Virtual Internship - Android Application Development Using Kotlin

Category: Kotlin

Skills Required:
Android App

Project Description:

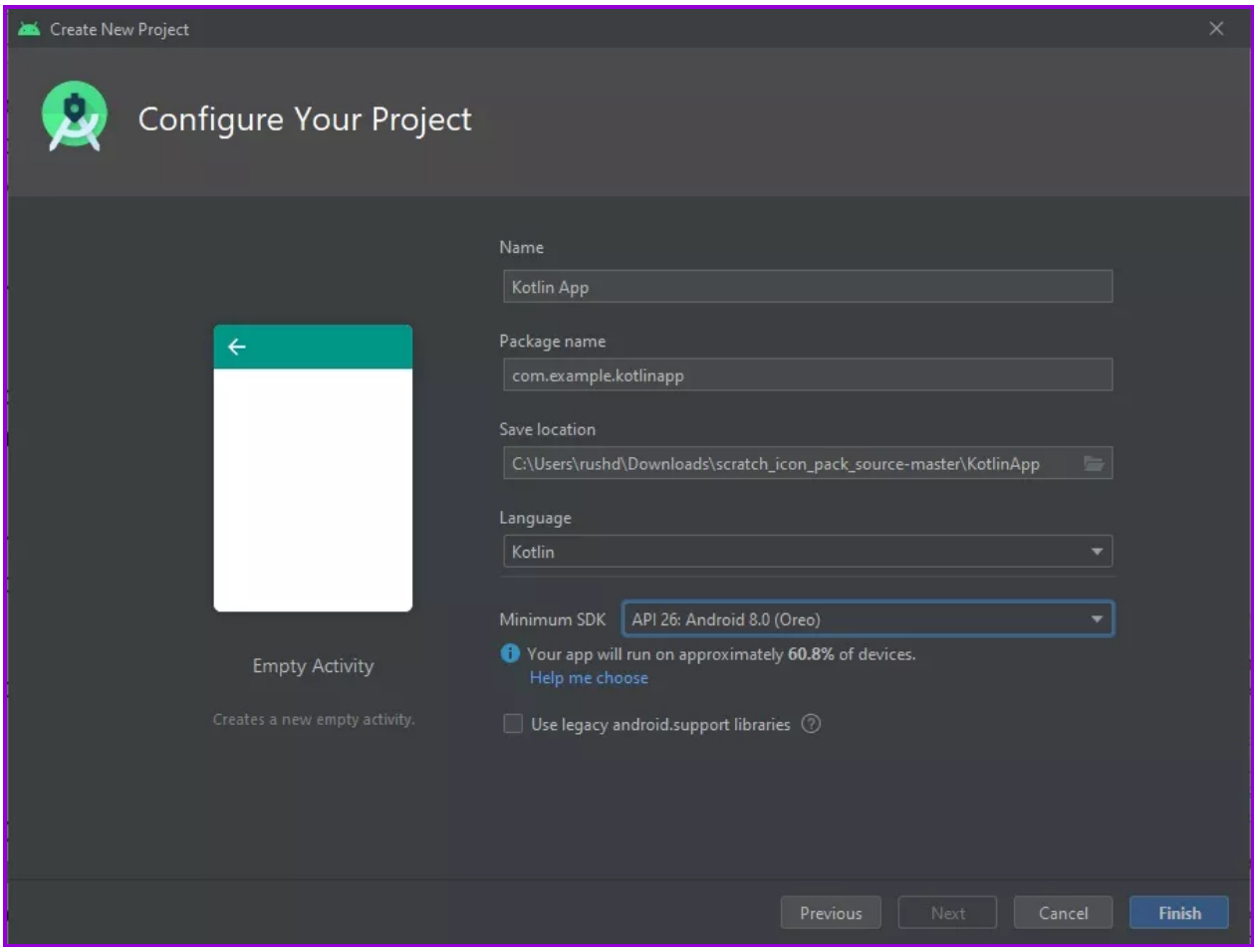
As demand for skilled Android developers increases in the job marketplace, there is an greater need for the next generation to develop the skills of Android developement.

