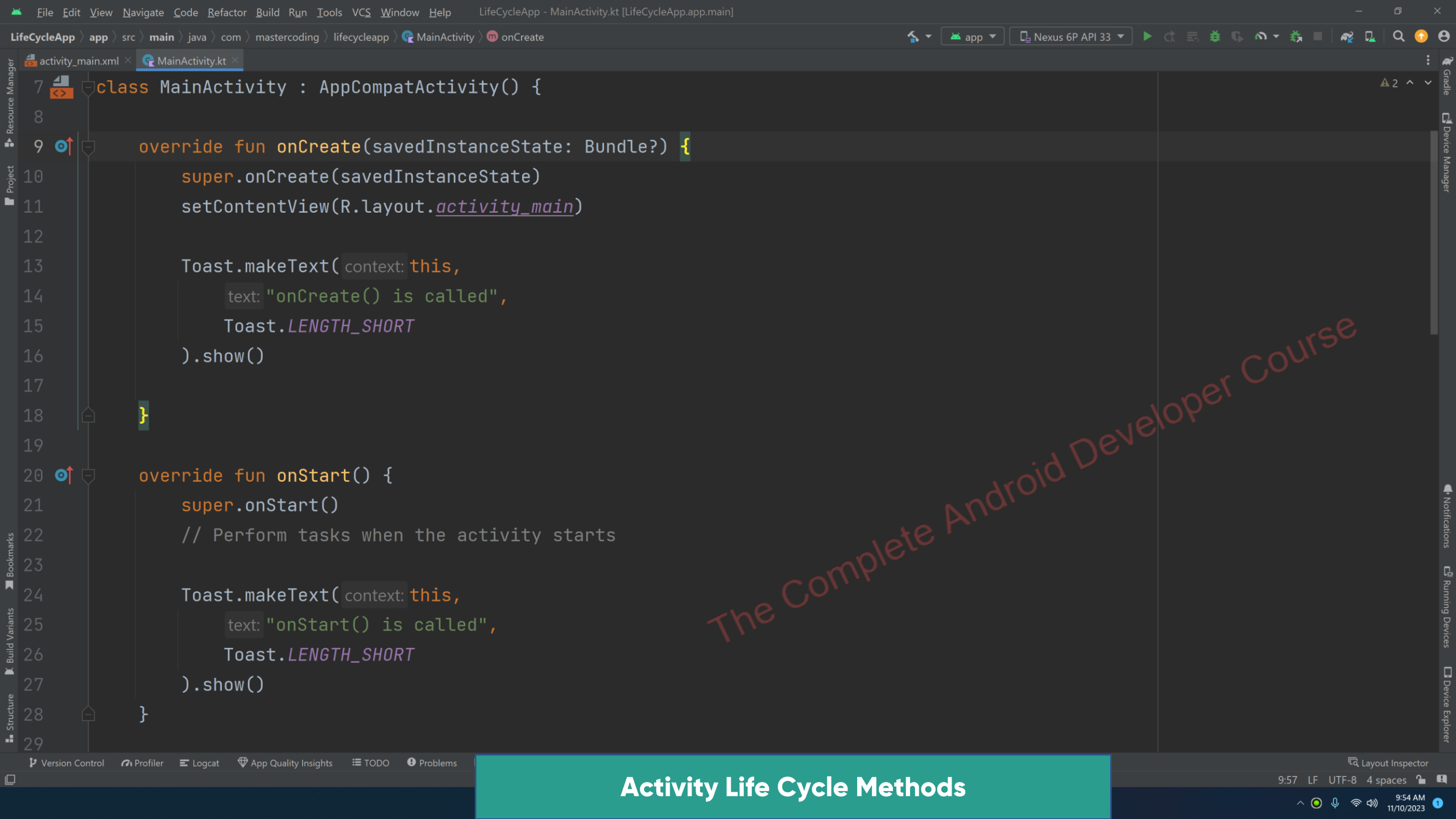


onDestroy()

The activity is about to be destroyed, either voluntarily because the activity has completed its tasks and has called the finish() method or because the runtime is terminating it either to release memory or due to a configuration change (such as the orientation of the device changing).

It is important to note that a call will not always be made to onDestroy() when an activity is terminated.





Activity Life Cycle Methods

LifeCycleApp > app > src > main > java > com > mastercoding > lifecycleapp > MainActivity

activity_main.xml x MainActivity.kt x

30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54

```
override fun onResume() {  
    super.onResume()  
    // Handle activities when the app is in the foreground  
    Toast.makeText(context: this,  
        text: "onResume() is called",  
        Toast.LENGTH_SHORT  
    ).show()  
}  
  
override fun onPause() {  
    super.onPause()  
    // pause or release resources when the activity loses focus  
  
    Toast.makeText(context: this,  
        text: "onPause() is called",  
        Toast.LENGTH_SHORT  
    ).show()  
}  
  
override fun onStop() {  
    super.onStop()  
    // release resources that should be closed when activity is not active  
}
```

Resource Manager
Project
Bookmarks
Build Variants
Structure

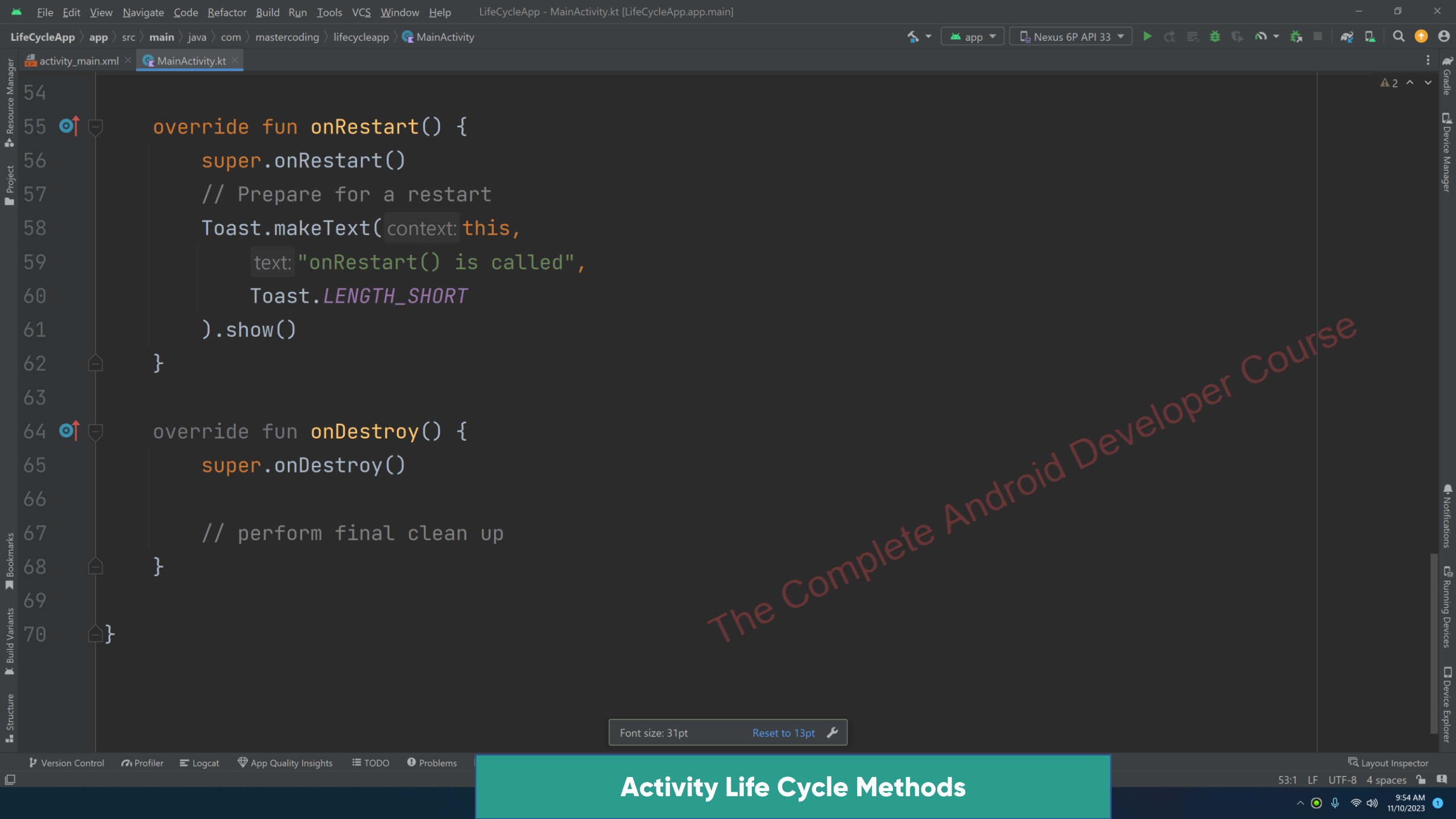
Gradle
Device Manager
Notifications
Running Devices
Device Explorer

Version Control
Profiler
Logcat
App Quality Insights
TODO
Problems

Layout Inspector
53:1 LF UTF-8 4 spaces
9:54 AM
11/10/2023

The Complete Android Developer Course

Activity Life Cycle Methods



LifeCycleApp > app > src > main > java > com > mastercoding > lifecycleapp > MainActivity

activity_main.xml x MainActivity.kt x

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

```
override fun onRestart() {  
    super.onRestart()  
    // Prepare for a restart  
    Toast.makeText(context: this,  
        text: "onRestart() is called",  
        Toast.LENGTH_SHORT  
    ).show()  
}
```

```
override fun onDestroy() {  
    super.onDestroy()  
  
    // perform final clean up  
}
```

Font size: 31pt

Reset to 13pt

Activity Life Cycle Methods

Layout Inspector

53:1 LF UTF-8 4 spaces

9:54 AM 11/10/2023

FileEditViewNavigateCodeRefactorBuildRunToolsVCSWindowHelp

LifeCycleApp - activity_second.xml [LifeCycleApp.app.main]

LifeCycleApp > app > src > main > res > layout > activity_second.xml

MainActivity.kt

SecondActivity.kt

activity_main.xml

activity_second.xml

CodeSplitDesign

CodeSplitDesign

CodeSplitDesign

CodeSplitDesign

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

```
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowCompat

class MainActivity : AppCompatActivity() {

    lateinit var myButton: Button
    lateinit var openWeb: Button

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

        // Intents: used to navigate from one component to another

        // Types of Intents:
        // 1- Explicit: they explicitly specify the target component
        // 2- Implicit

        myButton = findViewById(R.id.goToNextBtn)
        myButton.setOnClickListener { it.View()
            // Create an Explicit Intent
            val explicitIntent = Intent(packageContext, this,
                SecondActivity::class.java)

            explicitIntent.putExtra(name="myname", value="Jack")
            startActivity(explicitIntent)
        }

        openWeb = findViewById(R.id.openWebBtn)
        openWeb.setOnClickListener { it.View()

            // Create implicit intent with the action
            // VIEW and a URL
            val implicitIntent = Intent(
                Intent.ACTION_VIEW,
                Uri.parse(uriString="https://www.google.com")
            )

            startActivity(implicitIntent)
        }
    }
}
```

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

```
package com.mastercoding.lifecycleapp

import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowCompat

class SecondActivity : AppCompatActivity() {

    lateinit var titleTextView: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_second)

        titleTextView = findViewById(R.id.myTextView)

        // Receive data from the Intent
        val receivedData = intent.getStringExtra(name="myname")

        // using the received data as needed
        titleTextView.text =
            "Welcome $receivedData to second activity"
    }
}
```

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:app="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/goToNextBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="164dp"
        android:layout_marginBottom="163dp"
        android:text="Go to Second Activity"
        app:layout_constraintBottom_toBottomOf="@+id/textView"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/openWebBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="85dp"
        android:text="Go to Google"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView" />

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

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:app="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SecondActivity">

    <TextView
        android:id="@+id/myTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to Second Activity"
        android:textSize="32sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

androidx.constraintlayout.widget.ConstraintLayout

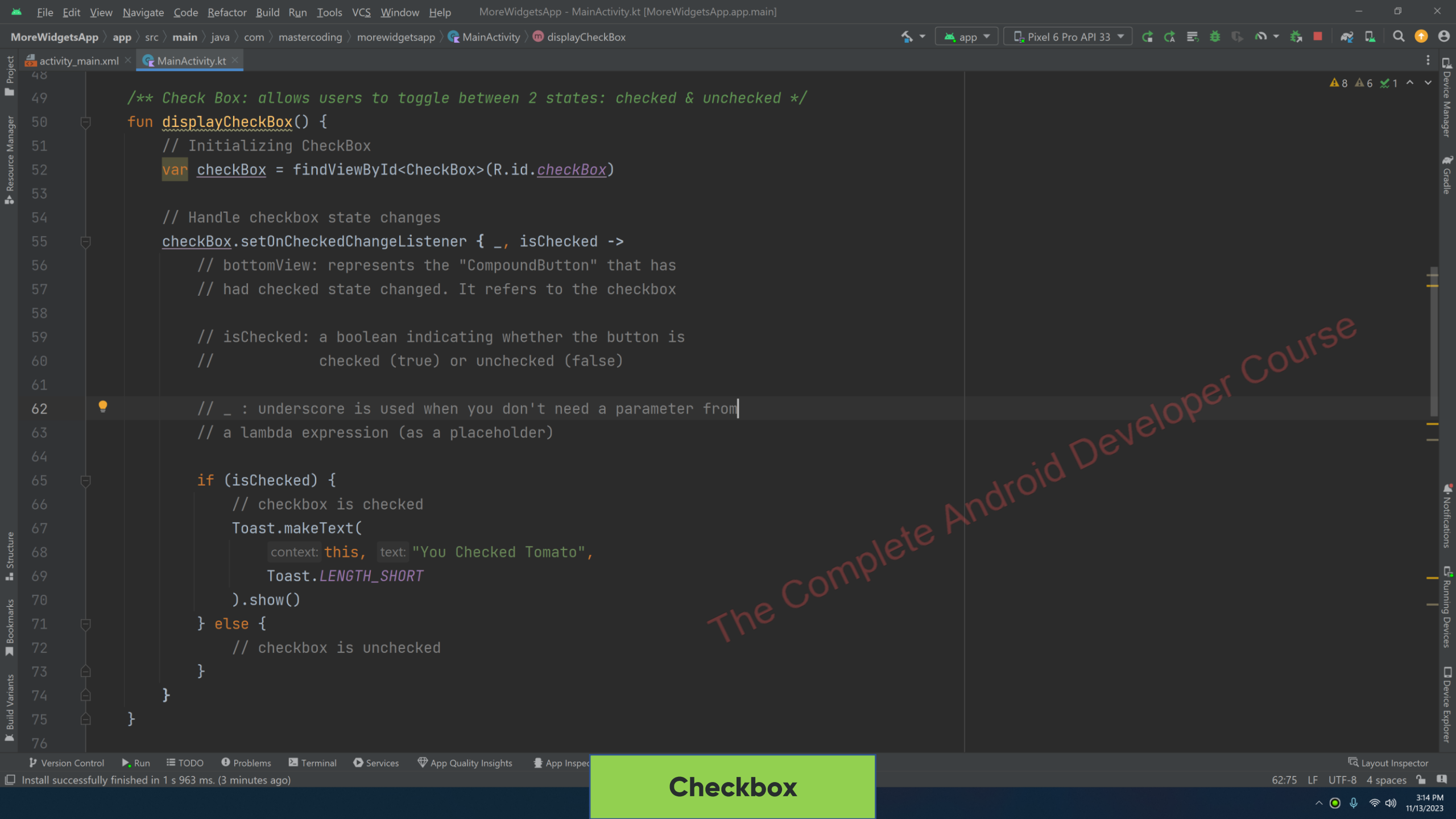
androidx.constraintlayout.widget.ConstraintLayout

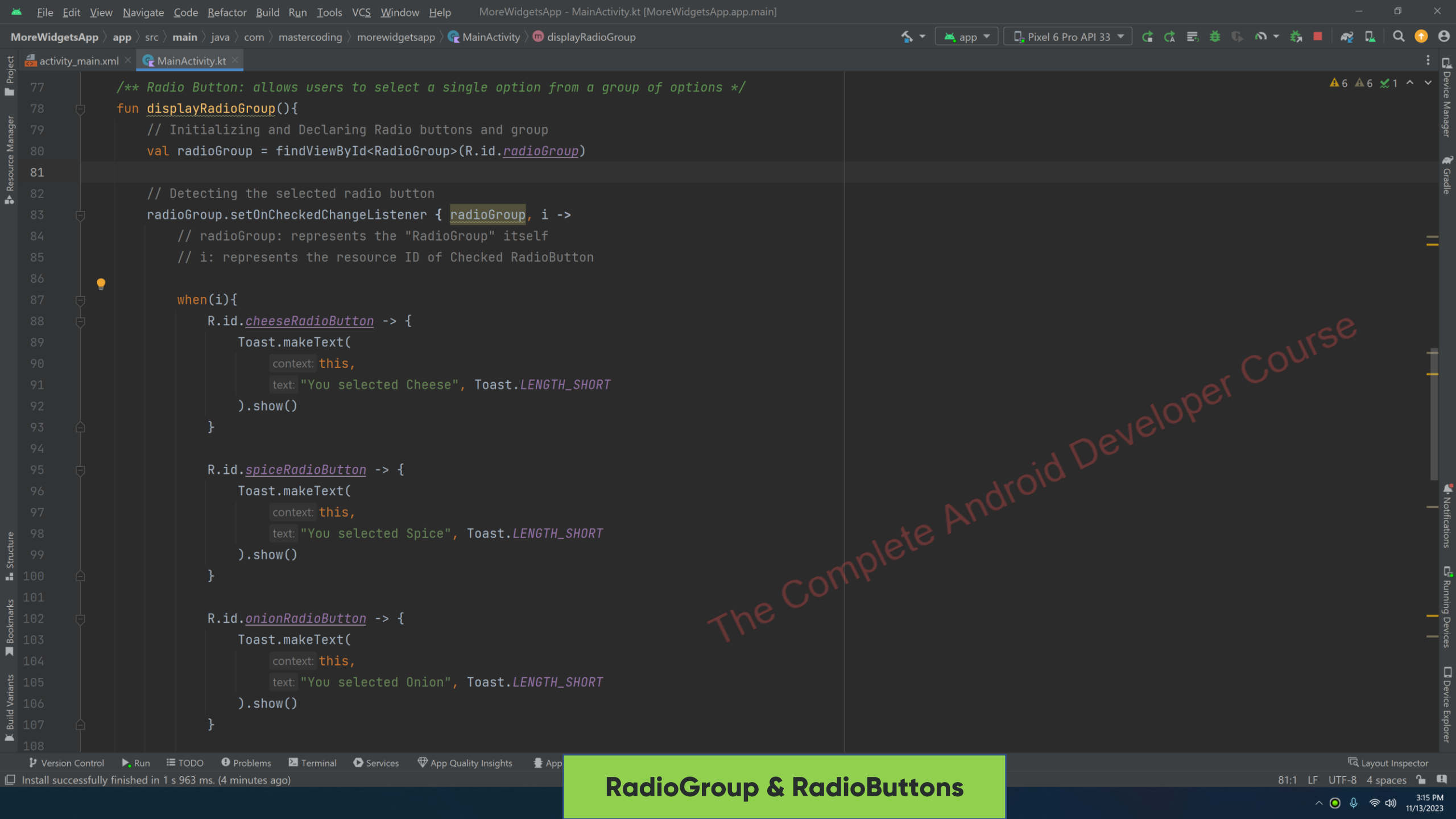
Version ControlRunProfilerLogcatApp Quality InsightsBuildTOD

Install successfully finished in 1 s 496 ms. (5 minutes ago)

The Complete Android Developer Course

20:18 LF UTF-8 4 spaces 6:31 PM 11/10/2023



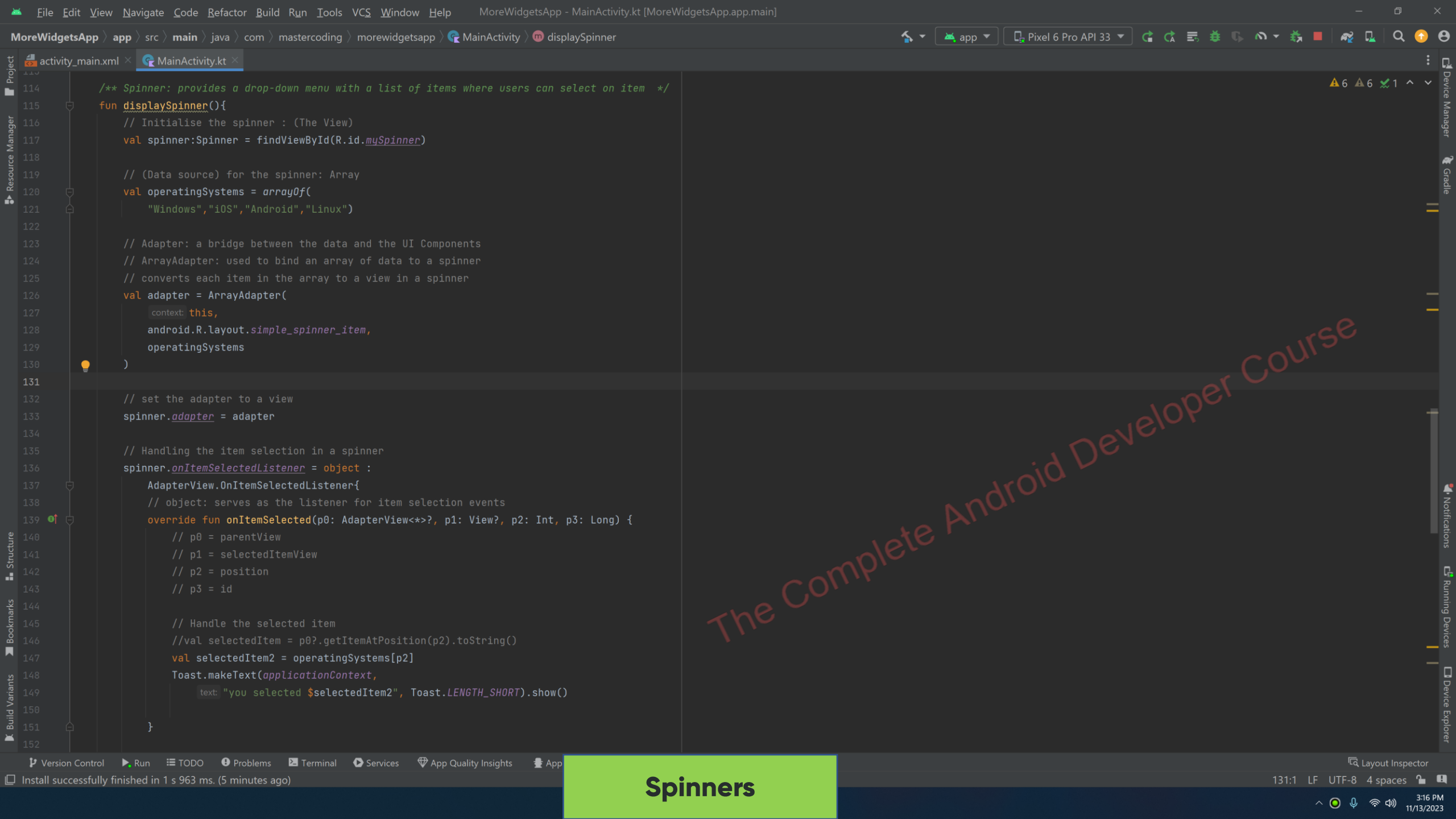


MoreWidgetsApp > app > src > main > java > com > mastercoding > morewidgetsapp > MainActivity > displayRadioGroup

activity_main.xml x MainActivity.kt x

```
77  /** Radio Button: allows users to select a single option from a group of options */
78  fun displayRadioGroup(){
79      // Initializing and Declaring Radio buttons and group
80      val radioGroup = findViewById<RadioGroup>(R.id.radioGroup)
81
82      // Detecting the selected radio button
83      radioGroup.setOnCheckedChangeListener { radioGroup, i ->
84          // radioGroup: represents the "RadioGroup" itself
85          // i: represents the resource ID of Checked RadioButton
86
87          when(i){
88              R.id.cheeseRadioButton -> {
89                  Toast.makeText(
90                      context: this,
91                      text: "You selected Cheese", Toast.LENGTH_SHORT
92                  ).show()
93              }
94
95              R.id.spiceRadioButton -> {
96                  Toast.makeText(
97                      context: this,
98                      text: "You selected Spice", Toast.LENGTH_SHORT
99                  ).show()
100              }
101
102              R.id.onionRadioButton -> {
103                  Toast.makeText(
104                      context: this,
105                      text: "You selected Onion", Toast.LENGTH_SHORT
106                  ).show()
107              }
108          }
109      }
```