This program will give to complete hands-on experience on android application development using Kotlin also you will develop an application on your own.

ANDROID BASICS IN KOTLIN

Unit 1: Kotlin basics

Build your first Android apps with the Kotlin programming language. Add images and text to your Android apps, and learn how to use classes, objects, and conditionals to create an interactive app for your users.



short program in Kotlin using functions and loops to print a happy birthday message.

```
fun main() {
    val age = 24
    val layers = 5
    printCakeCandles(age)
    printCakeTop(age)
    printCakeBottom(age, layers)
}

fun printCakeCandles(age: Int) {
    print (" ")
    repeat(age) {
        print(",")
    }
    println() // Print an empty line
```

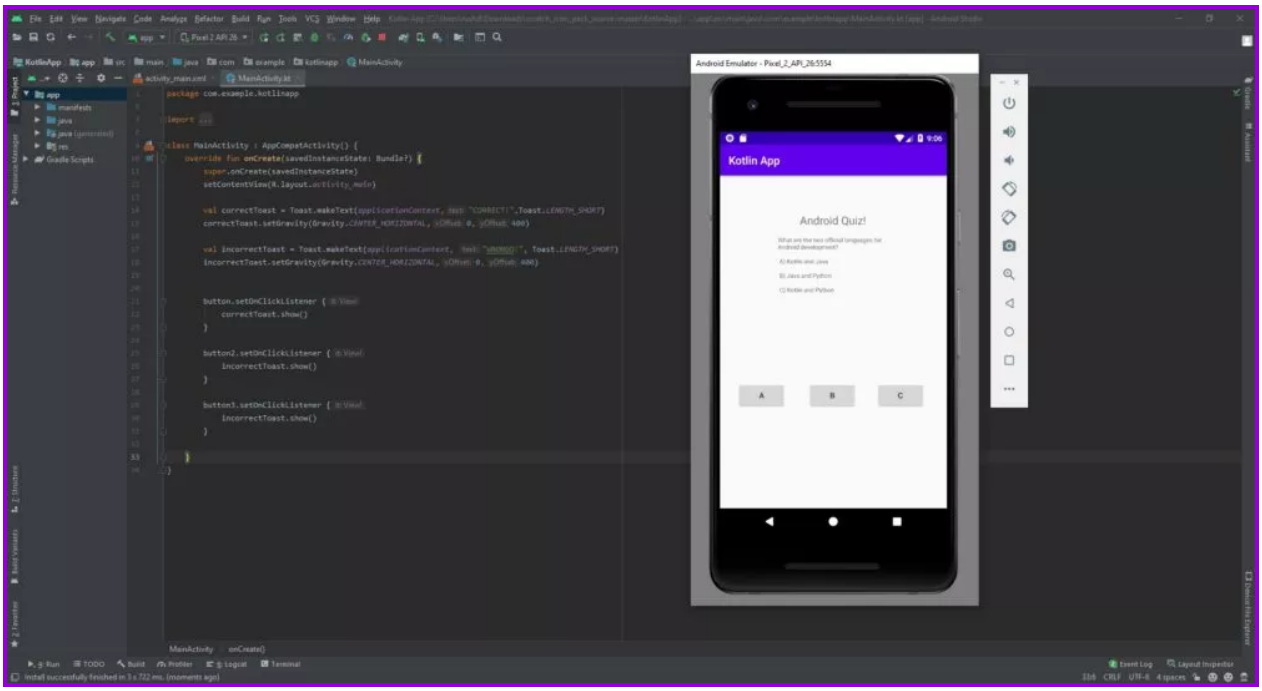
```
    print(" ") // Print the inset of the candles on the cake
    repeat(age) {
        print("|")
    }
    println()
}

fun printCakeTop(age: Int) {
    repeat(age + 2) {
        print("=")
    }
    println()
}

fun printCakeBottom(age: Int, layers: Int) {
    repeat(layers) {
        repeat(age + 2) {
            print("@")
        }
        println()
    }
}
```