RadioGroup & RadioButtons



FileEditViewNavigateCodeRefactorBuildRunToolsVCSWindowHelp

MoreWidgetsApp > app > src > main > java > com > mastercoding > morewidgetsapp > MainActivity > displayTimePicker

Pixel 6 Pro API 33

activity_main.xml x MainActivity.kt x

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

```
/** Time Picker: allows users to select a time of day */
fun displayTimePicker(){
    val timePicker:TimePicker = findViewById(R.id.timePicker)

    // Handle the changes in time
    timePicker.setOnTimeChangeListener { timePicker, hourOfDay, minute ->

        // timePicker = view : represents the TimePicker itself
        // i = hourOfDay : selected hour in 24-hour format
        // i2 = minute : selected minute

        // Formatting the selected time :      hh:mm|
        val selectedTime = String.format("%02d:%02d",hourOfDay,minute)
        Toast.makeText(applicationContext,
            text:"Your time: $selectedTime",
            Toast.LENGTH_SHORT
        ).show()
    }
}
```

1

7

7

1

Project

Resource Manager

Structure

Bookmarks

Build Variants

Device Manager

Gradle

Notifications

Running Devices

Device Explorer

Version Control

Run

TODO

Problems

Terminal

Services

App Quality Insights

App Inspector

Install successfully finished in 1 s 963 ms. (6 minutes ago)

Layout Inspector

175:56 LF UTF-8 4 spaces

3:17 PM 11/13/2023

Time Picker

MoreWidgetsApp > app > src > main > java > com > mastercoding > morewidgetsapp > MainActivity > displayDatePicker

activity_main.xml x MainActivity.kt x

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

```
/** Date Picker: allows users to select a date */
@RequiresApi(Build.VERSION_CODES.O)
fun displayDatePicker(){

    val datePicker = findViewById<DatePicker>(R.id.datePicker)

    // Handling Date Changes
    datePicker.setOnDateChangeListener { datePicker, i, i2, i3 ->
        // datePicker = view : the datePicker itself
        // i = year
        // i2 = monthOfYear
        // i3 = dayOfMonth
        Toast.makeText(
            applicationContext,
            text: "Year: $i , Month: ${i2+1} , Day: $i3",
            Toast.LENGTH_SHORT
        ).show()
    }
}
```

1 error 7 warnings 1 info

Device Manager

Gradle

Notifications

Running Devices

Device Explorer

Version Control Run TODO Problems Terminal Services App Quality Insights Logcat

Install successfully finished in 1 s 963 ms. (6 minutes ago)

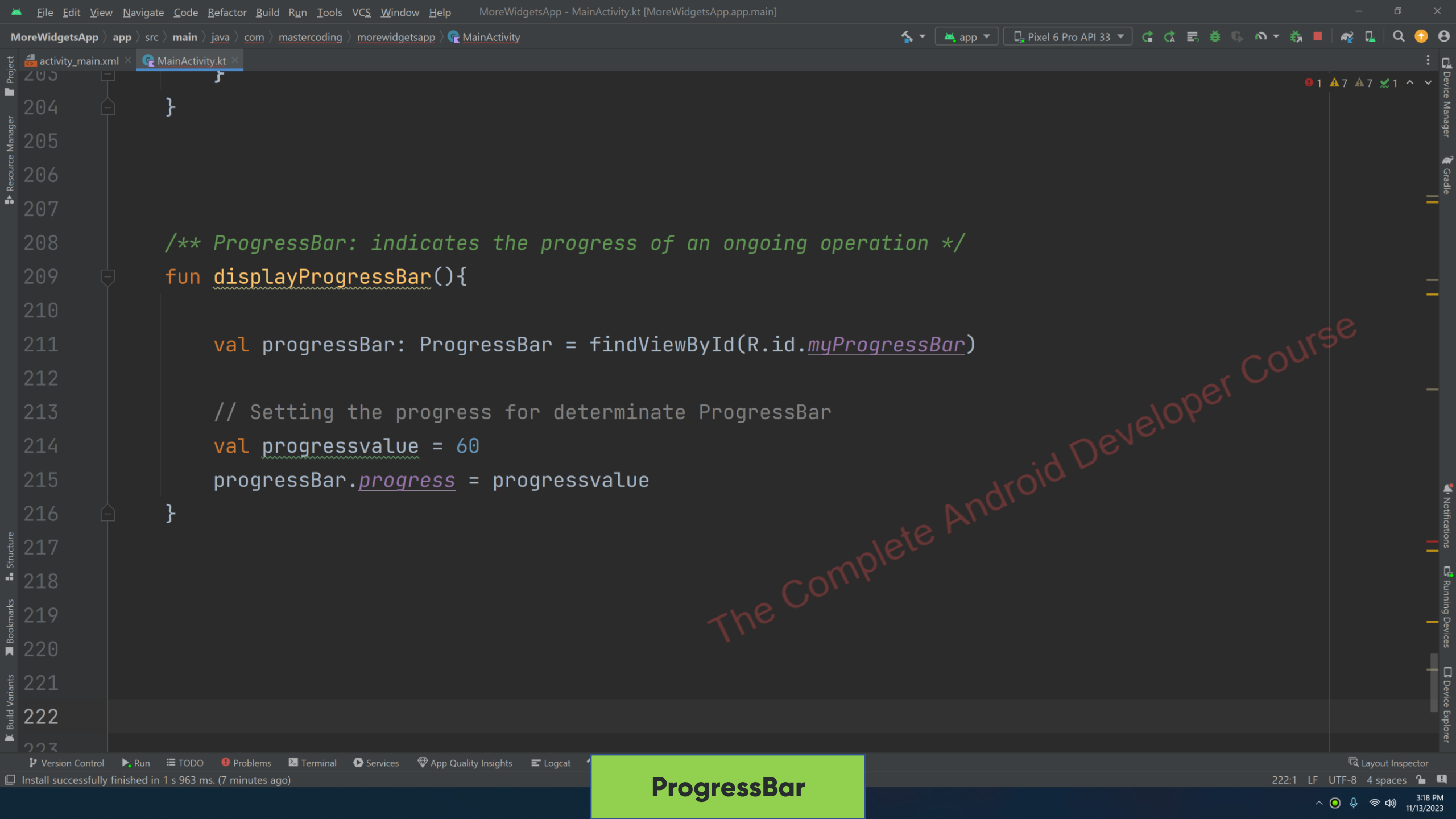
Layout Inspector

204:6 LF UTF-8 4 spaces

3:18 PM 11/13/2023

The Complete Android Developer Course

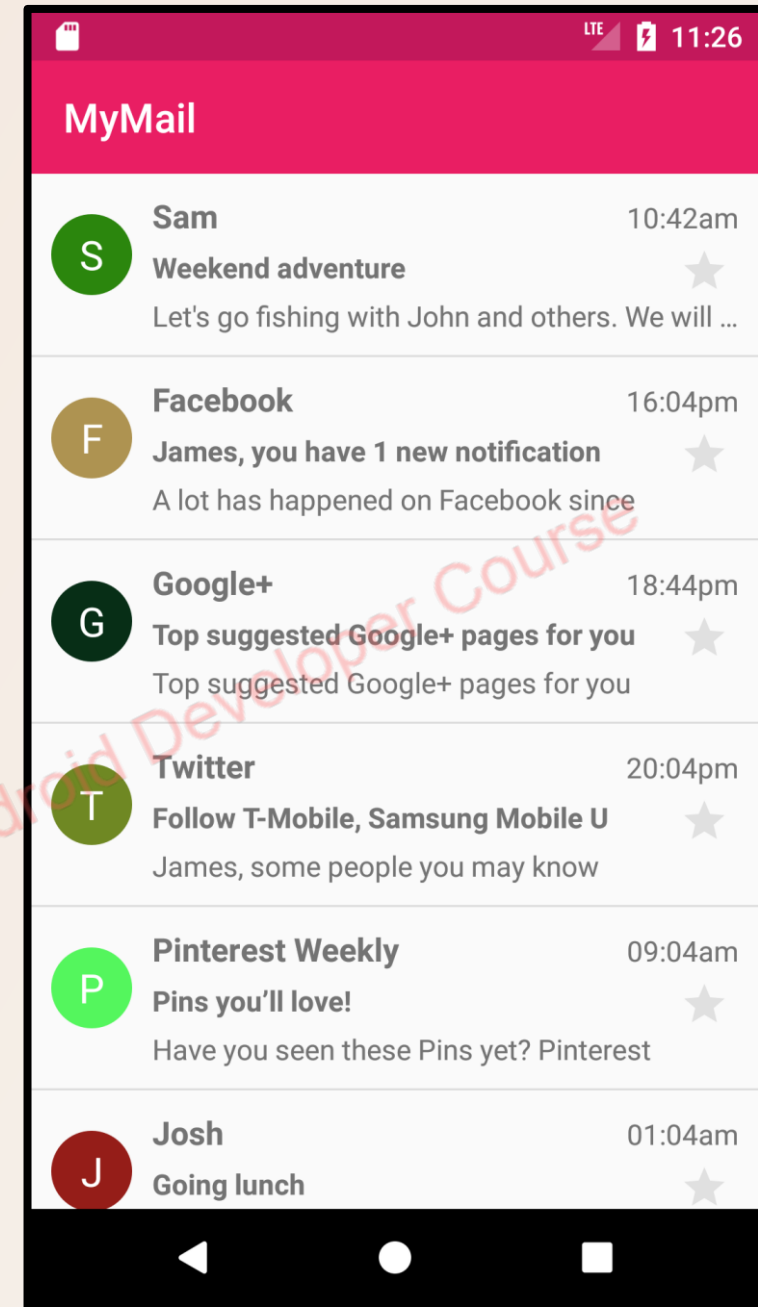
Date Picker



RecyclerView

RecyclerView makes it easy to efficiently display large sets of data.

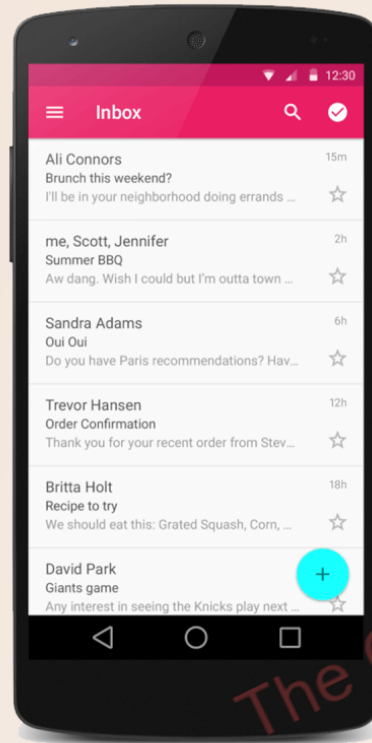
You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they're needed.



RecyclerView

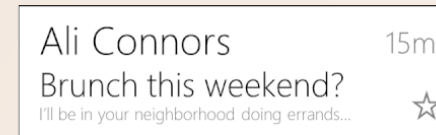
As the name implies, RecyclerView recycles those individual elements. When an item scrolls off the screen, RecyclerView doesn't destroy its view.

Instead, RecyclerView reuses the view for new items that have scrolled onscreen.

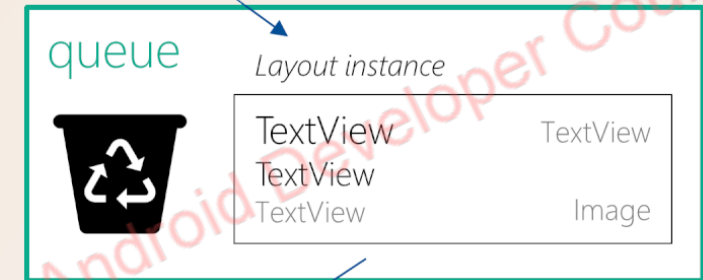


1. Layout instance scrolls out of view

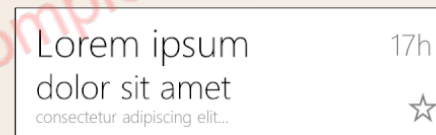
Layout instance



2. Placed in queue



Layout instance

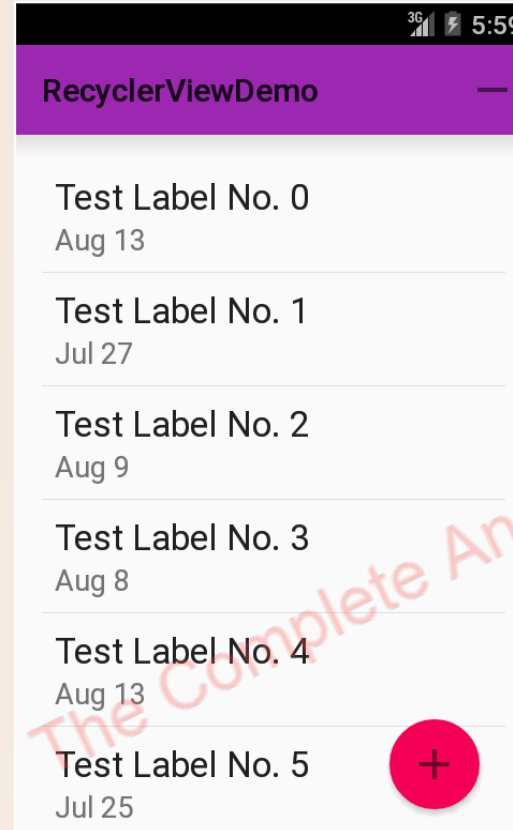


3. Filled with new content & scrolls in again

RecyclerView

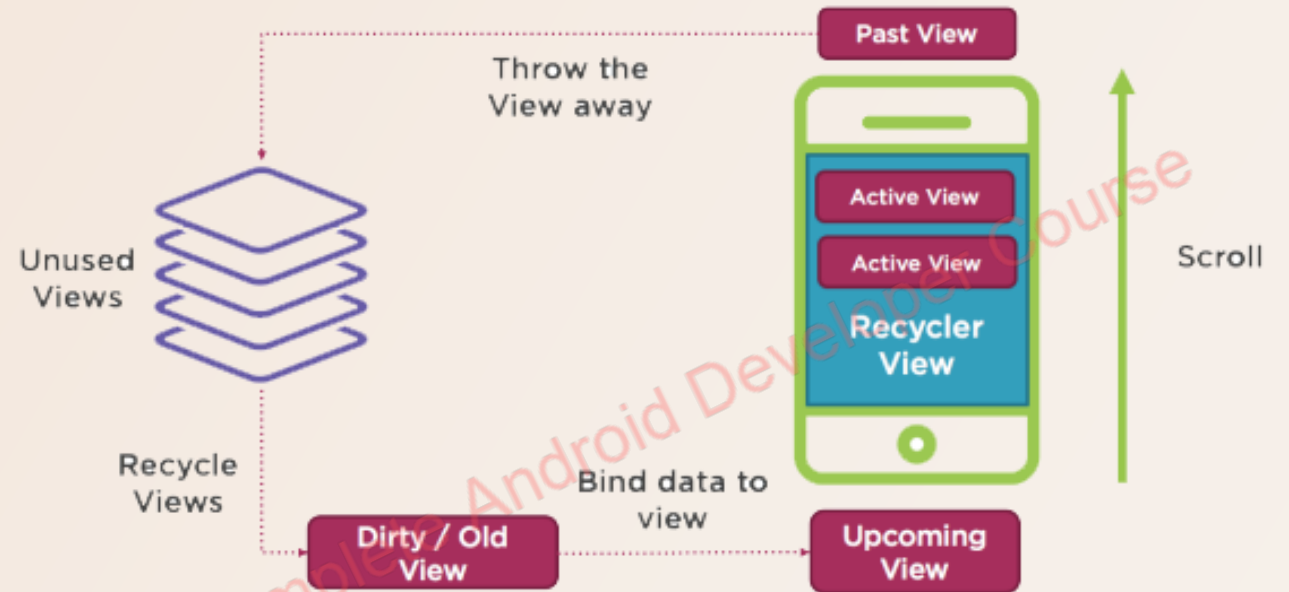
As the name implies, RecyclerView recycles those individual elements. When an item scrolls off the screen, RecyclerView doesn't destroy its view.

Instead, RecyclerView reuses the view for new items that have scrolled onscreen.



RecyclerView

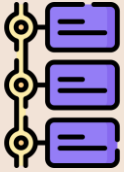
RecyclerView improves performance and your app's responsiveness, and it reduces power consumption.



Creating RecyclerView



1- The Item Layout: an xml layout representing the layout of a single item in the recyclerview.



2- RecyclerView: creating a recyclerview in the activity and initializing it.



3- Model Class: representing the data class that acts as a structure for holding the information for every item of recyclerview.



4- Adapter Class: Holding all methods dealing with recyclerview implementation. (creating, binding and determining the count of items).

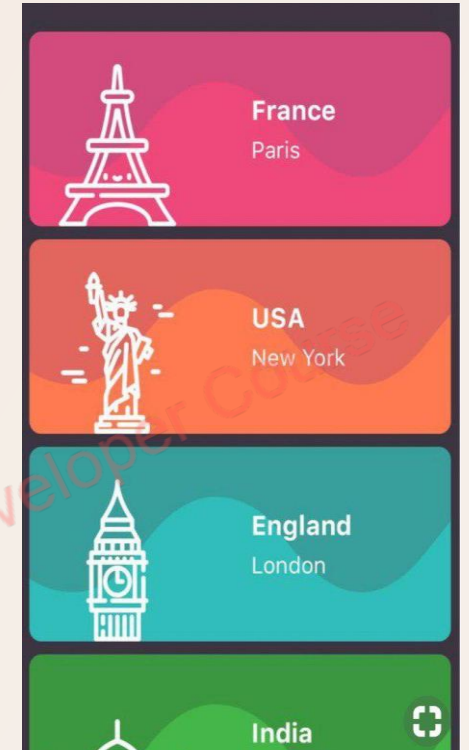
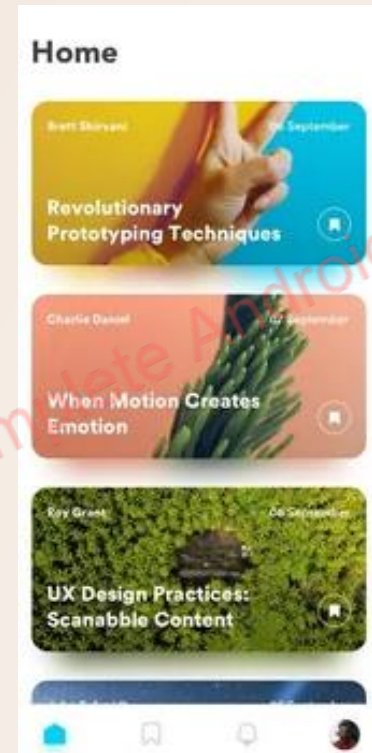
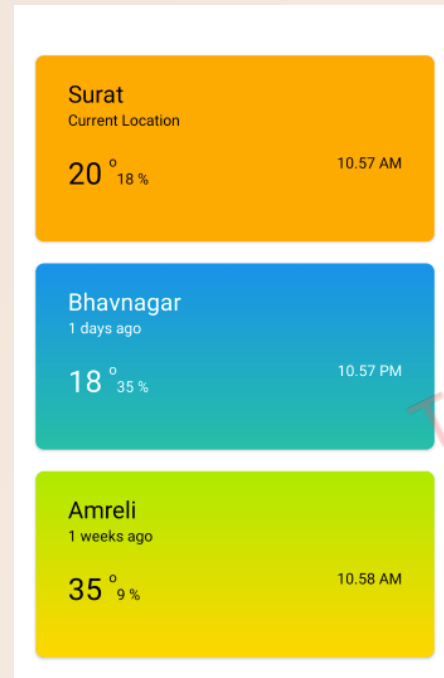


5- View Holder: Holds the references to the views within each item's layout. Optimizing view lookups.

CardView

Used to display any sort of data by providing a rounded corner layout along with a specific elevation.

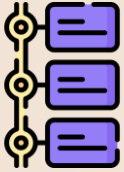
CardView can be used for creating items in listview or inside Recycler View.



Creating CardView App



1- The CardView Item Layout: an xml layout representing the layout of a single item in the recyclerview.



2- RecyclerView: creating a recyclerview in the activity and initializing it.



3- Model Class: representing the data class that acts as a structure for holding the information for every item of recyclerview.

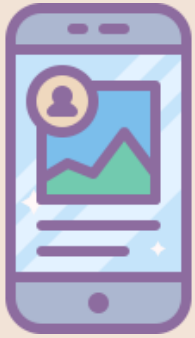


4- Adapter Class: Holding all methods dealing with recyclerview implementation. (creating, binding and determining the count of items).



5- View Holder: Holds the references to the views within each item's layout. Optimizing view lookups.