- **Explain:** Use `${}` to surround variables and calculations in the text of print statements. For example: `${age}` where `age` is a variable.
- Create a variable using the `val` keyword and a name. Once set, this value cannot be changed. Assign a value to a variable using the equal sign. Examples of values are text and numbers.
- A `String` is text surrounded by quotes, such as `"Hello"`.
- An `Int` is a whole positive or negative number, such as 0, 23, or -1024.
- You can pass one or more arguments into a function for the function to use, for example: `fun printCakeBottom(age:Int, layers:Int) {}`
- Use a `repeat() {}` statement to repeat a set of instructions several times. For example: `repeat (23) { print("%") }` or `repeat (layers) { print("@@@@@@@@@@") }`
- A *loop* is an instruction to repeat instructions multiple times. A `repeat()` statement is an example of a loop.

- You can nest loops, that is, put loops within loops. For example, you can create a `repeat()` statement within a `repeat()` statement to print a symbol a number of times for a number of rows, like you did for the cake layers.

Summary of using function arguments: To use arguments with a function, you need to do three things:

- Add the argument and type to the function definition: `printBorder(border: String)`
- Use the argument inside the function: `println(border)`
- Supply the argument when you call the function: `printBorder(border)`



Unit 2: Layouts

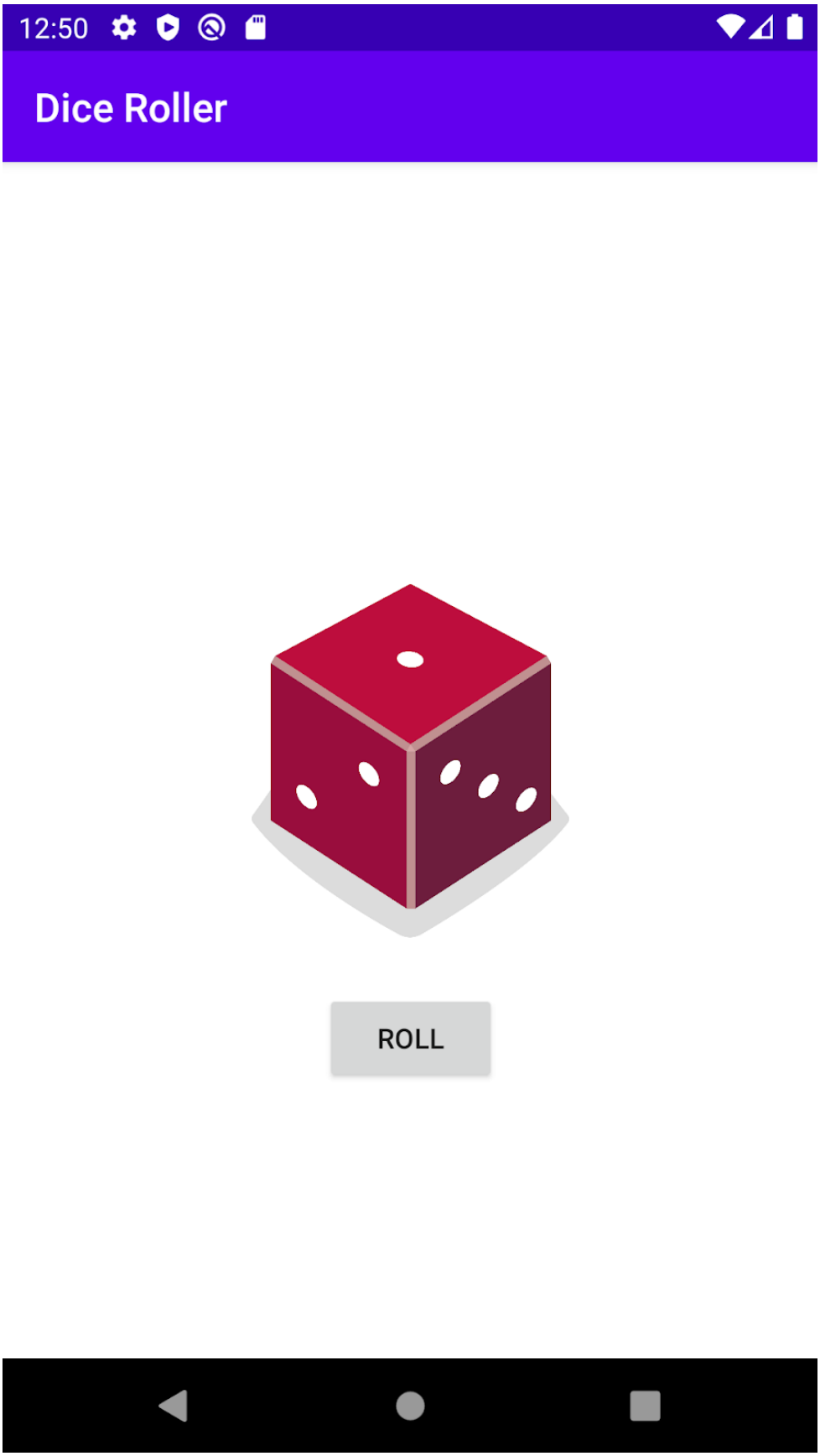
Improve the user interface of your app by learning about layouts, Material Design guidelines, and best practices for UI development.

Add images to the Dice Roller app

```
package
com.example.dicero
ller

                import android.os.Bundle
```

```
import android.widget.Button
import android.widget.ImageView
import androidx.appcompat.app.AppCompatActivity

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

    /**
     * This method is called when the Activity is created.
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Find the Button in the layout
        val rollButton: Button = findViewById(R.id.button)

        // Set a click listener on the button to roll the dice when the user taps the butt
        rollButton.setOnClickListener { rollDice() }

        // Do a dice roll when the app starts
        rollDice()
    }

    /**
     * Roll the dice and update the screen with the result.
     */
    private fun rollDice() {
        // Create new Dice object with 6 sides and roll it
        val dice = Dice(6)
        val diceRoll = dice.roll()

        // Find the ImageView in the layout
        val diceImage: ImageView = findViewById(R.id.imageView)

        // Determine which drawable resource ID to use based on the dice roll
        val drawableResource = when (diceRoll) {
            1 -> R.drawable.dice_1
            2 -> R.drawable.dice_2
            3 -> R.drawable.dice_3
```

```
            4 -> R.drawable.dice_4
            5 -> R.drawable.dice_5
            else -> R.drawable.dice_6
        }

        // Update the ImageView with the correct drawable resource ID
        diceImage.setImageResource(drawableResource)

        // Update the content description
        diceImage.contentDescription = diceRoll.toString()
    }
}

/**
 * Dice with a fixed number of sides.
 */
class Dice(private val numSides: Int) {

    /**
     * Do a random dice roll and return the result.
     */
    fun roll(): Int {
        return (1..numSides).random()
    }
}
```


12:50
Dice Roller
ROLL

Project: Lemonade App - Starter Code

Starter code for the first independent project for Android Basics in Kotlin

Introduction

This is the starter code for the Lemonade app project in the final pathway of Android Basics Unit 1. This project is an opportunity for you to demonstrate the concepts you learned in the unit.

Getting Started

1. Download the starter code
2. Open the project in Android Studio
3. Complete the project in accordance with the project instructions

Tips

- Use the provided tests to ensure your app is running as expected

DO NOT ALTER THE PROVIDED TESTS

Layouts

Overview

This Instructable will show you how to create a basic application using Android Studio. The application I will be showing you how to make is a tip calculator. The calculator will be fairly simple to create and use. You simply type in the amount, click on the desired tip value, and the total tip for the amount will be displayed.