Android Components



Activity

Represents a single screen with a user interface



Services

Runs in the background to perform long-running operations without a user interface



Broadcast Receivers

Listen for system-wide announcements or intents



Content Providers

Manages access to a structured set of data

The Complete Android Developer's Guide

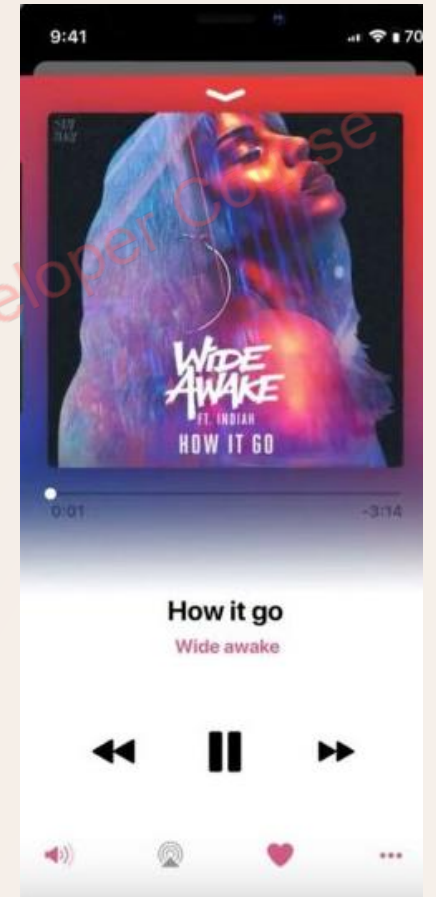
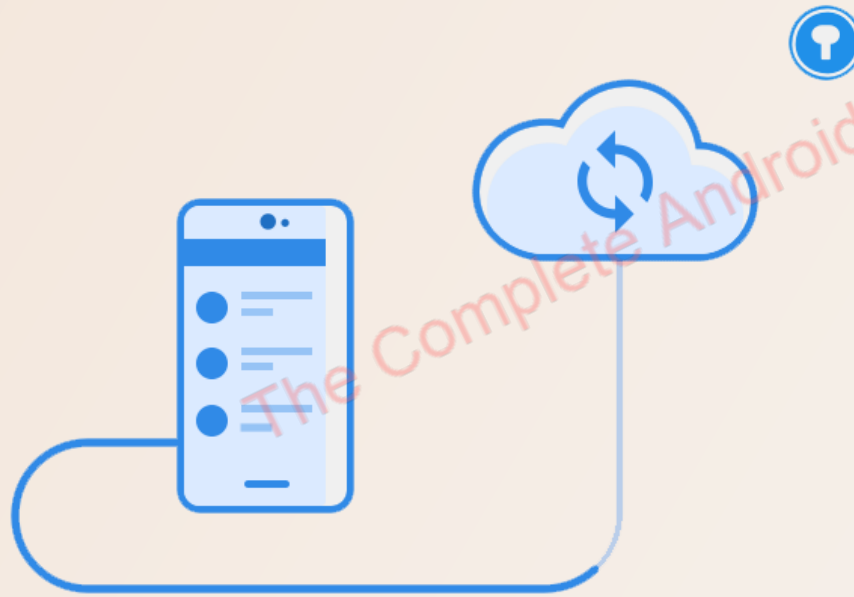
Services



Runs in the background to perform long-running operations without a user interface.

Types of Services:

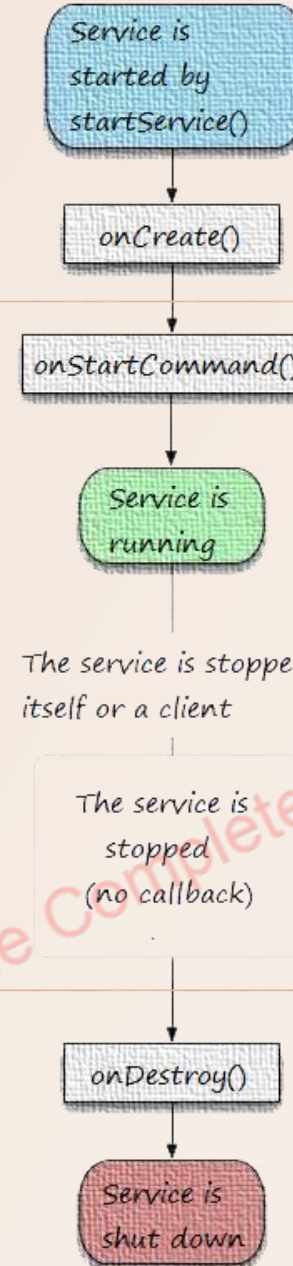
- 1- Foreground
- 2- Background
- 3- Bound Services



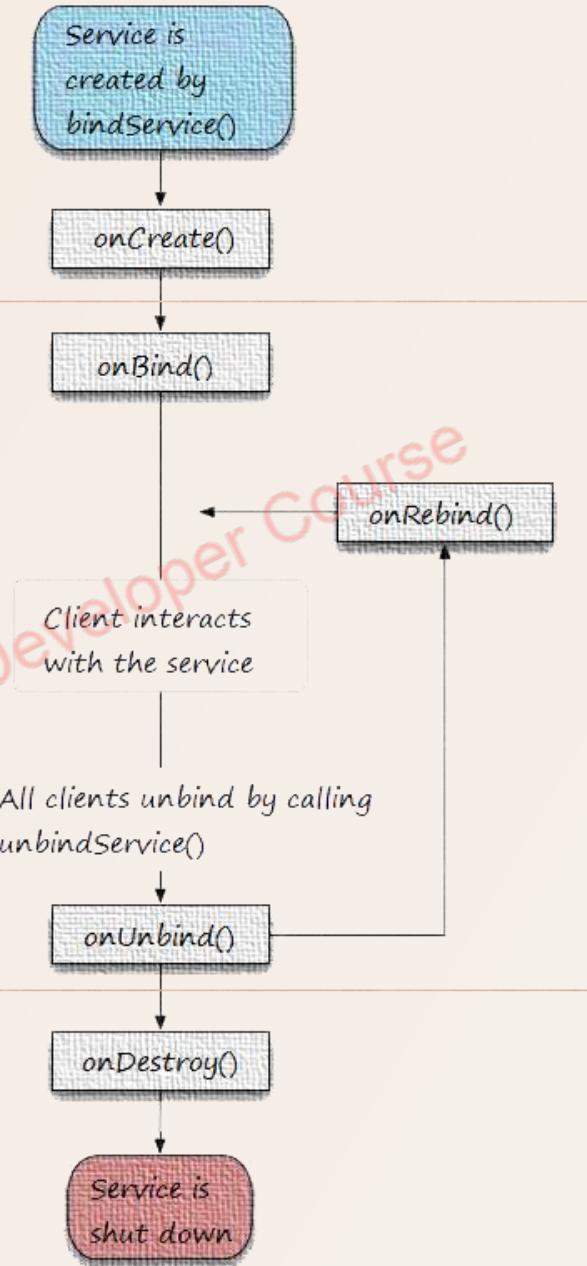
Services Life Cycle



Un Bounded Service



Bounded Service



Broadcast Receivers



Listens for system-wide broadcast events or intents and allows your application to respond to those events.

When any of these events occur, it brings the application into action by either creating a status bar notification or performing a task.

Unlike activities, android BroadcastReceiver doesn't contain any user interface.

The Complete Android Developer Course

Fragments

Fragments in Android are a bit like these LEGO bricks for building your app

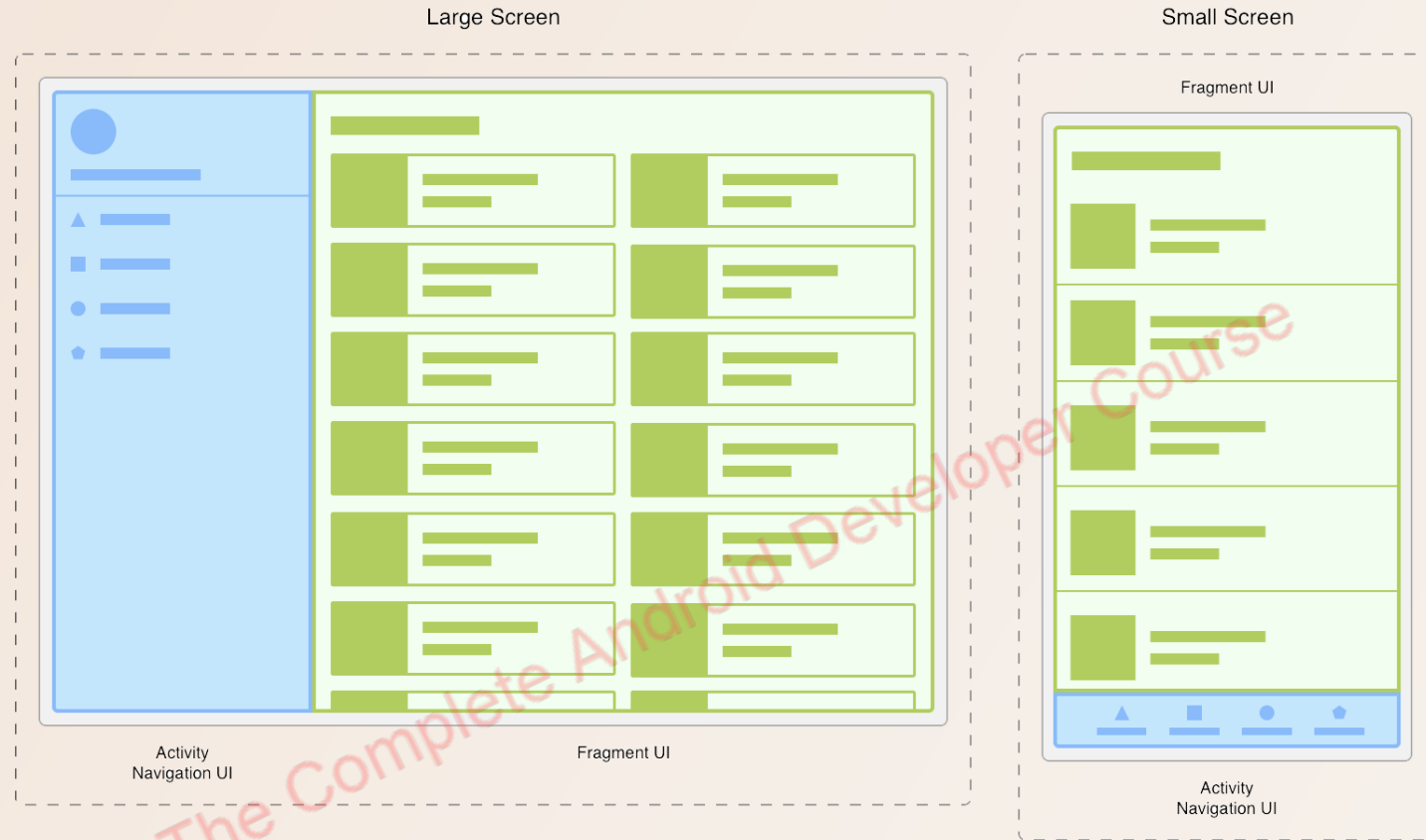


Fragments

A Fragment represents a reusable portion of your app's UI.

A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events.

Fragments can't live on their own. They must be hosted by an activity or another fragment.

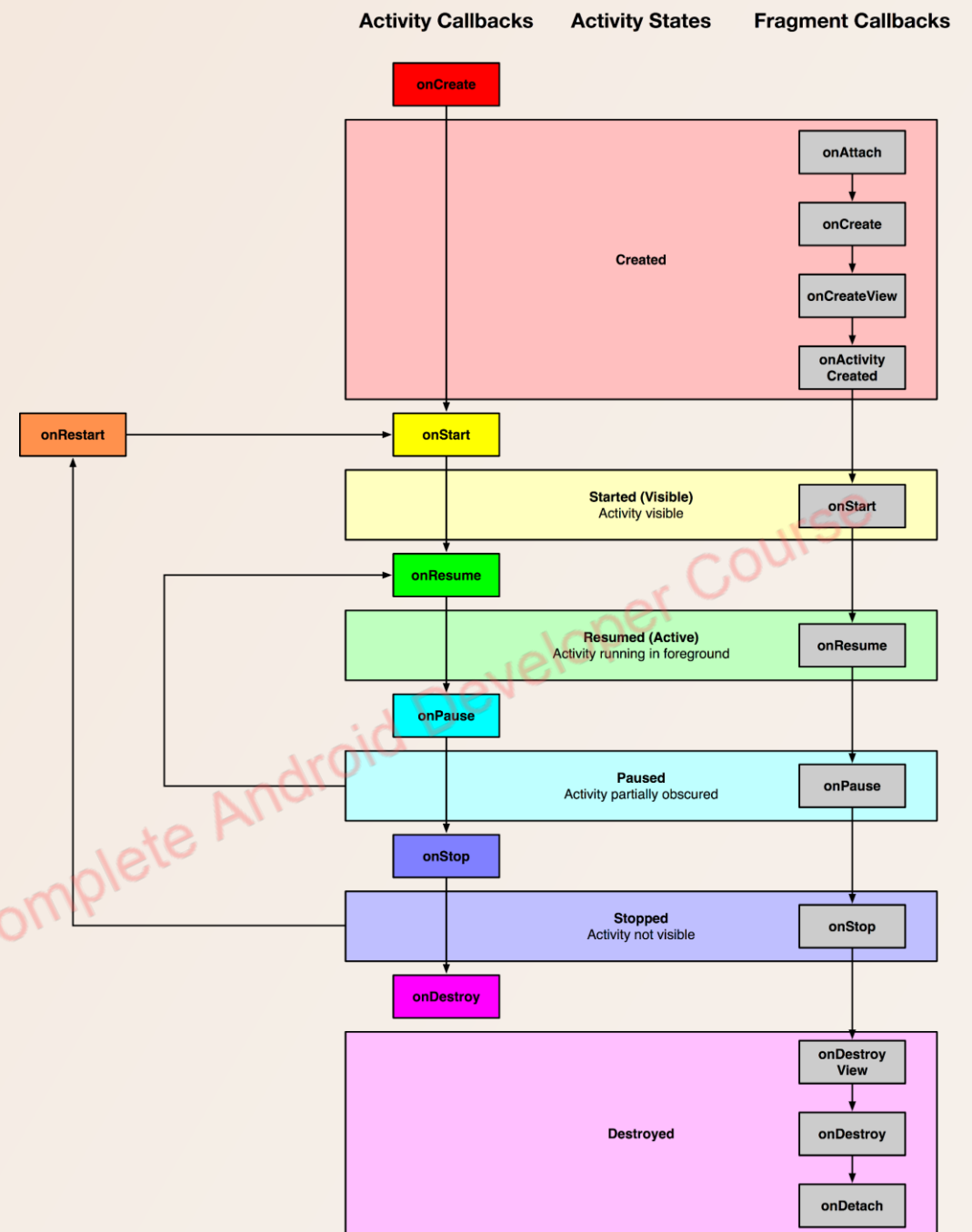


Fragment's Life Cycle

Each Fragment instance has its own lifecycle. When a user navigates and interacts with your app, your fragments transition through various states in their lifecycle as they are added, removed, and enter or exit the screen.

The Fragment class includes callback methods that correspond to each of the changes in a fragment's lifecycle.

These include `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()`.

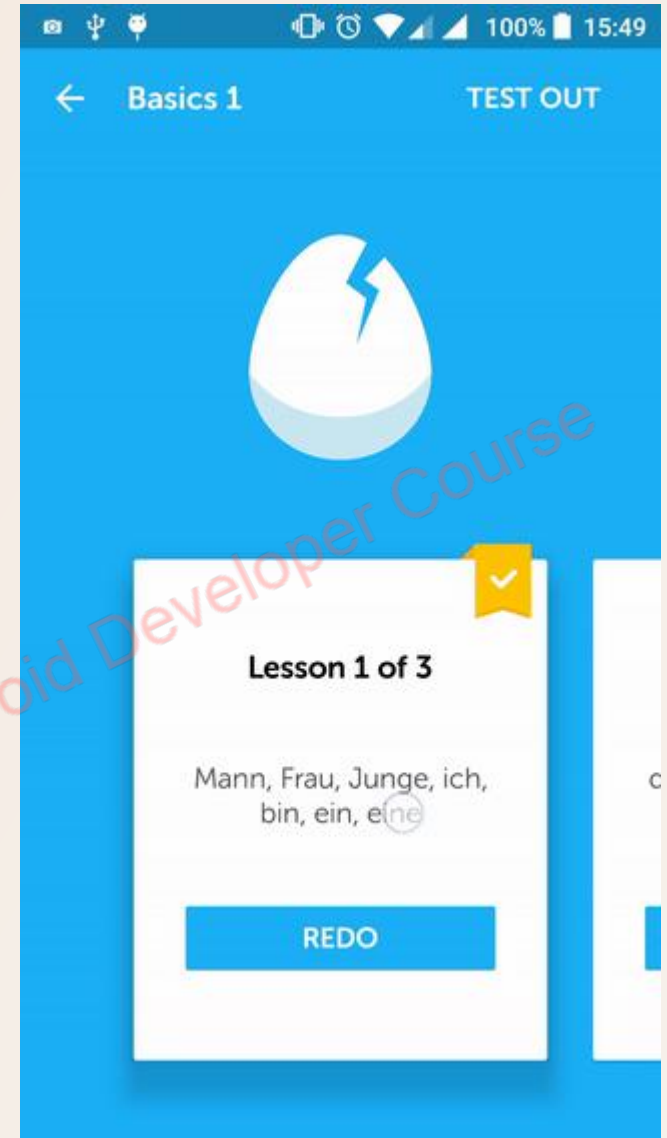


View Pager

ViewPager is a layout manager that allows the user to flip left and right through pages of data.

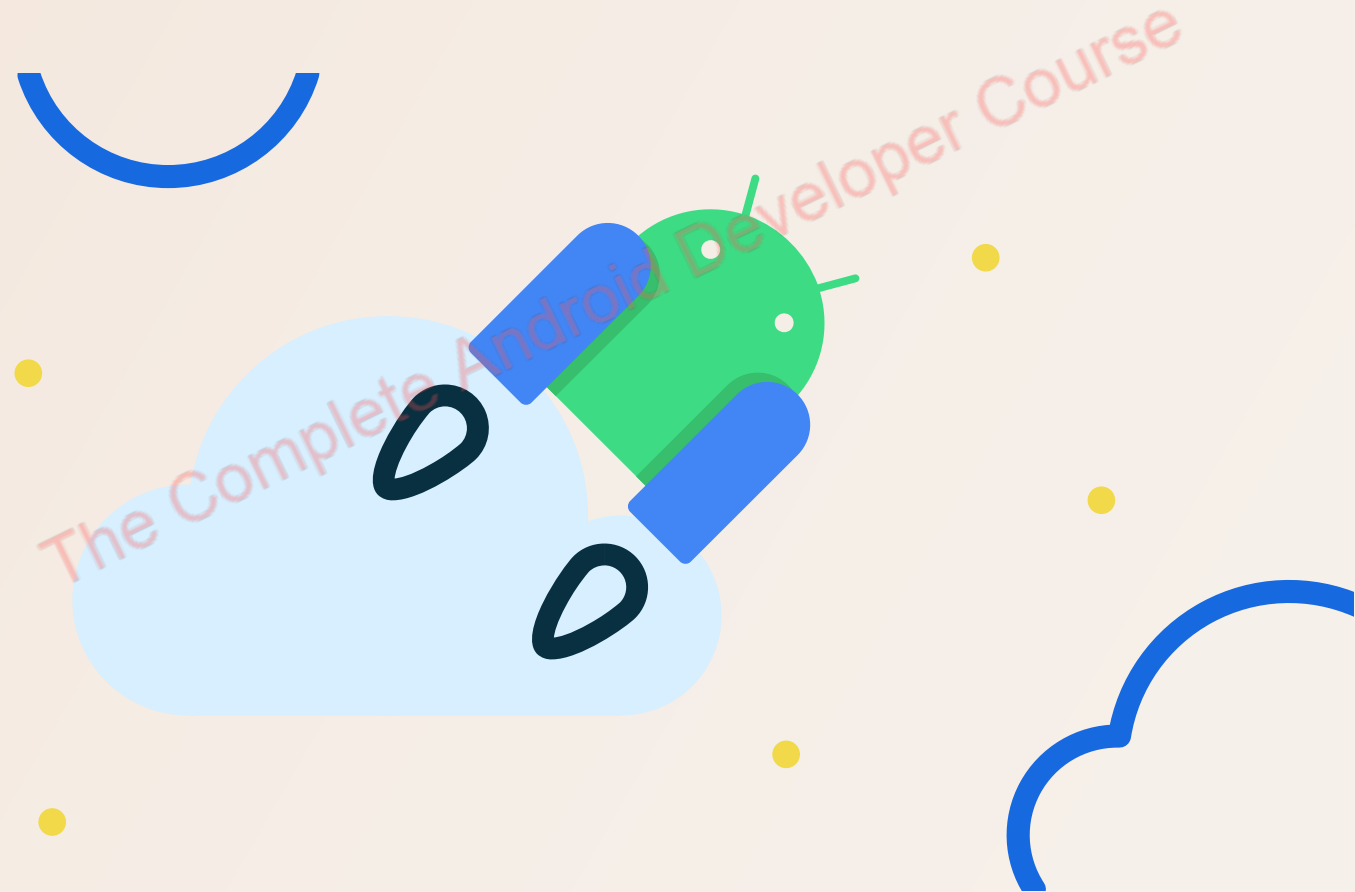
It is mostly found in apps like Youtube, Snapchat where the user shifts right – left to switch to a screen. Instead of using activities **fragments** are used.

It is also used to guide the user through the app when the user launches the app for the first time.



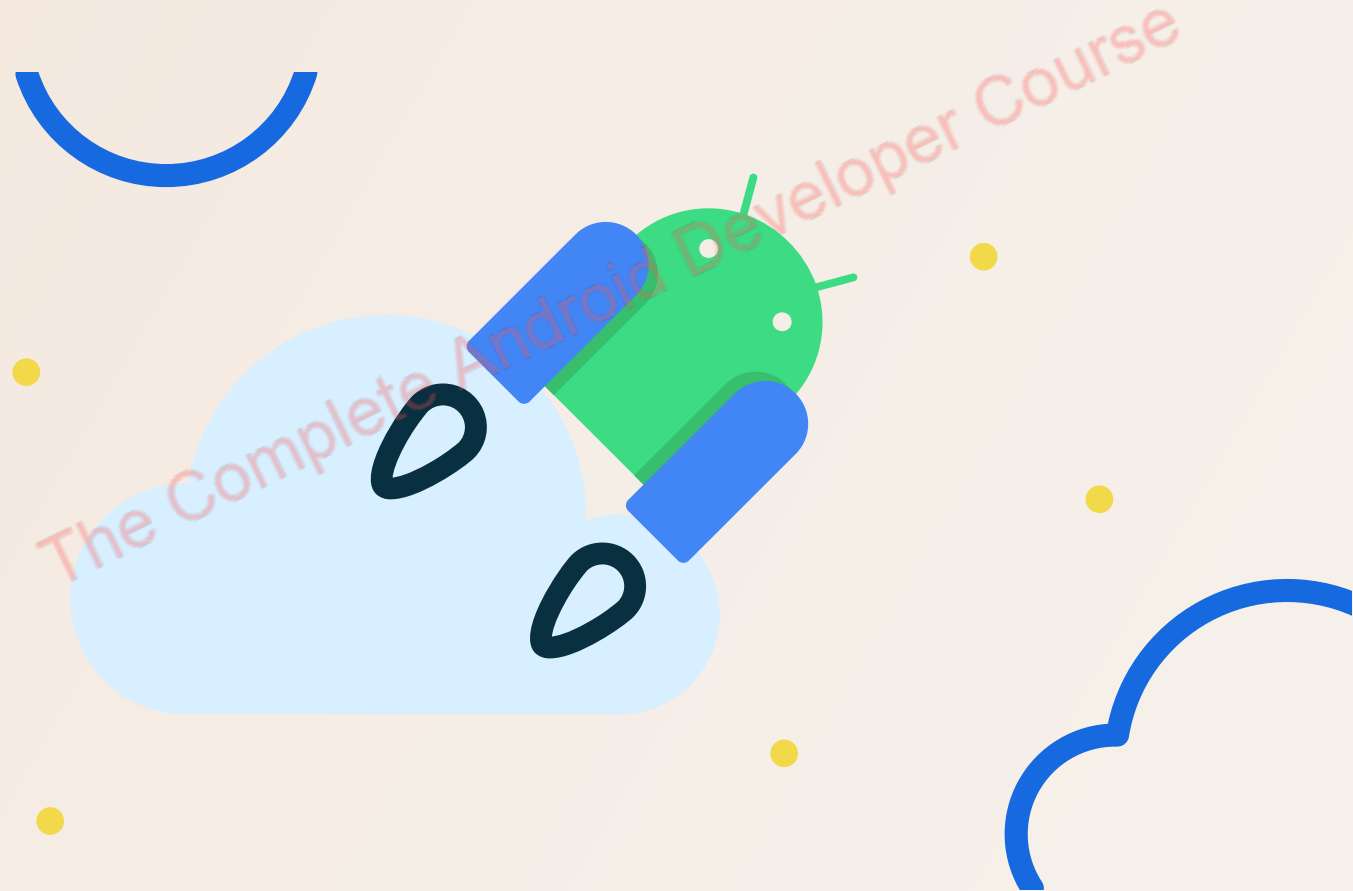
Android Jetpack

Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about.



Why Android Jetpack?

Android Jetpack helps solve major problems such as managing activity life cycles, configuration changes, and preventing memory leaks.



Jetpack Components

Android Jetpack components bring together the existing support library and architecture components and define them into four categories.



Architecture



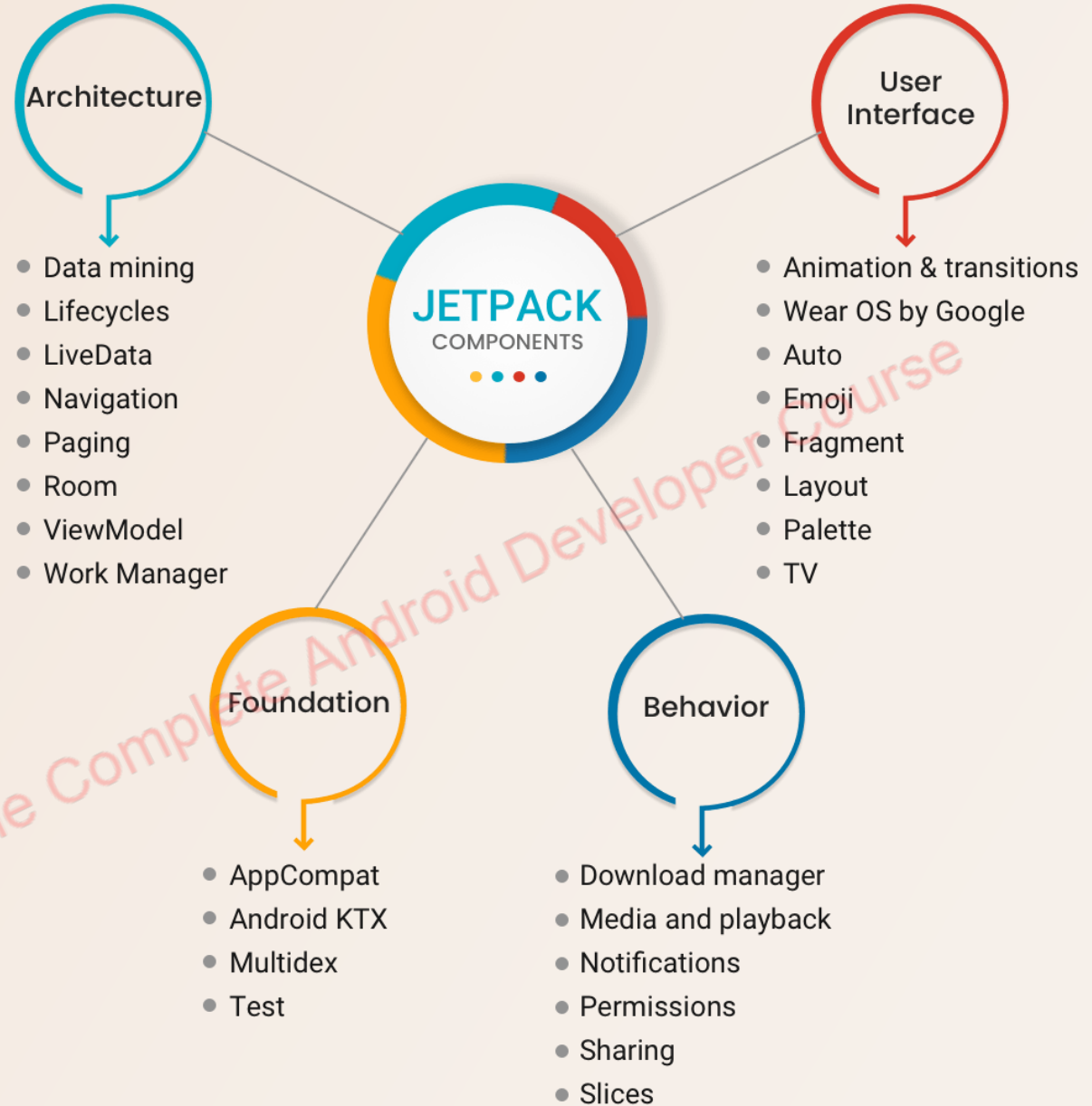
User Interface



Foundation



Behavior



findViewById()

Finds the first descendant view with the given ID, the view itself if the ID matches getId(), or null if the ID is invalid (< 0) or there is no matching view in the hierarchy.

Every time we use findViewById() to get reference to a view, Android system must go through the view hierarchy and find it at runtime. But what about large apps with hundreds of views and layouts????

Modern mobiles nowadays, have a refresh rate 60 HZ per 1 second (1000 milliseconds), so Android system will create our layout with all its views every 16.667 milliseconds.

What about Mobiles with 90 HZ, 120 HZ, 144 HZ ????

```
<Button
    android:id="@+id/btn_submit"
    android:text="Submit"
    android:textAllCaps="false"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