Difficulty

Beginner and intermediate

WARNING!

Make sure to download only the approved and certified Android Studio software from Google. Any other source can contain viruses and malware that could potentially harm your computer.

Materials Needed

- A computer that can support Android Studio
- Android Studio
- Android Phone (optional)
- Micro-USB cable (optional)

Dogglers - Starter Code

Starter code for the second independent project for Android Basics in Kotlin.

Introduction

This is the starter code for the Dogglers app project in the final pathway of Android Basics Unit 2. This project is an opportunity for you to demonstrate the concepts you learned in the unit.

Pre-requisites

- Complete Unit 2 of Android Basics in Kotlin

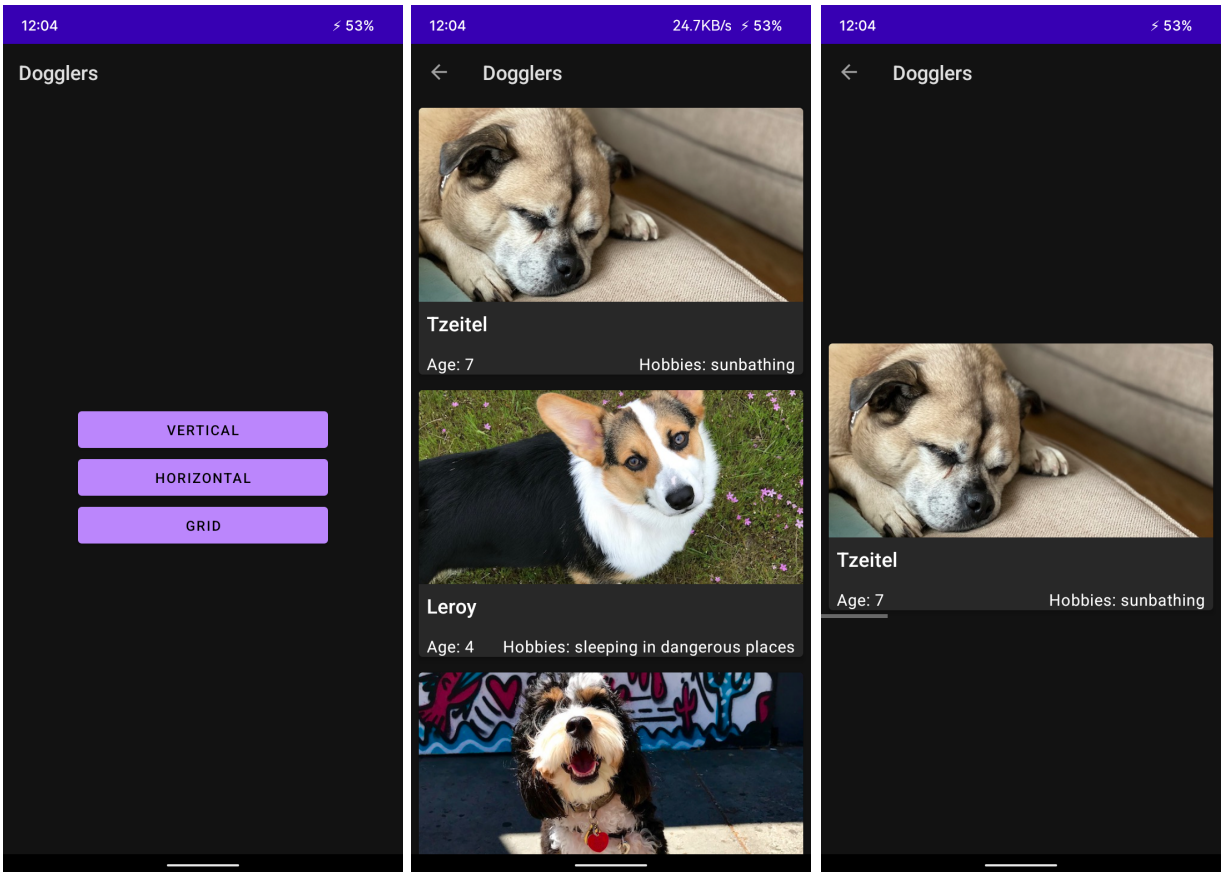
Getting Started

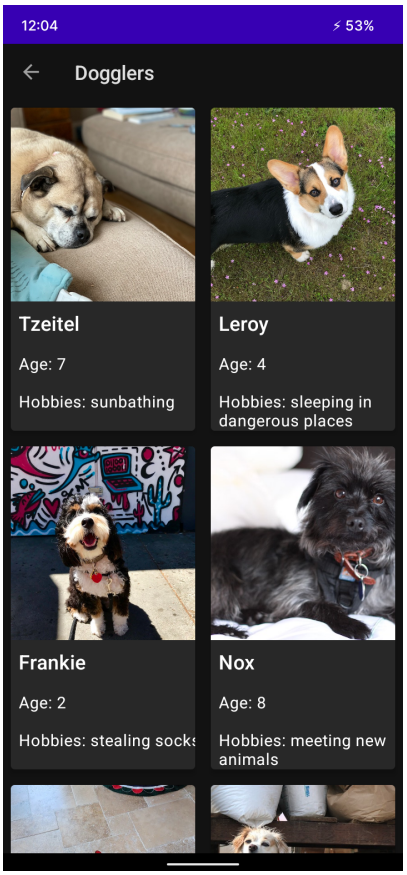
1. Download the starter code
2. Open the project in Android Studio
3. Complete the project in accordance with the project instructions

Tips

- Use the provided tests to ensure your app is running as expected
- DO NOT ALTER THE PROVIDED TESTS

Output Screenshots





Connect To The Internet

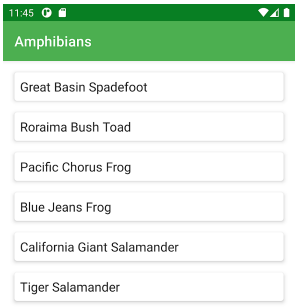
Introduction to coroutines

```
import kotlinx.coroutines.*