```
var btn_submit = findViewById(R.id.btn_submit) as Button
```

```
btn_submit.setOnClickListener {
}
```

Get the reference to the View

Use the reference to access its properties

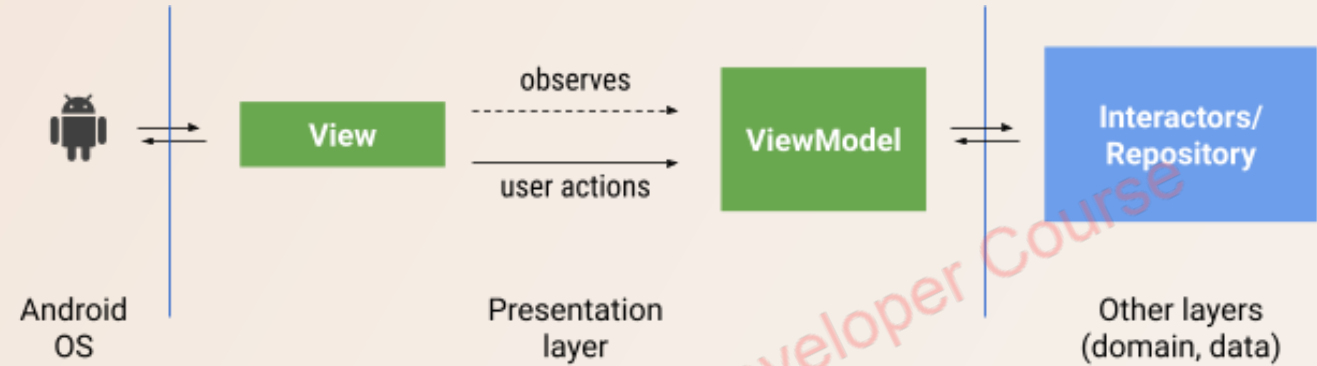
The Complete Android Developer Course

Data Binding

The Data Binding Library is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

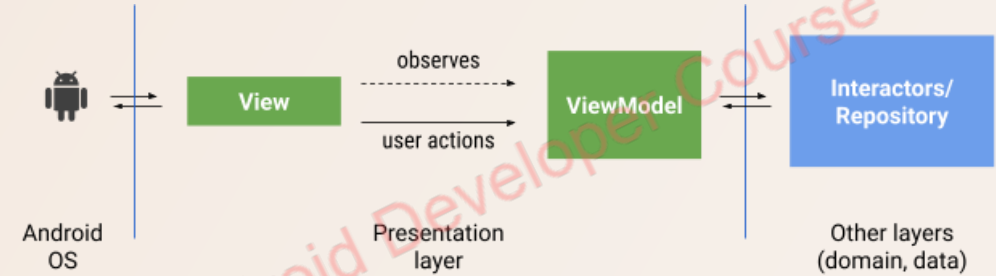
Data binding is the process of integrating views in an XML layout with data objects. The Data Binding Library is responsible for generating the classes required for this procedure.

Using Data binding, a binding object will be created that contains a reference to each view of a layout, so the android system will not go and search the views by their ids again and again....



Why Data Binding?

- You can reduce findViewById calls and enhance your app's performance
- Helps get rid of memory leaks or nullPointerExceptions
- Uses declarative layout, which is more adaptable
- Supercharges developer productivity by writing error-free, shorter, simpler-to-understand, and more maintainable code
- Data and Views are separated from each other
- The compiler verifies types during compile time and displays errors if and when you attempt to assign the incorrect type to a variable, thanks to type-safety



Using Data Binding

The name of the layout we wrapped with `<layout></layout>` in `activity_main`.

Using that, android data binding library will create a binding object with the name of: `ActivityMainBinding`

activity_main

ActivityMainBinding

The Complete Android Developer Course



Quadratic Equations

Quadratic equations are algebraic equations that have the form $ax^2+bx+c=0$

Considering this form, the discriminant of a quadratic equation is the value b^2-4ac . This value goes inside the square root of the general quadratic formula and determines the type of solutions that we will have.

$$ax^2 + bx + c = 0$$

$$D = b^2 - 4ac$$

When $D > 0$, the quadratic equation has two real roots.

When $D < 0$, the quadratic equation has no real roots.

When $D = 0$, the quadratic equation has a repeated root.

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

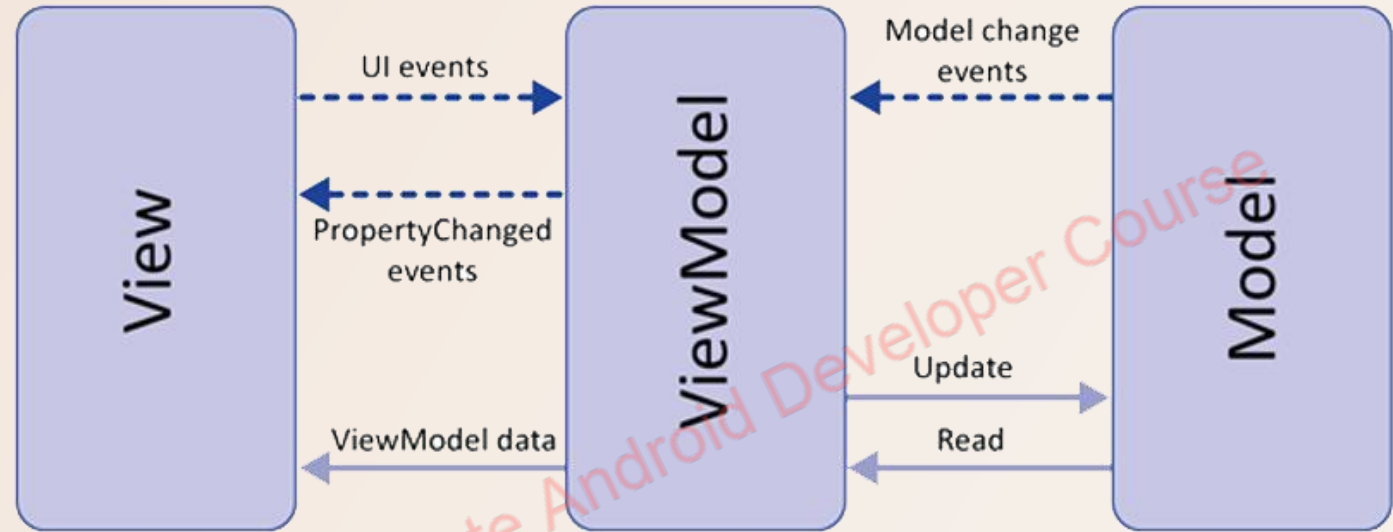
We can understand this better if we remember that the general quadratic formula, which allows us to solve any quadratic equation, is the following:

View Model

ViewModel is a class that is responsible for preparing and managing the data for an Activity or a Fragment. It also handles the communication of the Activity / Fragment with the rest of the application.

A ViewModel is always created in association with a scope (a fragment or an activity) and will be retained as long as the scope is alive. E.g. if it is an Activity, until it is finished.

In other words, this means that a ViewModel will not be destroyed if its owner is destroyed for a configuration change (e.g. rotation).

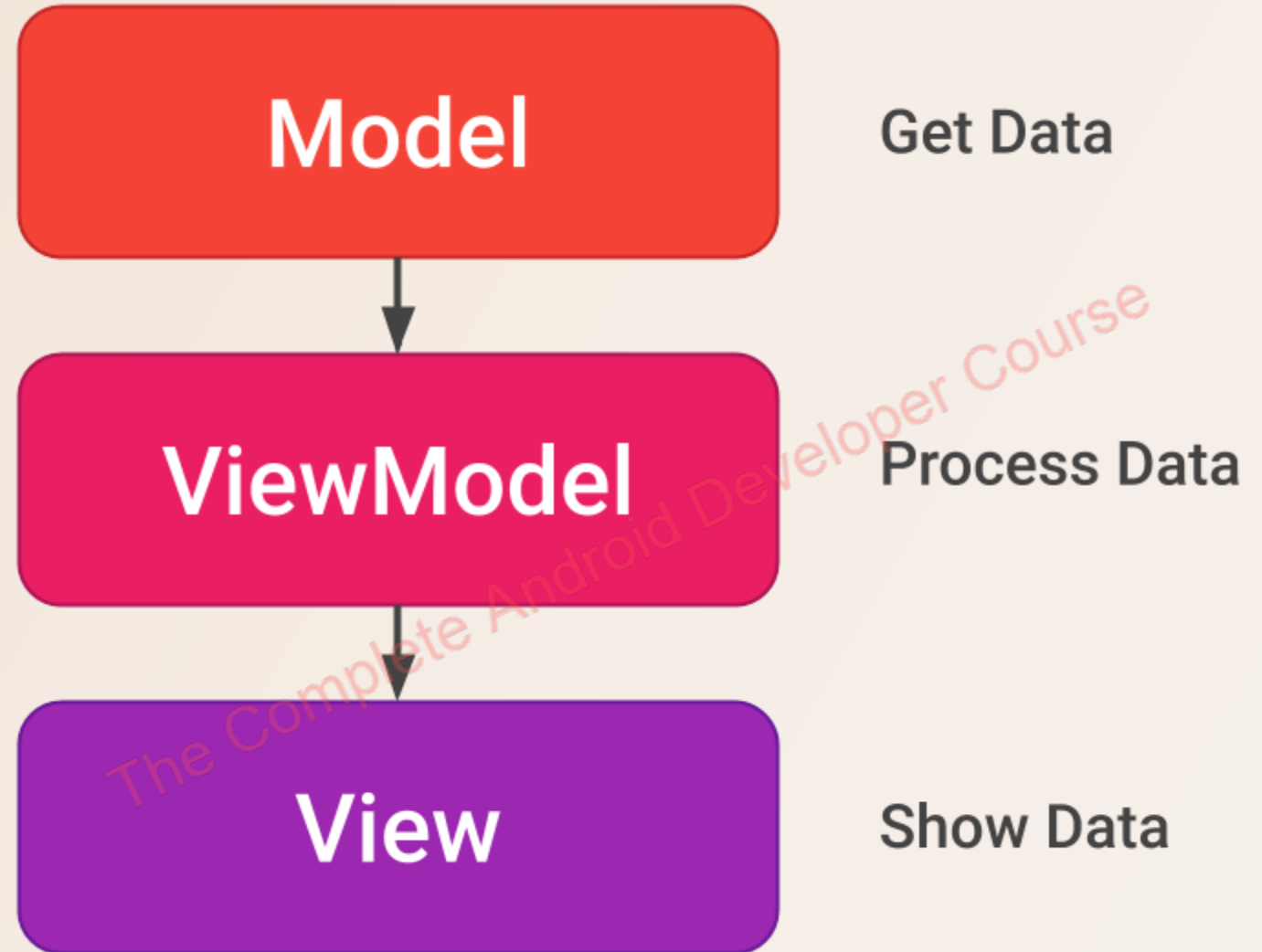


View Model

The ViewModel class is a business logic or screen level state holder. It exposes state to the UI and encapsulates related business logic.

Its principal advantage is that it caches state and persists it through configuration changes.

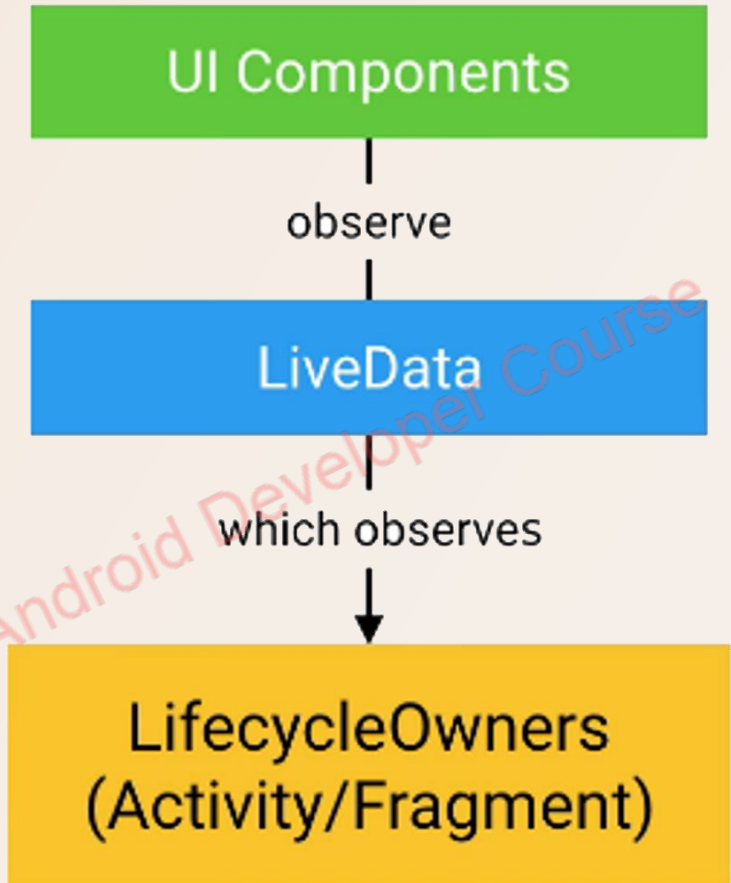
This means that your UI doesn't have to fetch data again when navigating between activities, or following configuration changes, such as when rotating the screen.



Live Data

LiveData is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services.

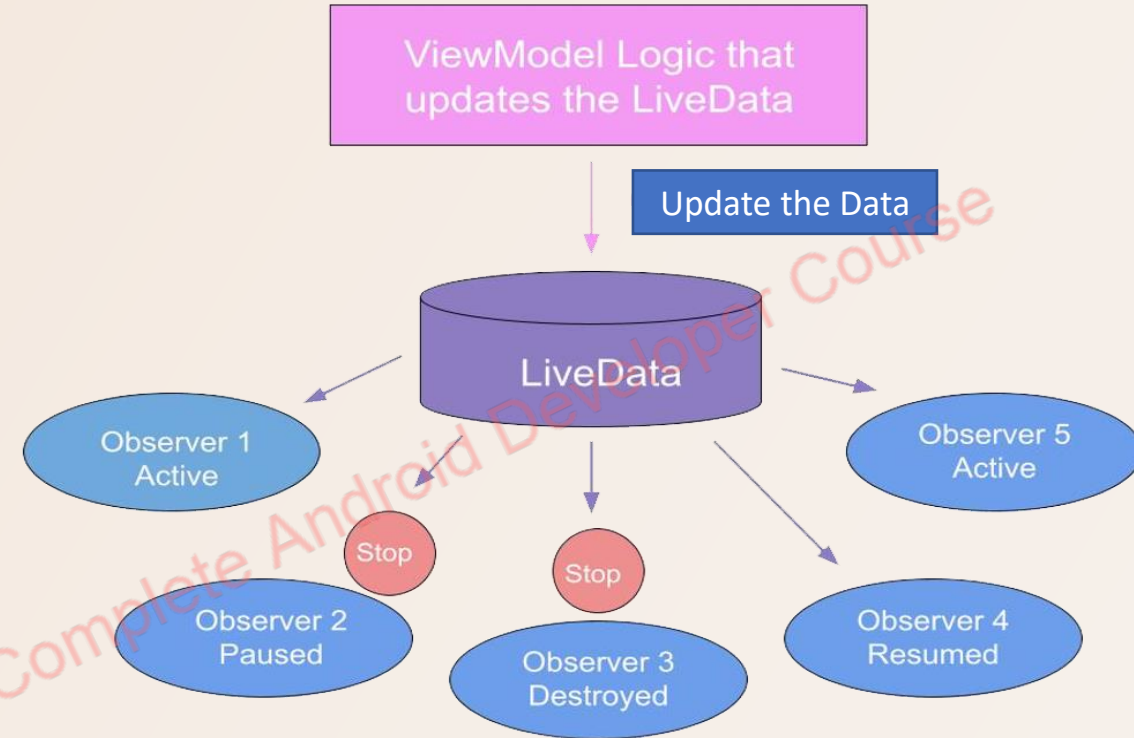
This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.



Live Data

If the lifecycle status is STARTED or RESUMED, LiveData considers an observer to be in an **active state**. Only active observers are notified of lifecycle updates by LiveData.

- It eliminates memory leaks caused by the multiple callbacks that send results to the UI thread, ensuring that the UI is always up to date.
- It de-couples tight integration between data, mediator, and the UI to avoid crashed activities.
- UI components continuously monitor relevant data, and LiveData manages all these tasks automatically as the relevant lifecycle status changes.
- If the activity or fragment is recreated due to configuration changes such as device rotation (Portrait, Landscape), it immediately receives the latest information from LiveData.
- Extended Live Data wraps system services so that they can be shared within the app.