fun main() {
  val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
  repeat(3) {
    GlobalScope.launch {
      println("${Thread.currentThread()} has started")
      for (i in states) {
        println("${Thread.currentThread()} - $i")
      }
    }
  }
}
```

- Why concurrency is needed
- What a thread is, and why threads are important for concurrency
- How to write concurrent code in Kotlin using coroutines
- When and when not to mark a function as "suspend"
- The roles of a CoroutineScope, Job, and Dispatcher
- The difference between Deferred and Await

Project: Amphibians app

Take an app that displays information about different amphibian species, and use your knowledge of networking, JSON parsing, and view models to enable the app to use data from the network. The app will get its data from a custom API for this project and display it in a list view.

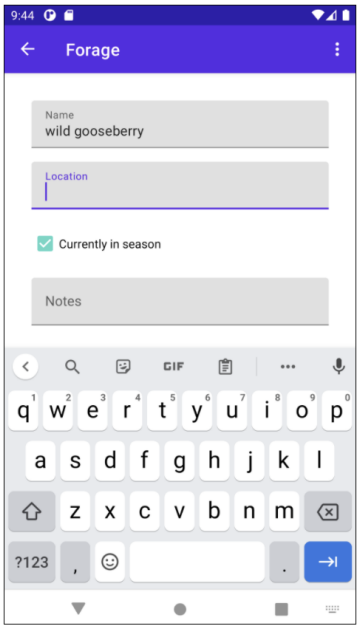


USE ROOM FOR DATA PERSISTENCE

Project: Forage app

The completed Forage app allows users to keep track of items, food for example, that they've foraged for in nature. This data is persisted between sessions using Room. You'll use your knowledge of Room and performing read, write, update, and delete

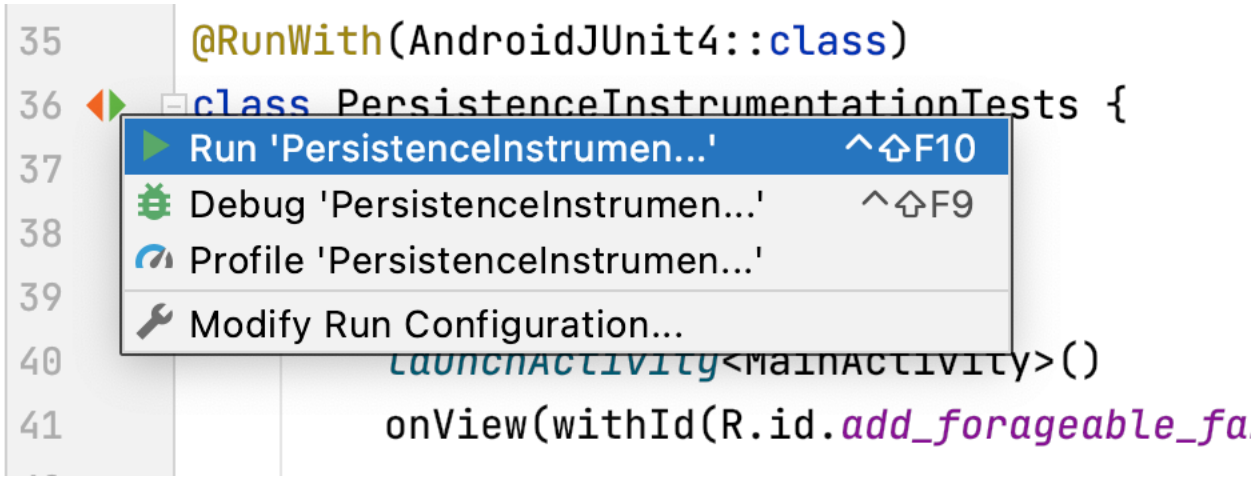
operations on a database to implement persistence in the Forage app.



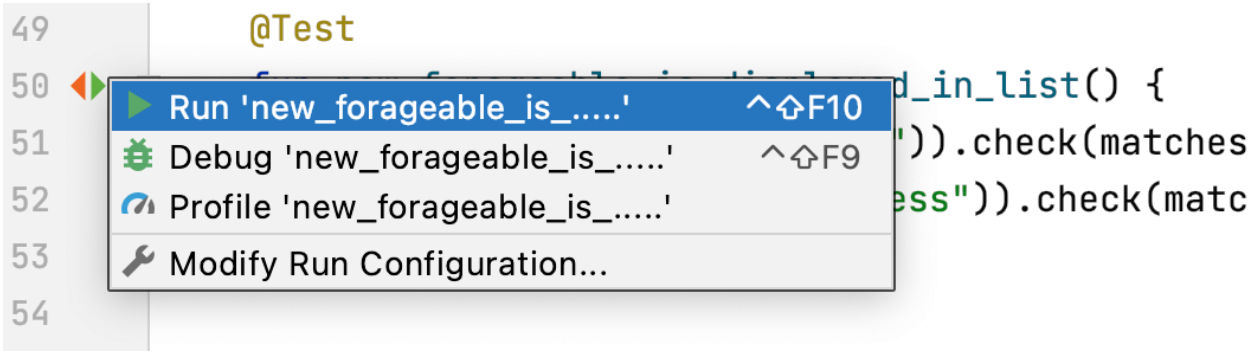
Running your tests

To run your tests, you can do one of the following.