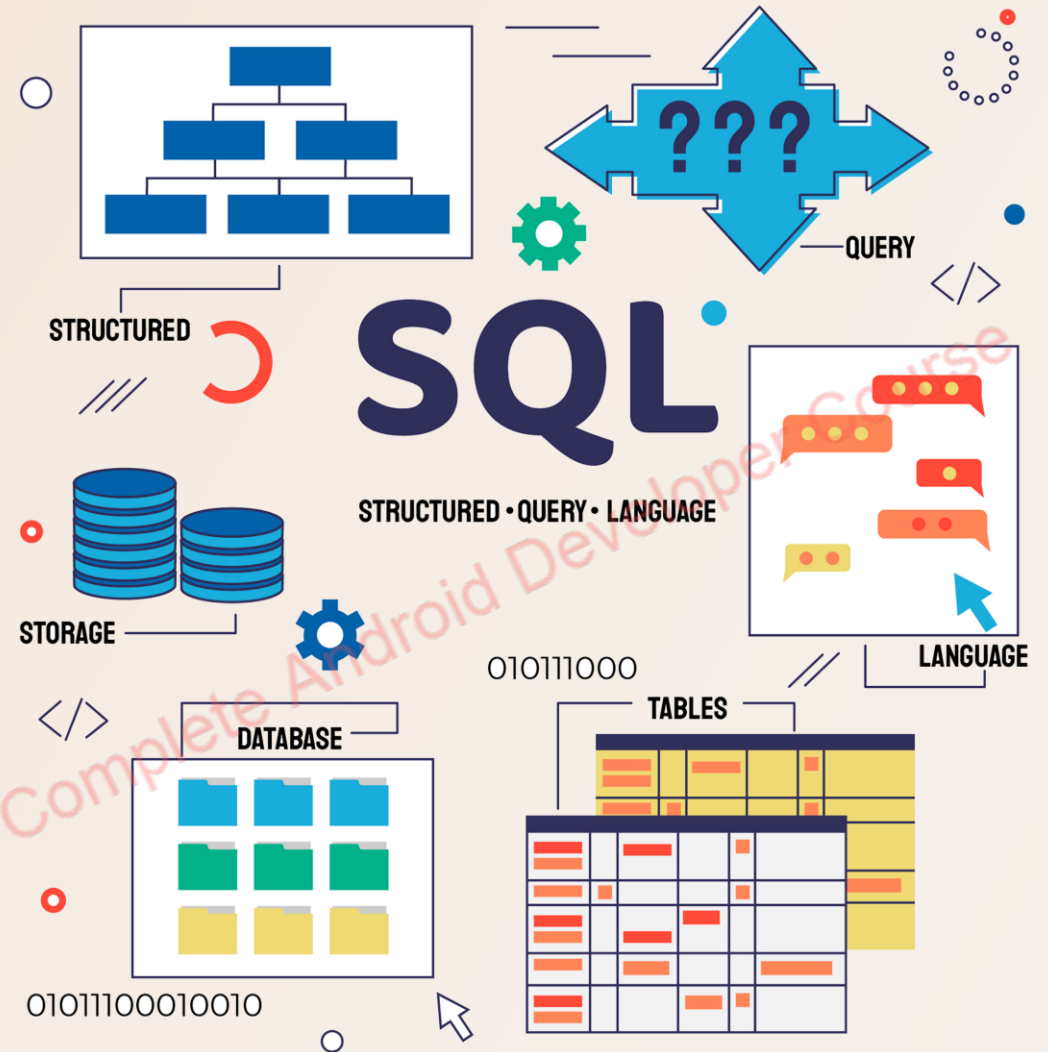


SQLite Database

SQLite Database is an open-source database provided in Android which is used to store data inside the user's device in the form of a Text file.

We can perform so many operations on this data such as adding new data, updating, reading, and deleting this data.

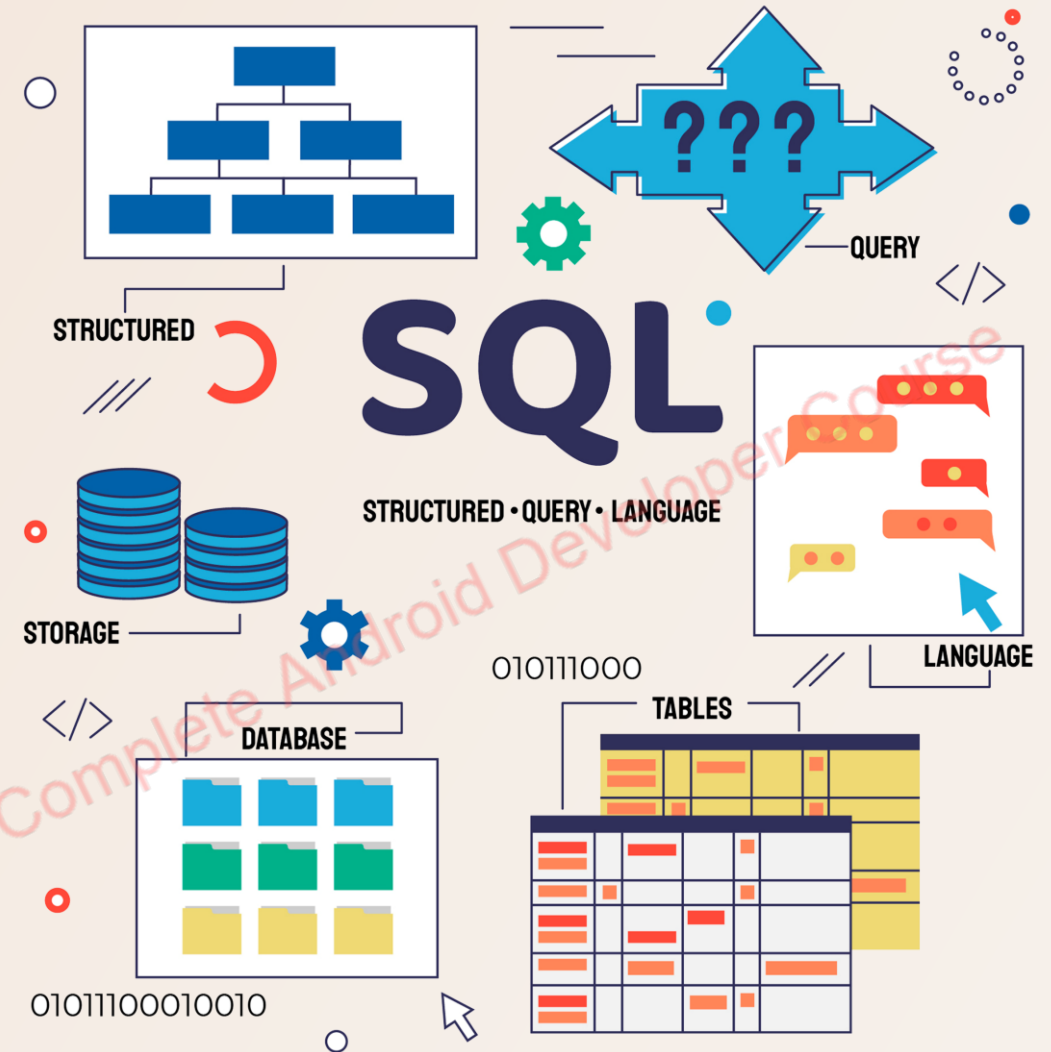
SQLite is an offline database that is locally stored in the user's device and we do not have to create any connection to connect to this database.



ROOM Database

Room is an ORM, Object Relational Mapping library. In other words, Room will map our database objects to Java objects.

Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.



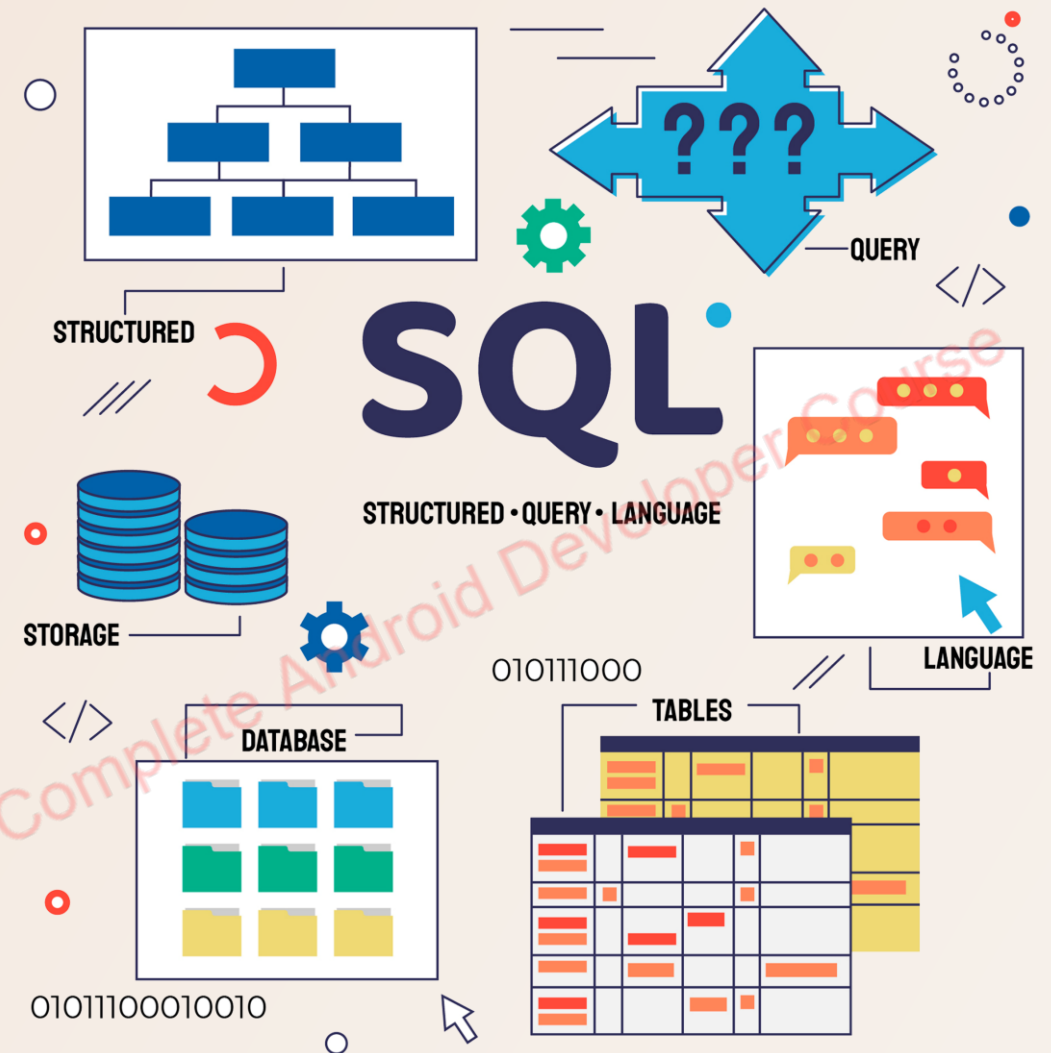
SQLite vs ROOM Database

In the case of SQLite, There is no compile-time verification of raw SQLite queries. But in Room, there is SQL validation at compile time.

You need to use lots of boilerplate code to convert between SQL queries and Java data objects. But, Room maps our database objects to Java Object without boilerplate code.

As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.

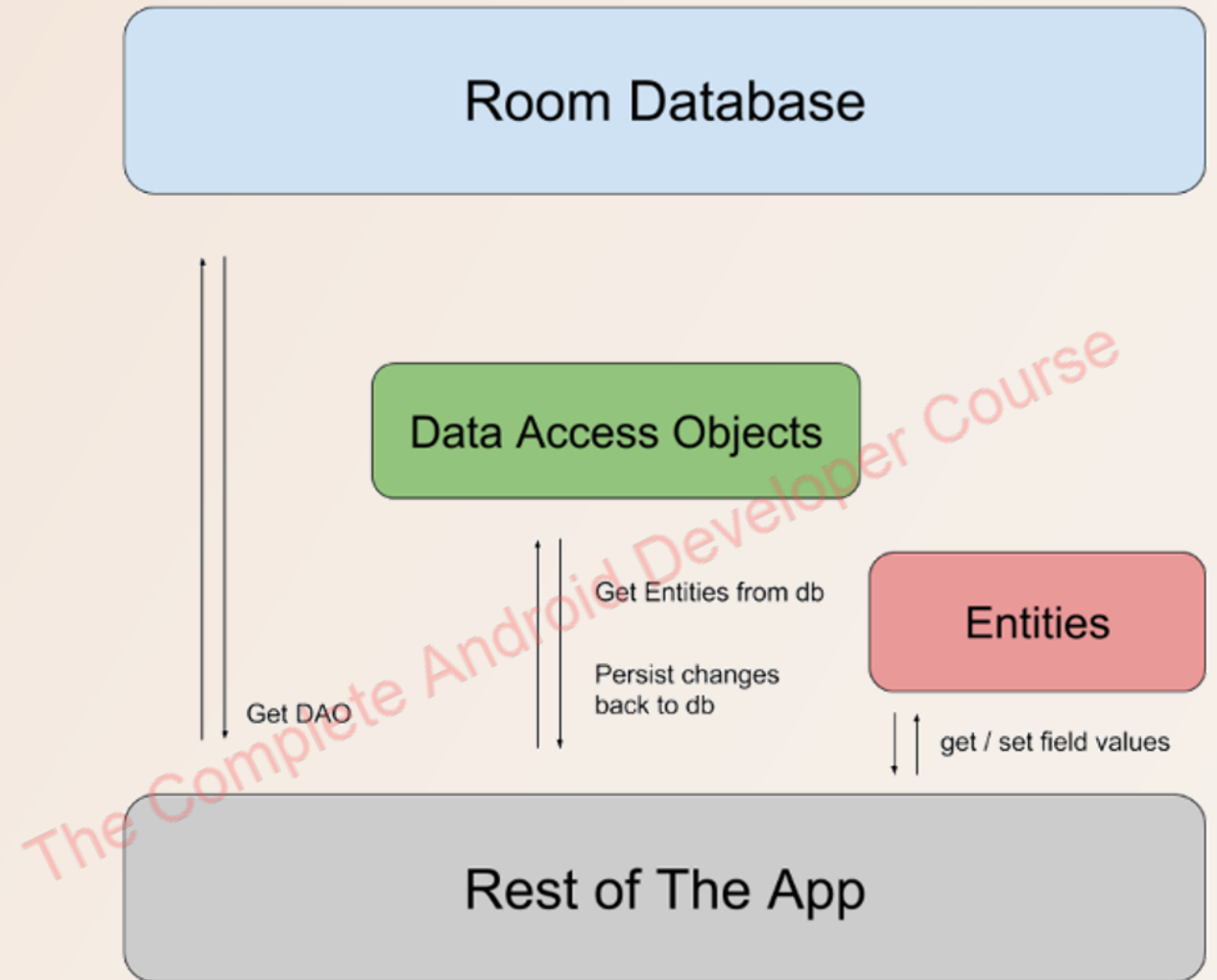
Room is built to work with LiveData and RxJava for data observation, while SQLite does not.



ROOM Database

Room has three main components of Room DB :

- Entity
- Dao
- Database



ROOM Database

DAO (Database Access Object):

In Room you use data access objects to access and manage your data. The DAO is the main component of Room and includes methods that offer access to your app's database. It has to be annotated with `@Dao`.

DAOs are used instead of query builders and let you separate different components of your database e.g. current data and statistics, which allows you to easily test your database.



Entity:

The Entity represents a table within the database and has to be annotated with `@Entity`.

Each Entity consists of a minimum of one field that has to define a primary key.



Database:

Serves as the database holder and is the main access point to your relational data.

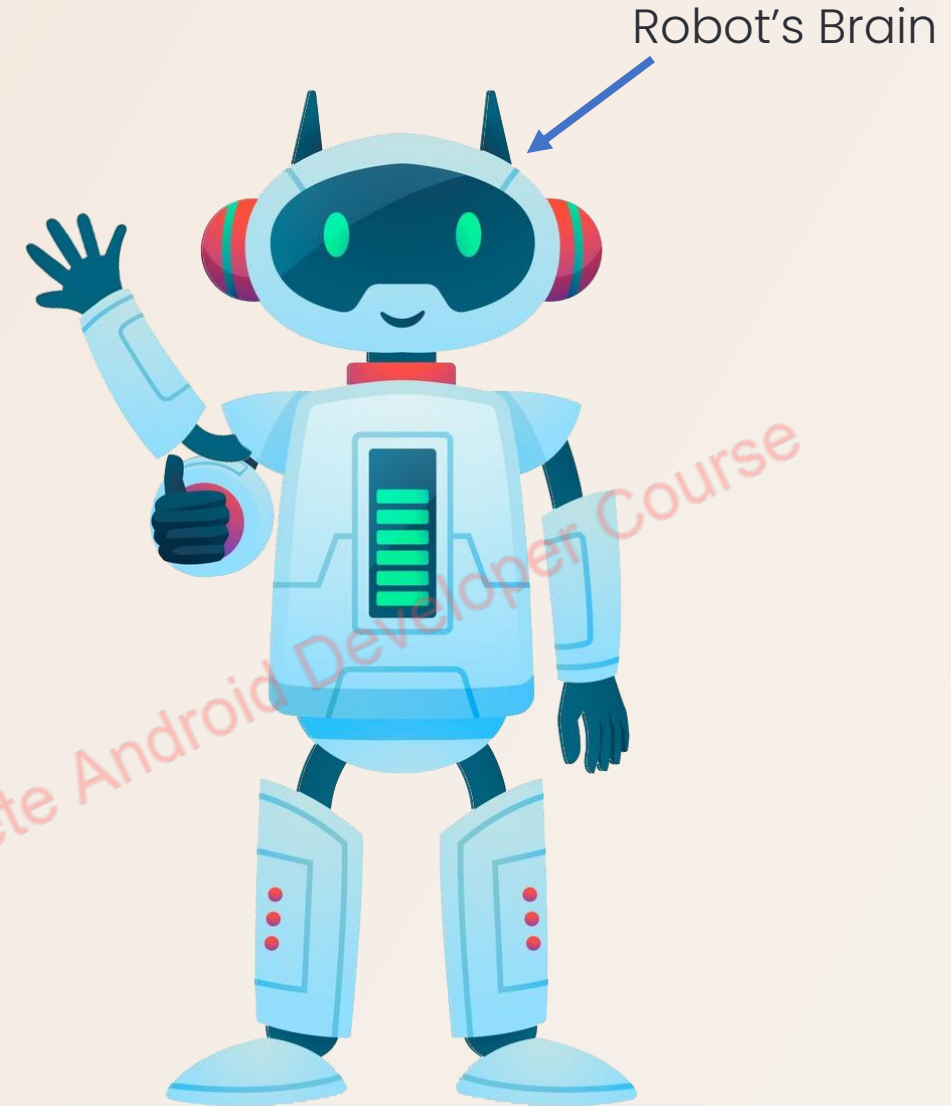
It has to be annotated with `@Database` and extends the `RoomDatabase`. It also contains and returns the Dao (Database Access Object).



What's MVVM?

Here's how MVVM works with our robot:

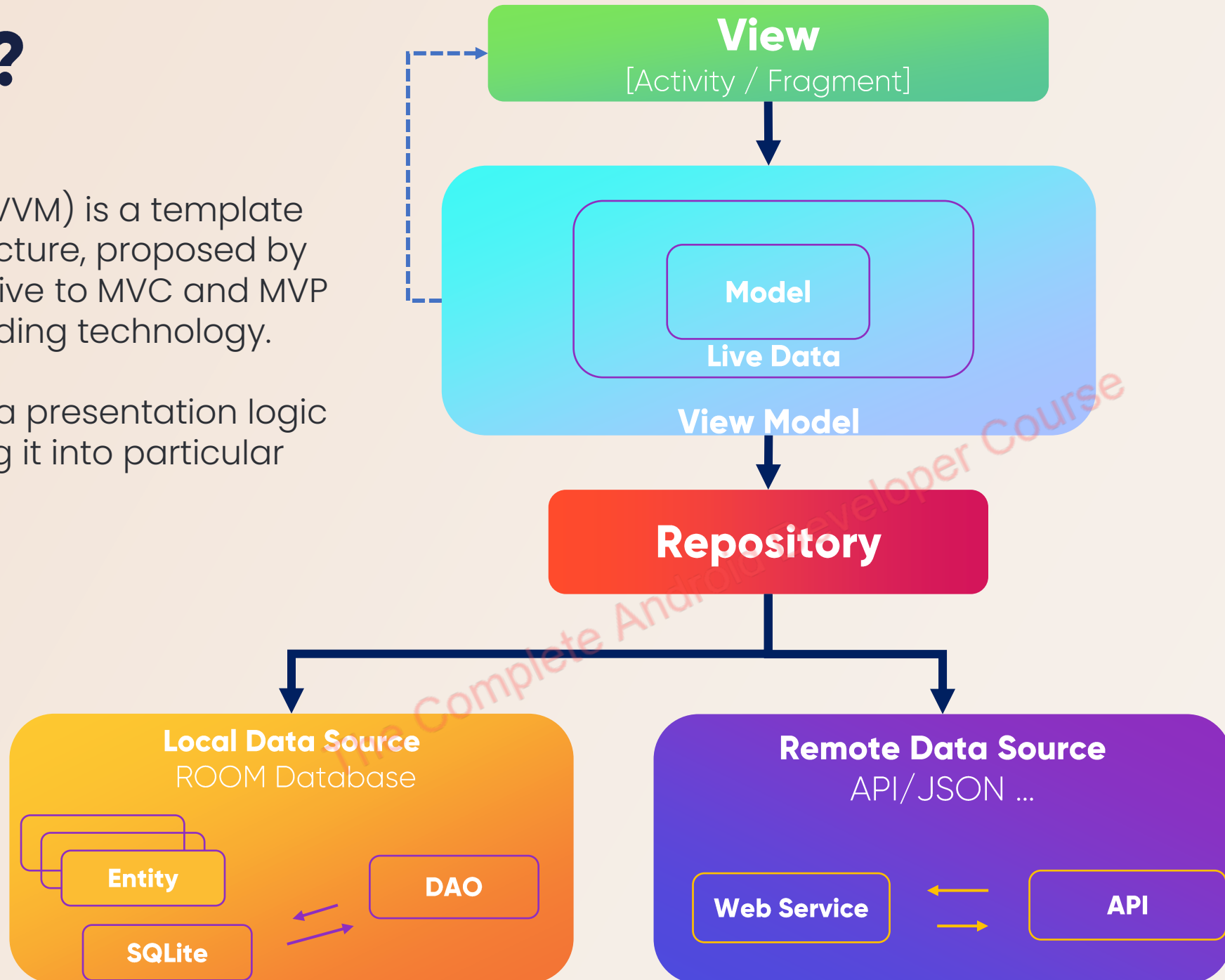
- Model (Robot's Brain)
- View (Remote Control)
- View Model (Translator)



What's MVVM?

Model-View-ViewModel (ie MVVM) is a template of a client application architecture, proposed by John Gossman as an alternative to MVC and MVP patterns when using Data Binding technology.

Its concept is to separate data presentation logic from business logic by moving it into particular class for a clear distinction.



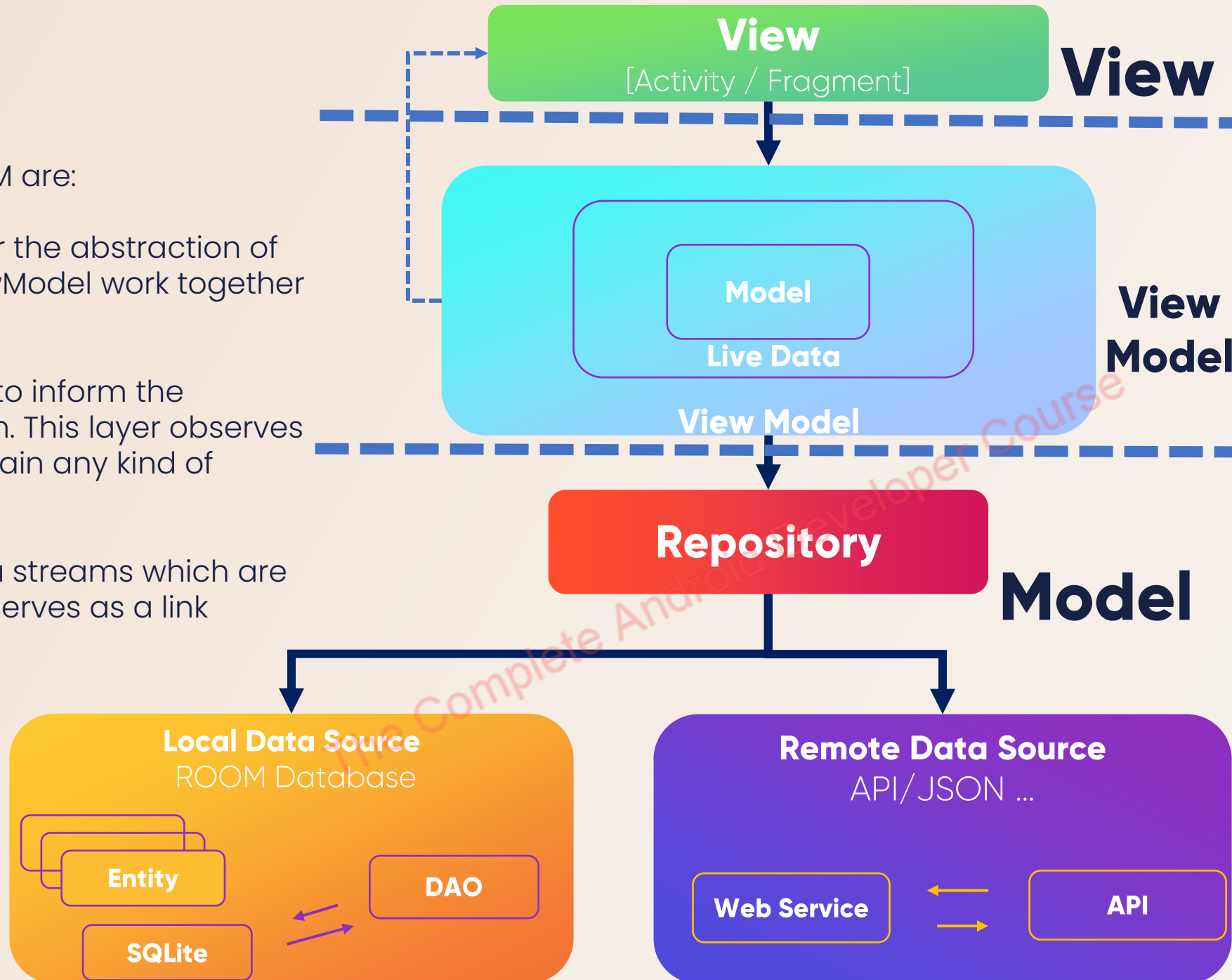
MVVM Layers

The separate code layers of MVVM are:

Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

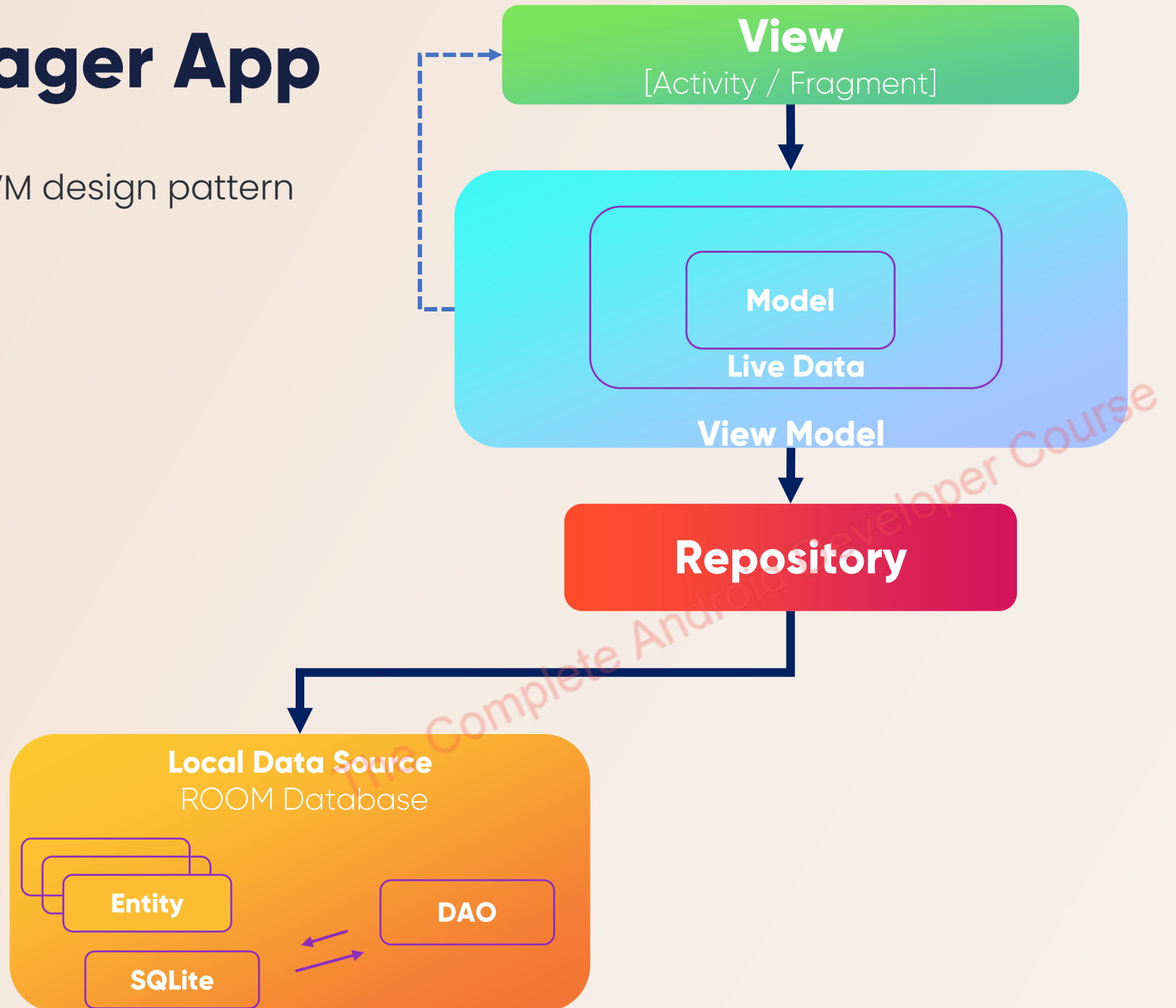
View: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.

ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.



Contacts Manager App

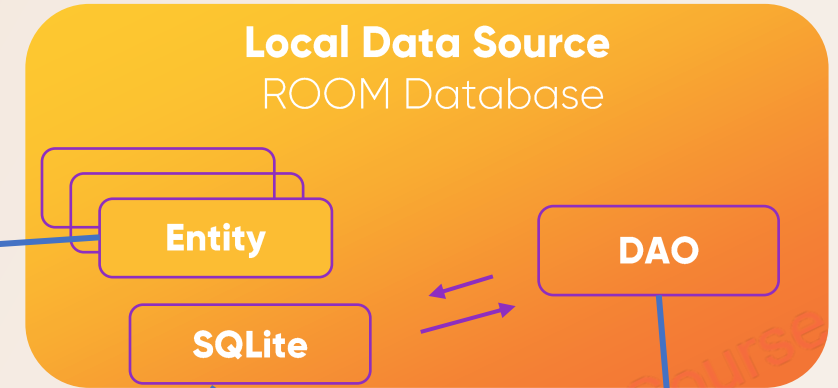
In this app, we'll follow the MVVM design pattern with ROOM database.



Contacts Manager App – [ROOM DB]

contact_id	contact_name	contact_email
1	John	john@gmail.com
2	Sara	sara@gmail.com
3	Ali	ali@gmail.com

Entity: contacts_table



DAO

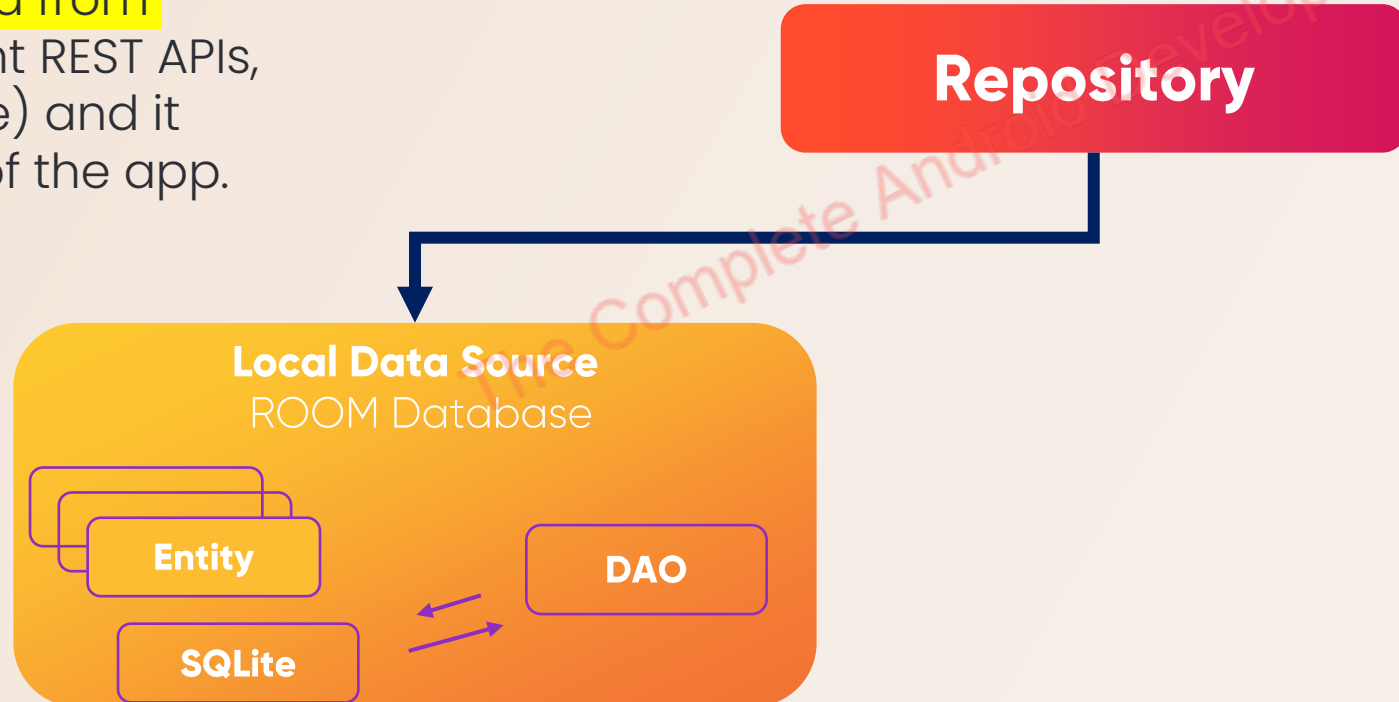
- **Insert**
- **Delete**
- **Update**
- **Select**
- **Other queries**

Contacts Manager App – [Repository]

Repository is a class that is mainly used to manage multiple sources of data.

The repository class isolates the data sources from the rest of the app and provides a clean API for data access to the rest of the app.

the Repository can gather data from different data sources (different REST APIs, cache, local database storage) and it provides this data to the rest of the app.



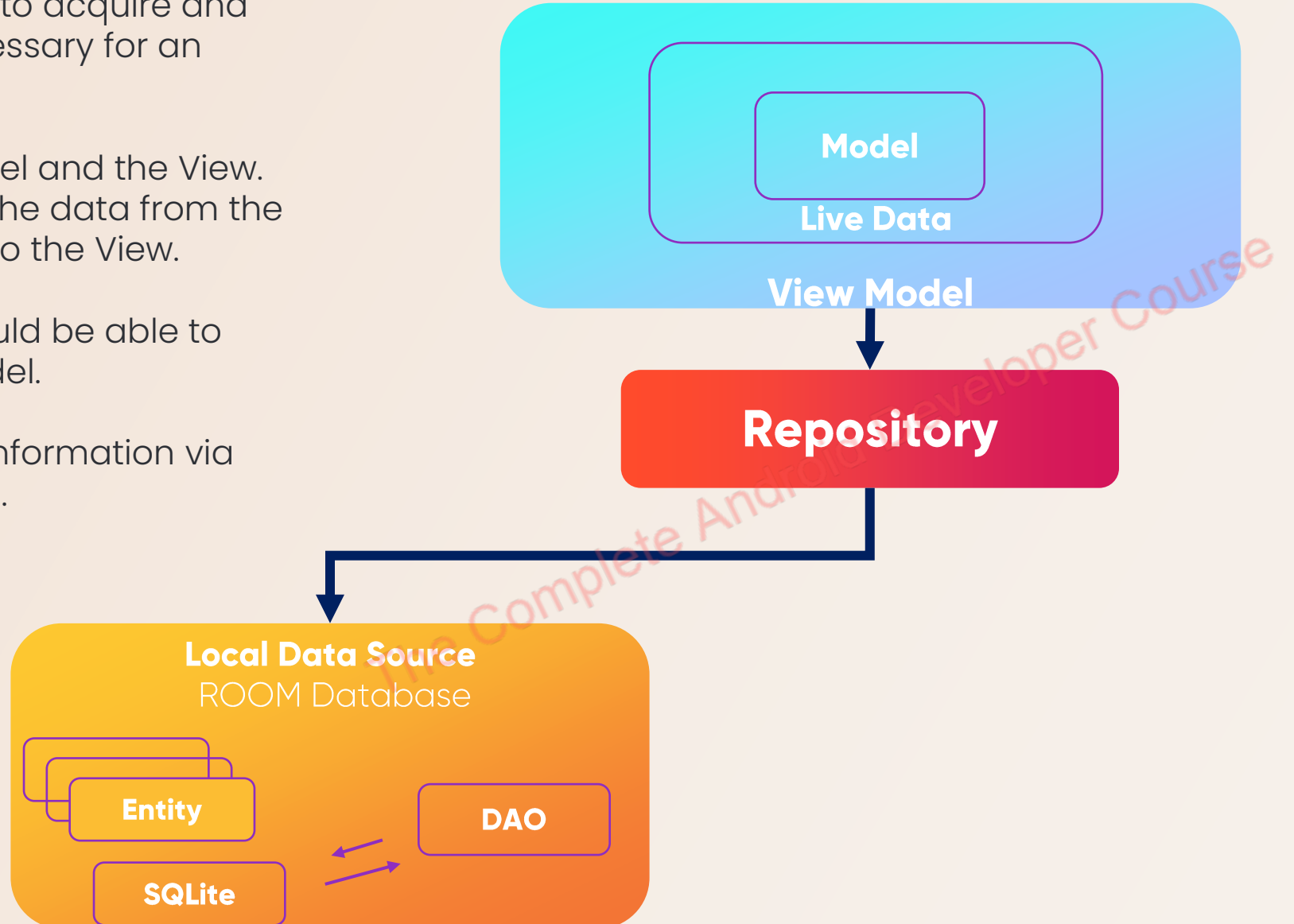
Contacts Manager App – [View Model]

The purpose of the ViewModel is to acquire and keep the information that is necessary for an Activity or a Fragment.

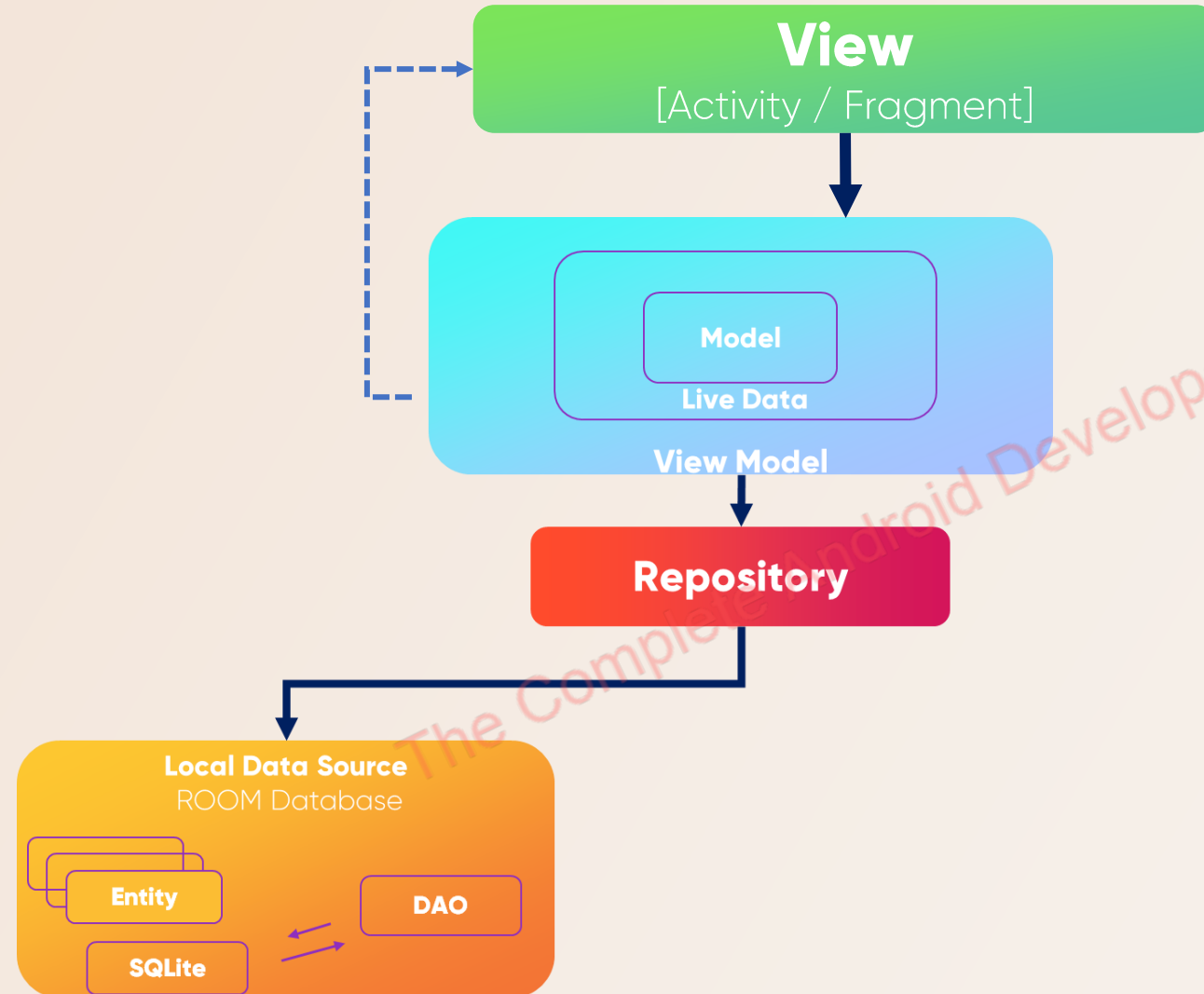
It acts as a link between the Model and the View. It's responsible for transforming the data from the Model. It provides data streams to the View.

The Activity or the Fragment should be able to observe changes in the ViewModel.

ViewModels usually expose this information via LiveData or Android Data Binding.



Contacts Manager App – [View]



FIREBASE

Firebase is a powerful and comprehensive platform developed by Google that offers a wide range of tools and services for building and managing mobile and web applications.

Firebase for Android is a popular choice among developers because it simplifies many aspects of app development, including authentication, real-time database management, cloud storage, and more.










The Complete Android Developer Course

FIREBASE SERVICES







Build better apps

-  **Cloud Firestore**
Store and sync app data at global scale
-  **Firebase ML** BETA
Machine learning for mobile developers
-  **Cloud Functions**
Run mobile backend code without managing servers
-  **Authentication**
Authenticate users simply and securely
-  **Hosting**
Deliver web app assets with speed and security
-  **Cloud Storage**
Store and serve files at Google scale
-  **Realtime Database**
Store and sync app data in milliseconds










Improve app quality

-  **Crashlytics**
Prioritize and fix issues with powerful, realtime crash reporting
-  **Performance Monitoring**
Gain insight into your app's performance
-  **Test Lab**
Test your app on devices hosted by Google
-  **App Distribution** BETA
Distribute pre-release versions of your app to your trusted testers



Grow your business

-  **In-App Messaging** BETA
Engage active app users with contextual messages
-  **Google Analytics**
Get free and unlimited app analytics
-  **Predictions**
Smart user segmentation based on predicted behavior
-  **A/B Testing** BETA
Optimize your app experience through experimentation
-  **Cloud Messaging**
Send targeted messages and notifications
-  **Remote Config**
Modify your app without deploying a new version
-  **Dynamic Links**
Drive growth by using deep links with attribution

The Complete Android Developer Course

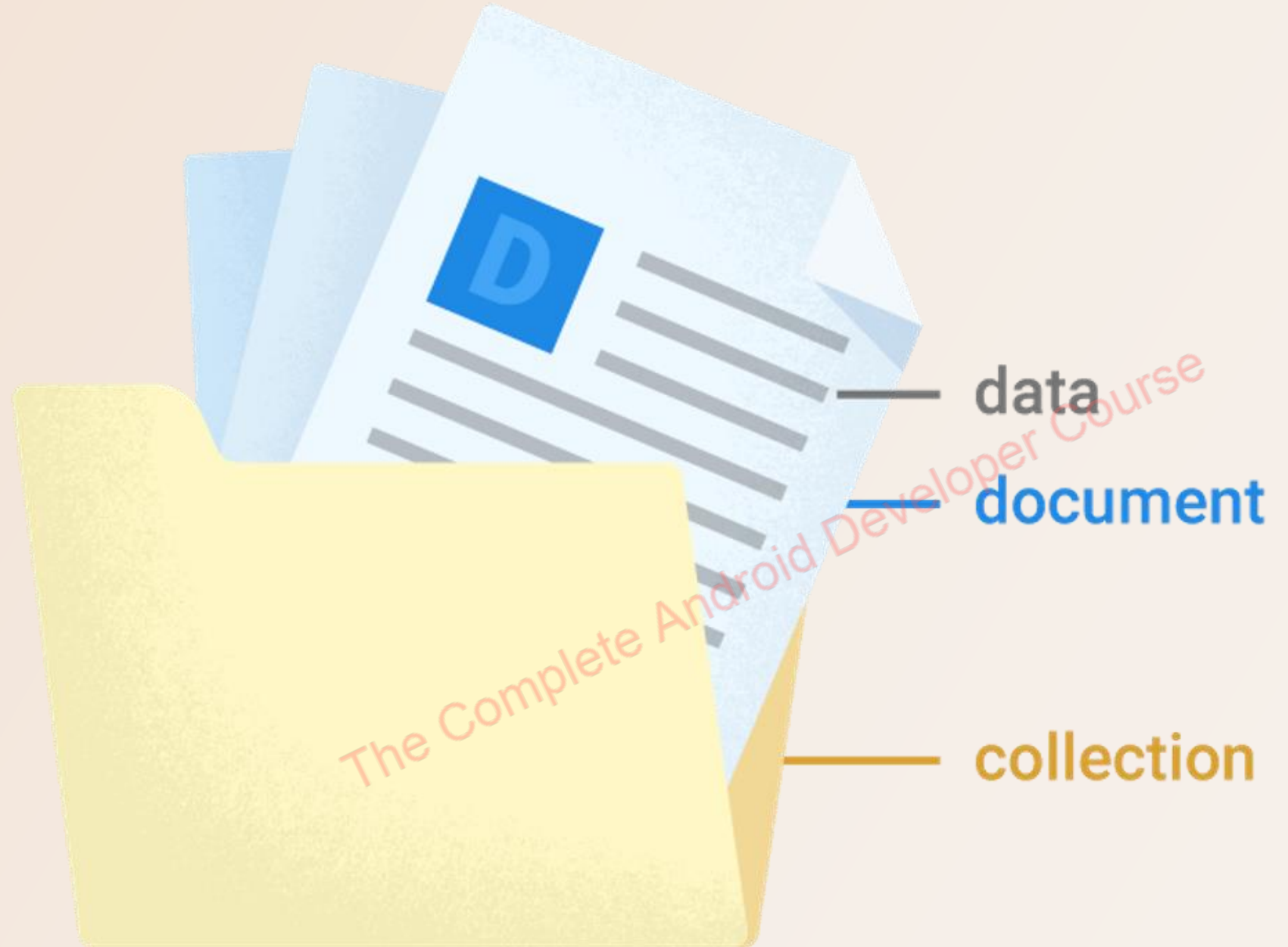
FIRESTORE

In Cloud Firestore, the unit of storage is the document. A document is a lightweight record that contains fields, which map to values. Each document is identified by a name.

A document representing a user might look like this:

User:

first : "jack"
last : "reacher"
born : 1992



FIRESTORE

Documents live in collections, which are simply containers for documents. For example, you could have a "Users" collection to contain your various users, each represented by a document:

Users:

user1

first : "jack"

last : "reacher"

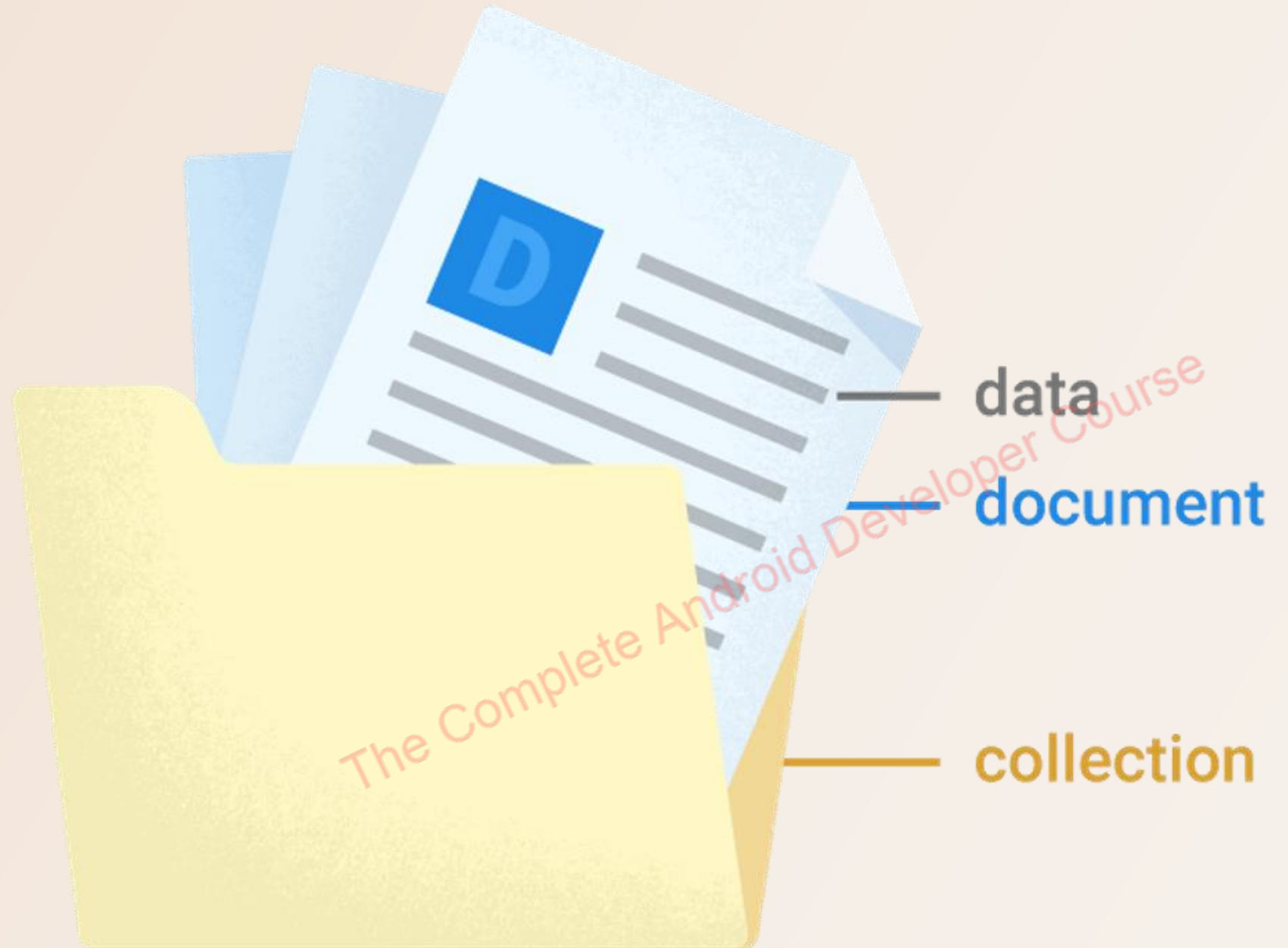
born : 1992

user2

first : "john"

last : "travolta"

born : 1982



WORK MANAGER

WorkManager is the recommended solution for persistent work. Work is persistent when it remains scheduled through app restarts and system reboots.

Because most background processing is best accomplished through persistent work, WorkManager is the primary recommended API for background processing.

Work manager simplifies and manages background tasks in android apps. It allows you to schedule and run tasks in the background, even when the app is not currently in the foreground



WORK MANAGER

Work manager offers several advantages, such as:

- handling tasks efficiently
- considering network connectivity
- considering device state
- supporting periodic scheduling



WORK MANAGER

Worker

A worker is a class that performs a background task.

Work Request:

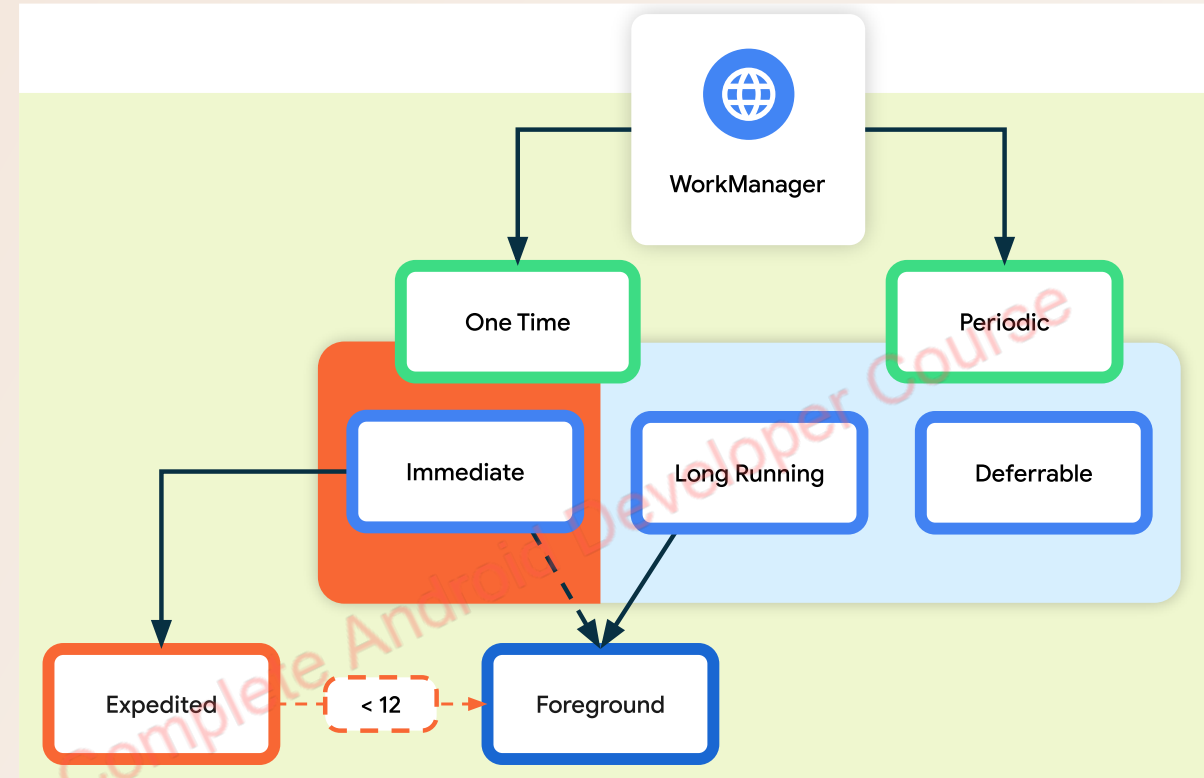
Defines the task to be executed and its constraints.

Types of WorkRequest:

- OneTimeWorkRequest
- PeriodicWorkRequest

Work Manager:

The central component of managing background tasks. It is responsible for scheduling and executing WorkRequest.



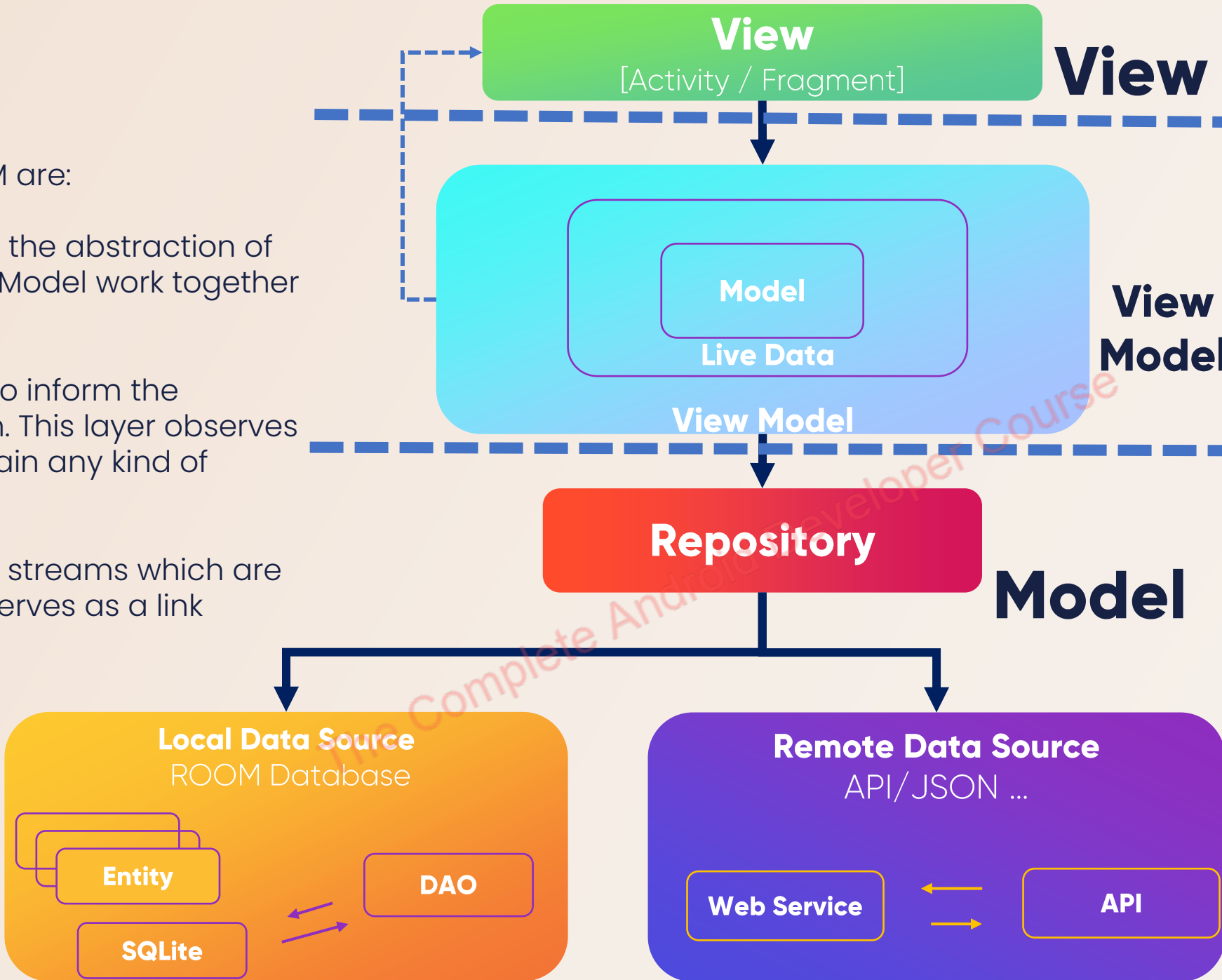
MVVM Layers

The separate code layers of MVVM are:

Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

View: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.

ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.



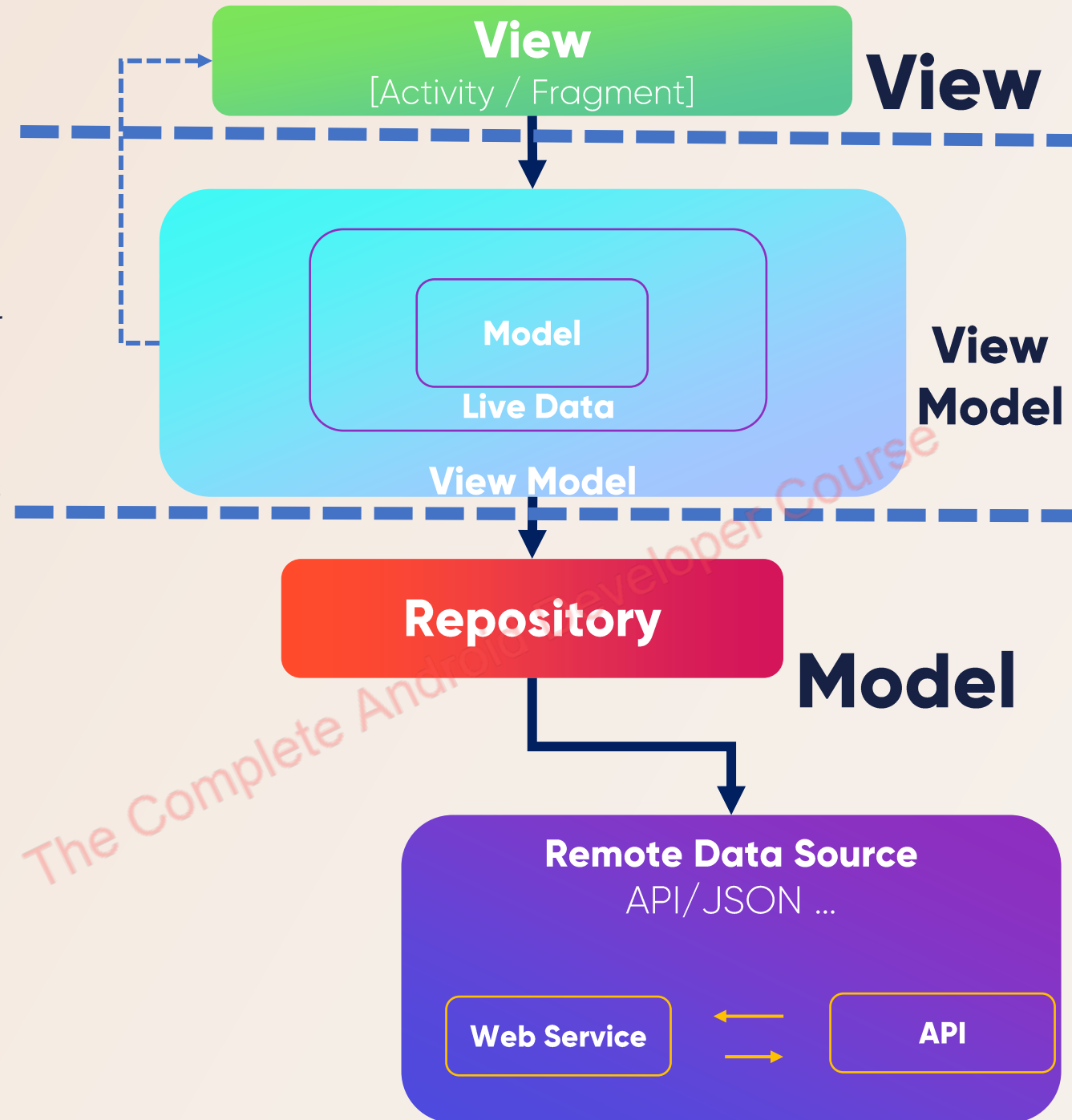
MVVM Layers

The separate code layers of MVVM are:

Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

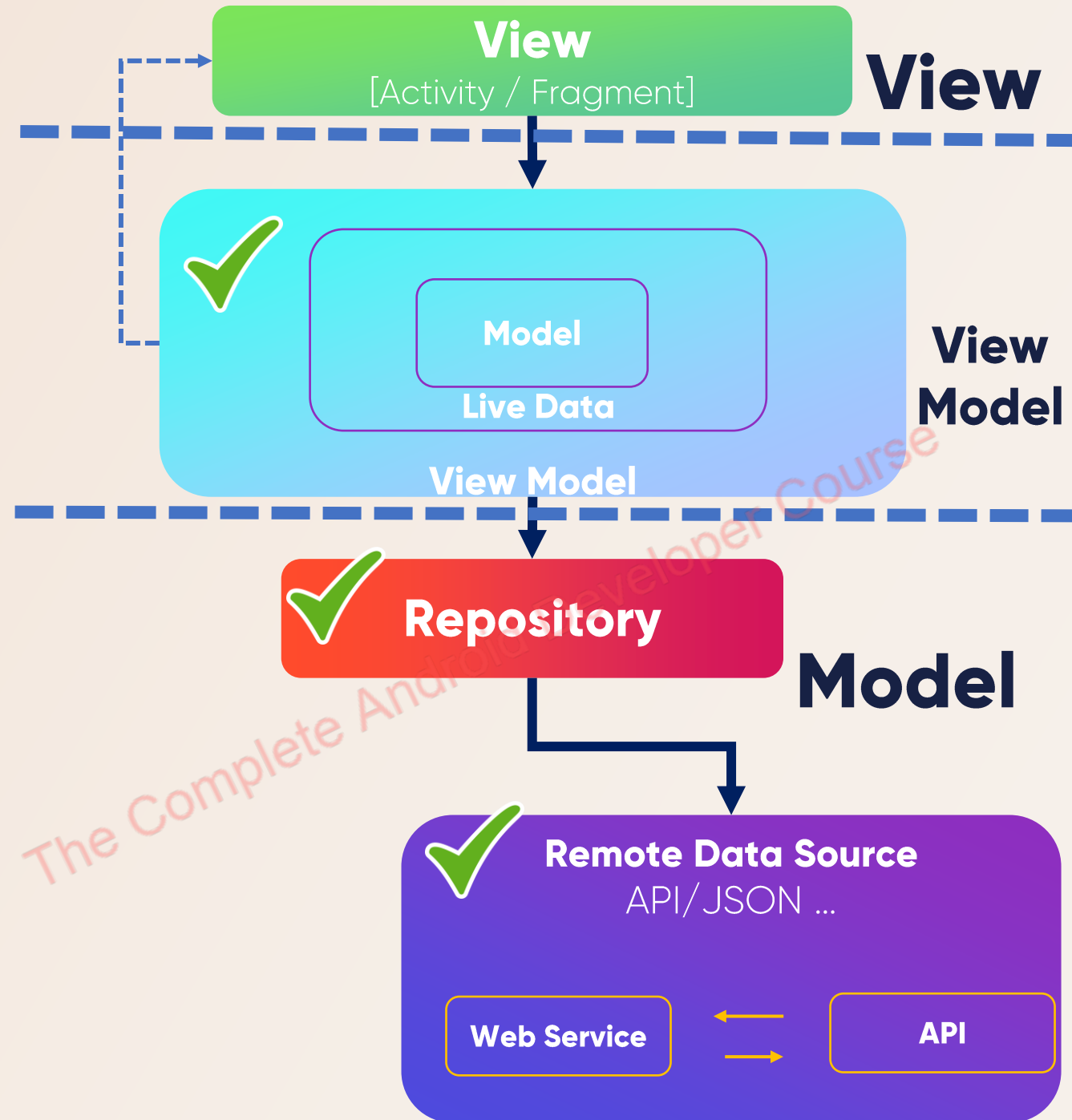
View: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.

ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.

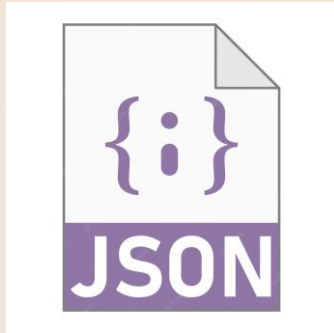


The Quiz App

- Create our local database (MYSQL)
- Create our own API
- Fetch the JSON response from database
- Using Retrofit to create Java Objects
- Display questions in Android App
- Follow MVVM Architecture



Our Own API



```
"results": [  
  {  
    "popularity": 167.729,  
    "vote_count": 3823,  
    "video": false,  
    "poster_path": "/xBHvZcjRiWyobQ9kx8h06B2dtRI.jpg",  
    "id": 419704,  
    "adult": false,  
    "backdrop_path": "/5BwqwxMEjeFtdknRV792Svo0K1v.jpg",  
    "original_language": "en",  
    "original_title": "Ad Astra",  
    "genre_ids": [  
      18,  
      878  
    ],  
    "title": "Ad Astra",  
    "vote_average": 6.1,  
    "overview": "The near future, a time when both hope  
    "release_date": "2019-09-17"  
  },  
]
```



App UI

The Complete Android Developer Course

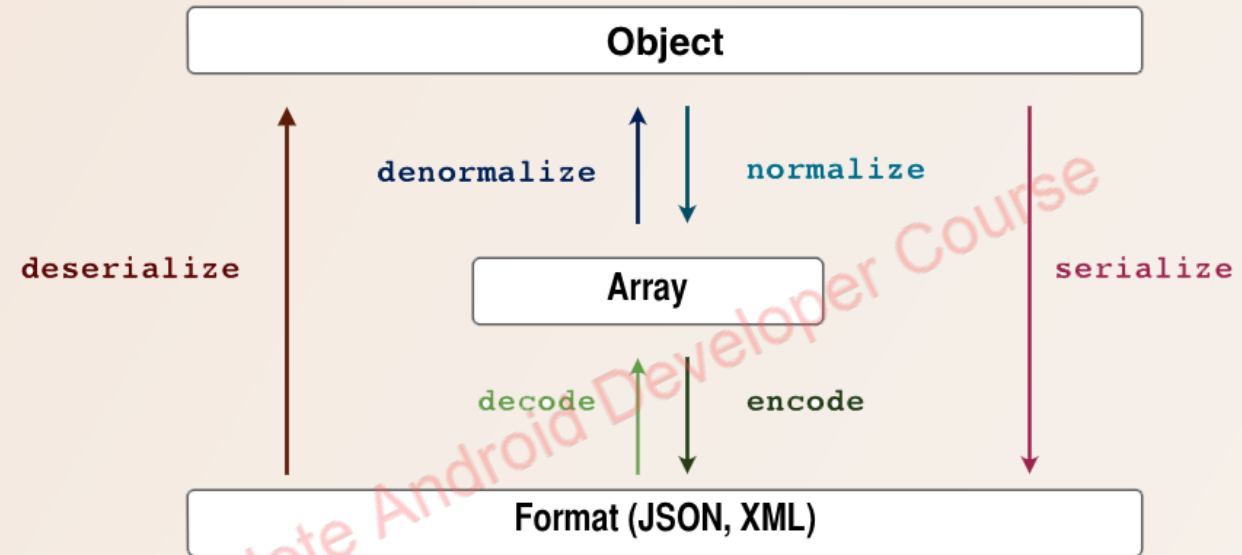
GSON CONVERTER

A Converter which uses Gson for serialization to and from JSON.

Used to convert Java objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.

For this purpose, Gson provides several built in serializers and deserializers. A serializer allows to convert a Json string to corresponding Java type. A deserializers allows to convert from Java to a JSON representation.

A default Gson instance will be created or one can be configured and passed to the GsonConverterFactory to further control the serialization.



USING RETROFIT



API/Service Interface

We need to create an Interface to define our different methods that will be used for network transactions.

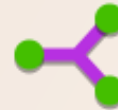


Data Classes: POJO

For fetching response we need create POJO class that automatically parse the JSON data using Gson in background. We just need to create this POJO class.

For creating POJO class first method is defining each and every variable by ourself and other best and fast method is using <http://www.jsonschema2pojo.org/> platform.

To serialize JSON we require a converter.



Retrofit Instance

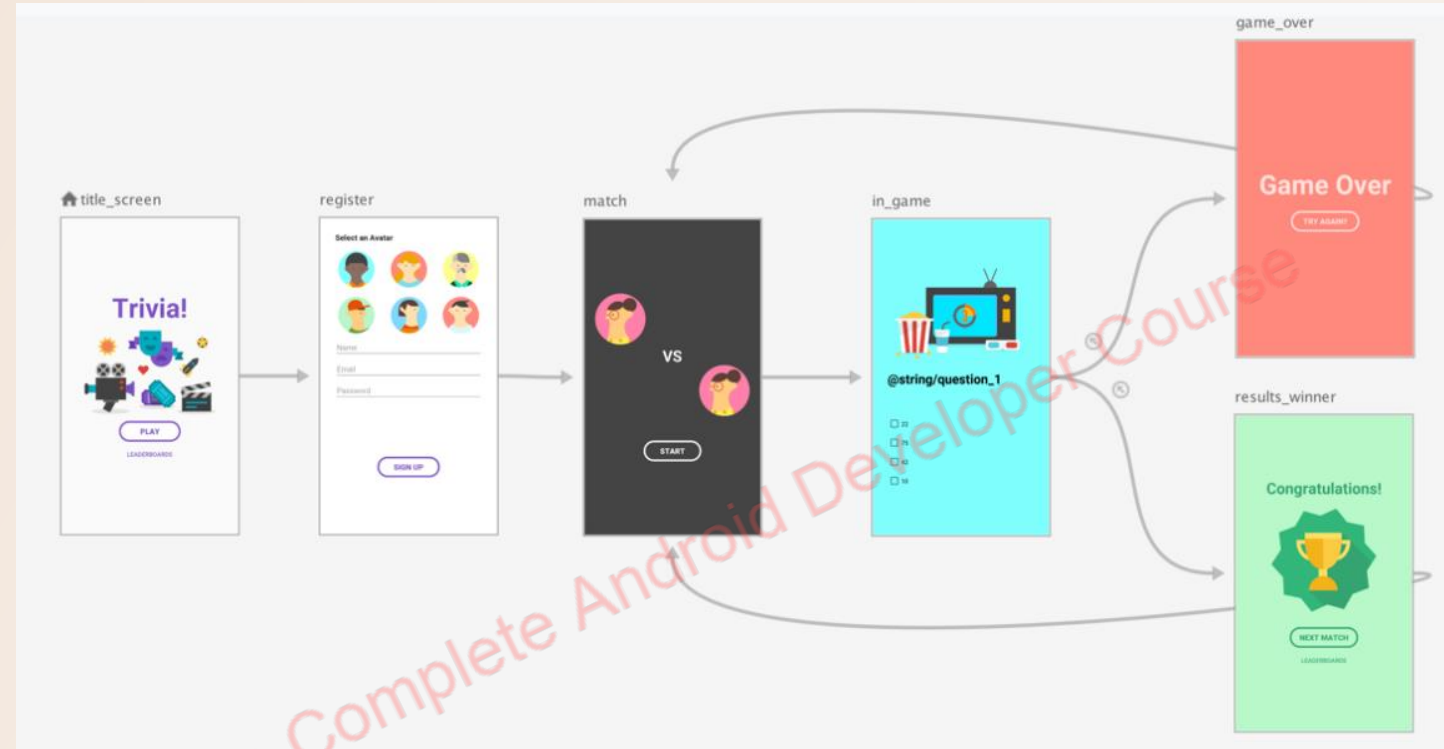
since we are using Retrofit for Network calls, let's create a class that provides us the instance of the Retrofit.

The Complete Android Developer Course

NAVIGATION

The Navigation Architecture Component simplifies implementing navigation, while also helping you visualize your app's navigation flow. The library provides several benefits, including:

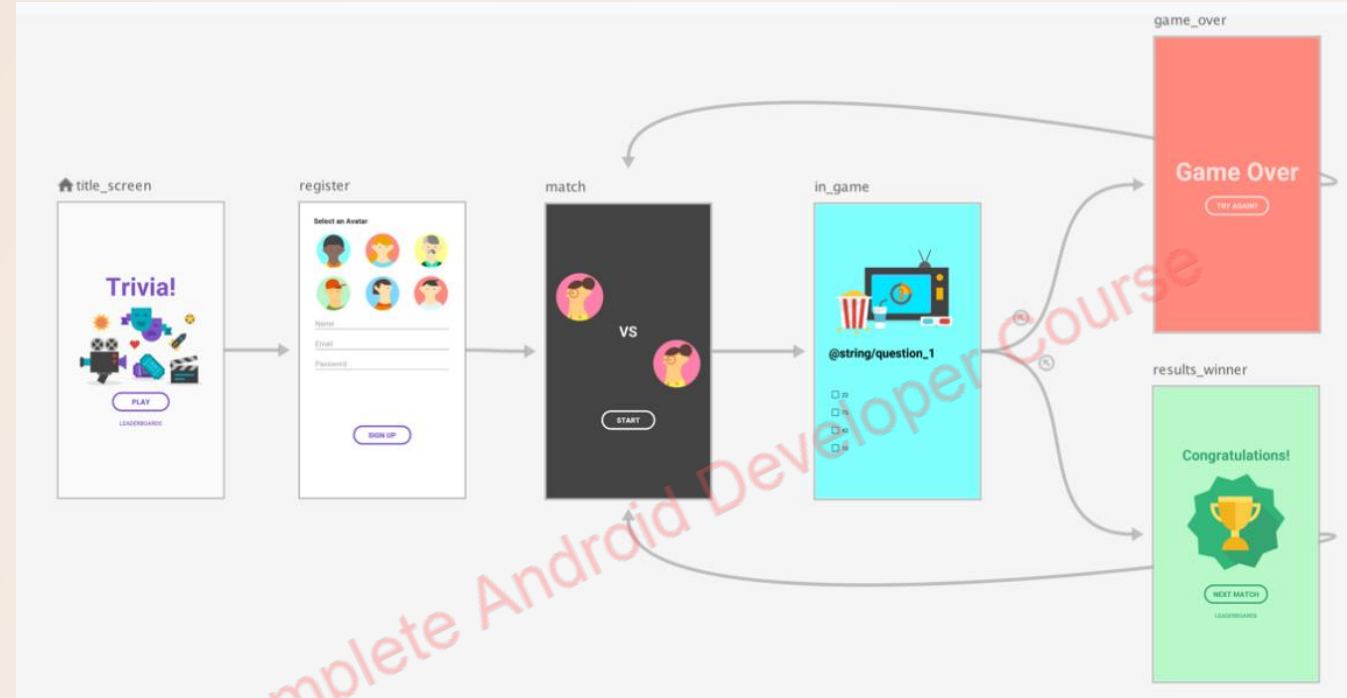
- Automatic handling of fragment transactions
- Correctly handling up and back by default
- Default behaviors for animations and transitions
- Deep linking as a first-class operation
- Implementing navigation UI patterns (like navigation drawers and bottom nav) with little additional work
- Type safety when passing information while navigating
- Android Studio tooling for visualizing and editing the navigation flow of an app



NAVIGATION PARTS

The Navigation Component consists of three key parts, working together in harmony. They are:

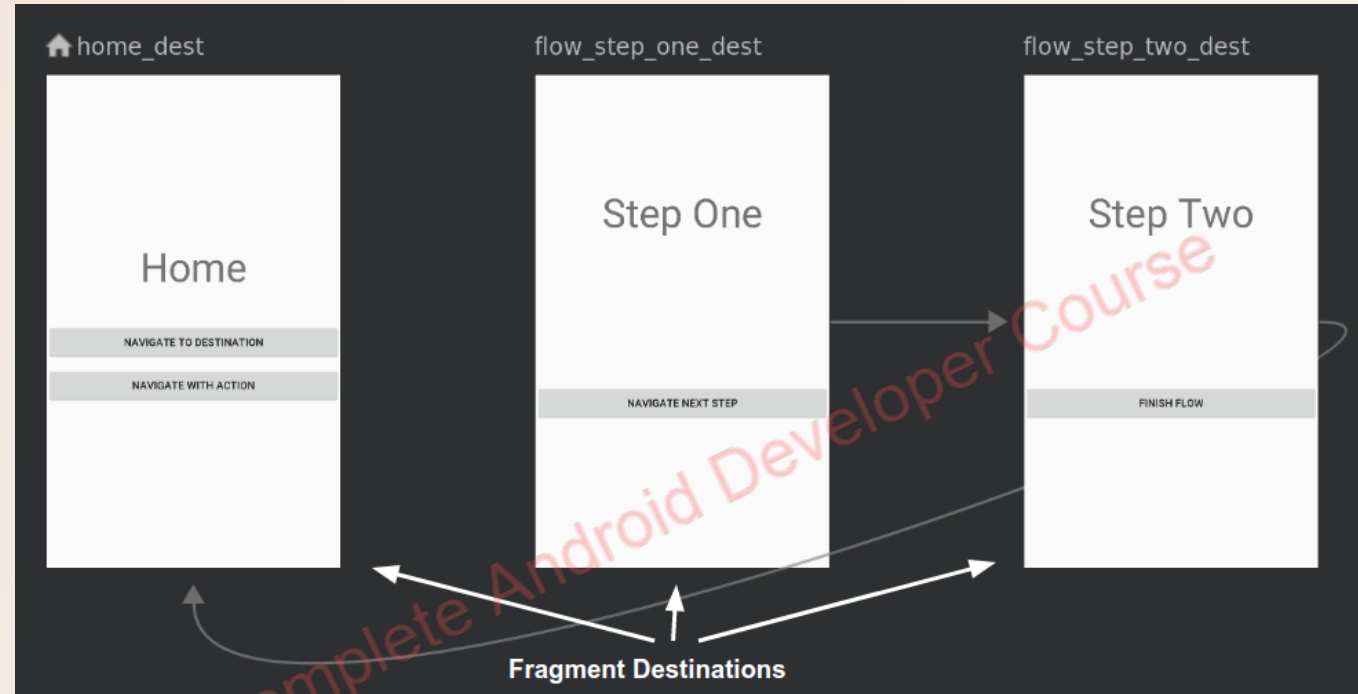
- **Navigation Graph (New XML resource)** – This is a resource that contains all navigation-related information in one centralized location. This includes all the places in your app, known as destinations, and possible paths a user could take through your app.
- **NavHostFragment (Layout XML view)** – This is a special widget you add to your layout. It displays different destinations from your Navigation Graph.
- **NavController (Kotlin/Java object)** – This is an object that keeps track of the current position within the navigation graph. It orchestrates swapping destination content in the NavHostFragment as you move through a navigation graph.



NAVIGATION GRAPH

A navigation graph is a new resource type that defines all the possible paths a user can take through an app. It shows visually all the destinations that can be reached from a given destination.

Android Studio displays the graph in its Navigation Editor.



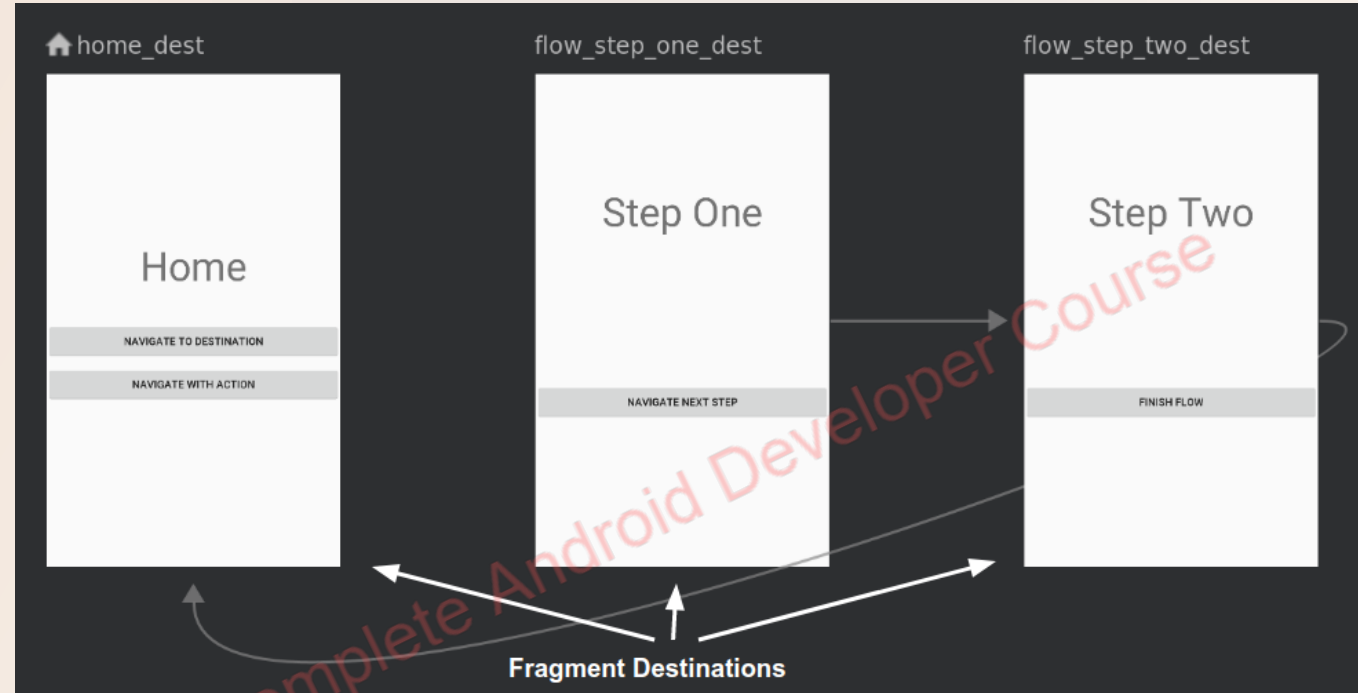
NAV HOST FRAGMENT

NavHostFragment provides an area within your layout for self-contained navigation to occur.

NavHostFragment is intended to be used as the content area within a layout resource defining your app's chrome around it.

Each NavHostFragment has a NavController that defines valid navigation within the navigation host.

This includes the NavGraph as well as navigation state such as current location and back stack that will be saved and restored along with the NavHostFragment itself.



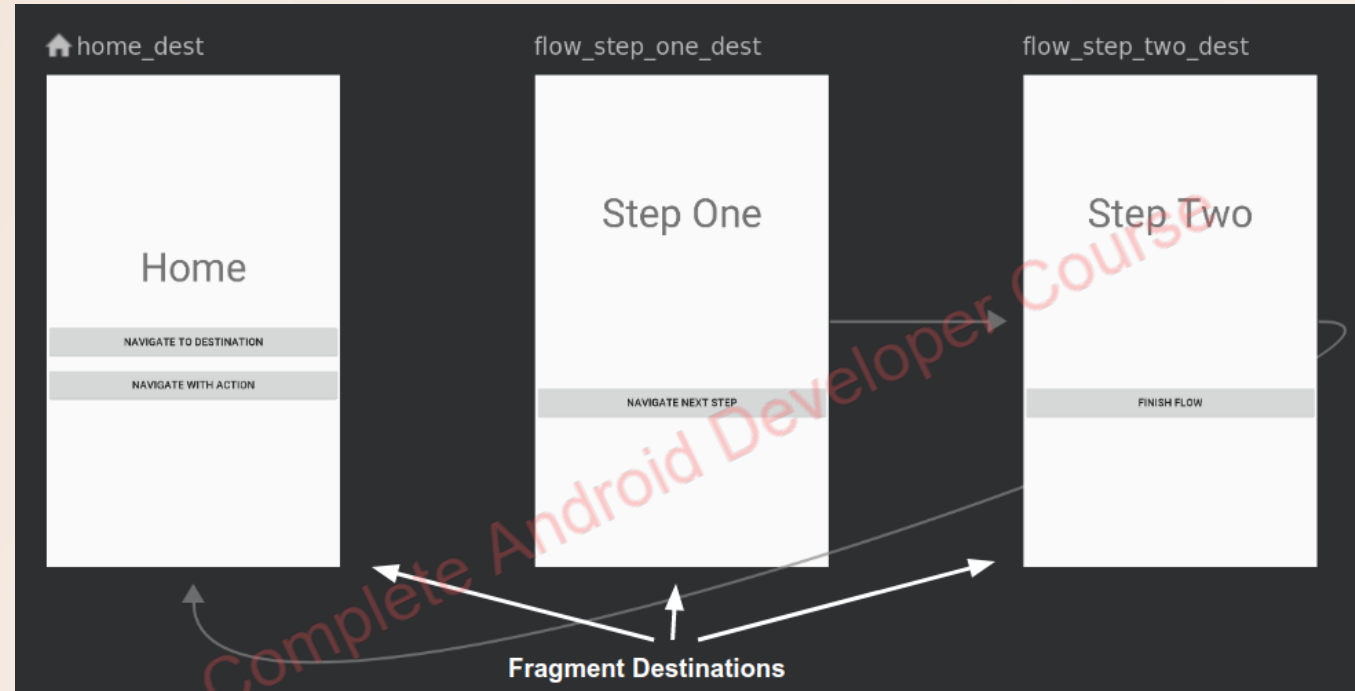
NAV CONTROLLER

Navigating to a destination is done using a NavController, an object that manages app navigation within a NavHost.

Each NavHost has its own corresponding NavController. You can retrieve a NavController by using one of the following methods:

- `Fragment.findNavController()`
- `View.findNavController()`
- `Activity.findNavController(viewId: Int)`

Actions: The routes user can take between your app's destinations. Which are represented by the arrow sign in the Design view.



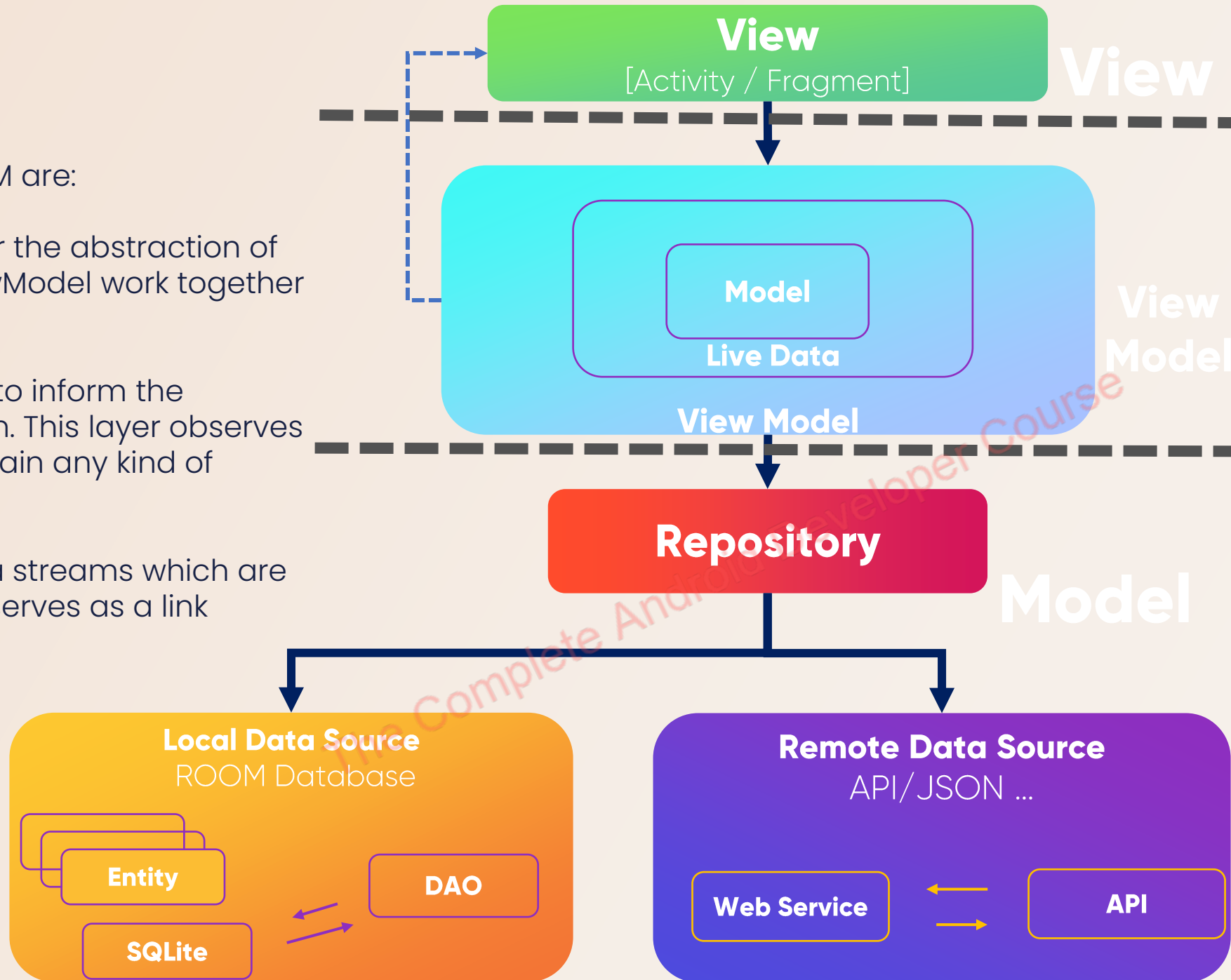
MVVM Layers

The separate code layers of MVVM are:

Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

View: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.

ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.



MVVM Layers

The separate code layers of MVVM are:

Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

View: The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.

ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.