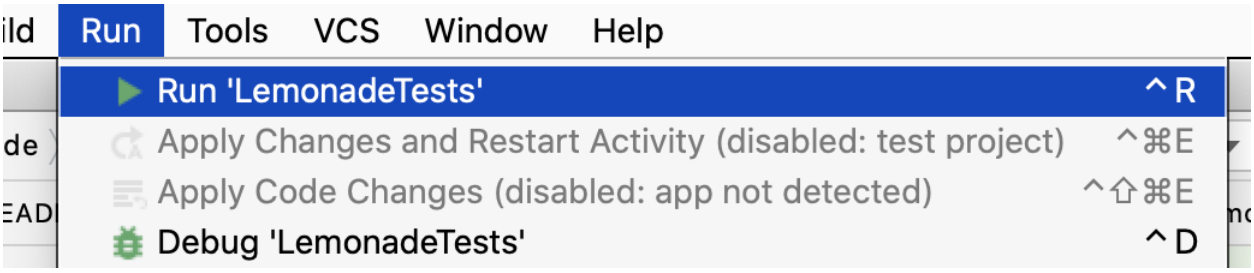
For a single test case, open up a test case class, PersistenceInstrumentationTests.kt and click the green arrow to the left of the class declaration. You can then select the Run option from the menu. This will run all of the tests in the test case.



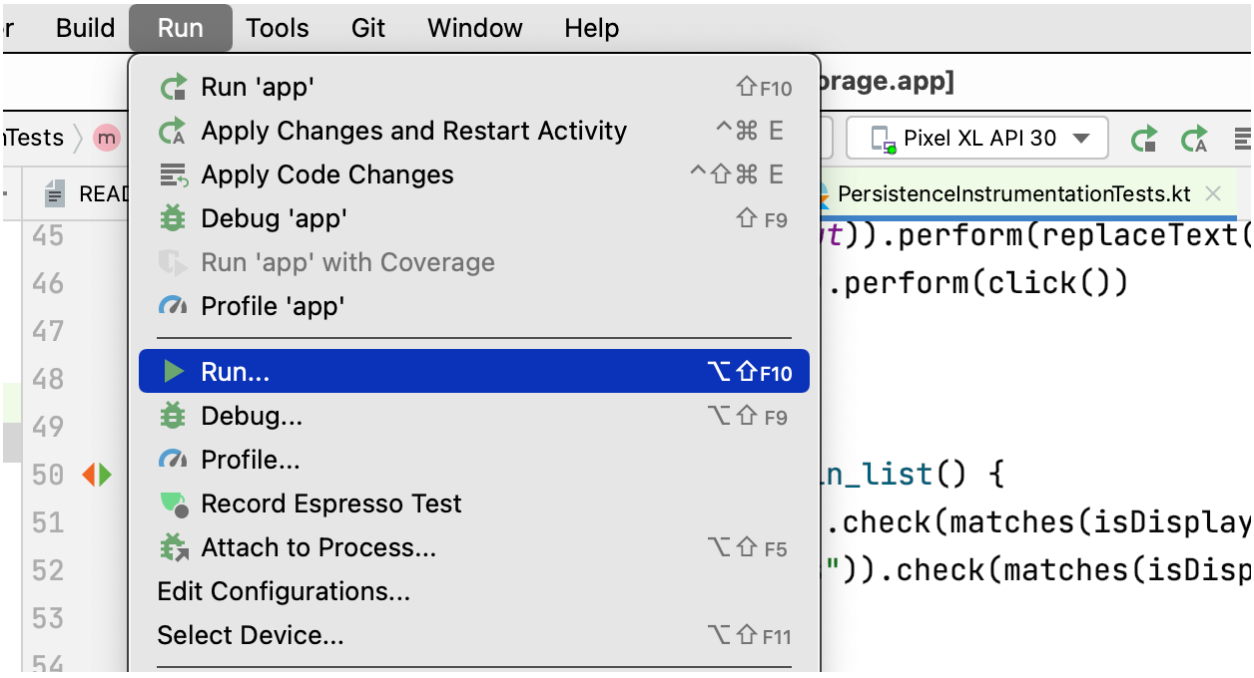
Often you'll only want to run a single test, for example, if there's only one failing test and the other tests pass. You can run a single test just as you would the entire test case. Use the green arrow and select the **Run** option.



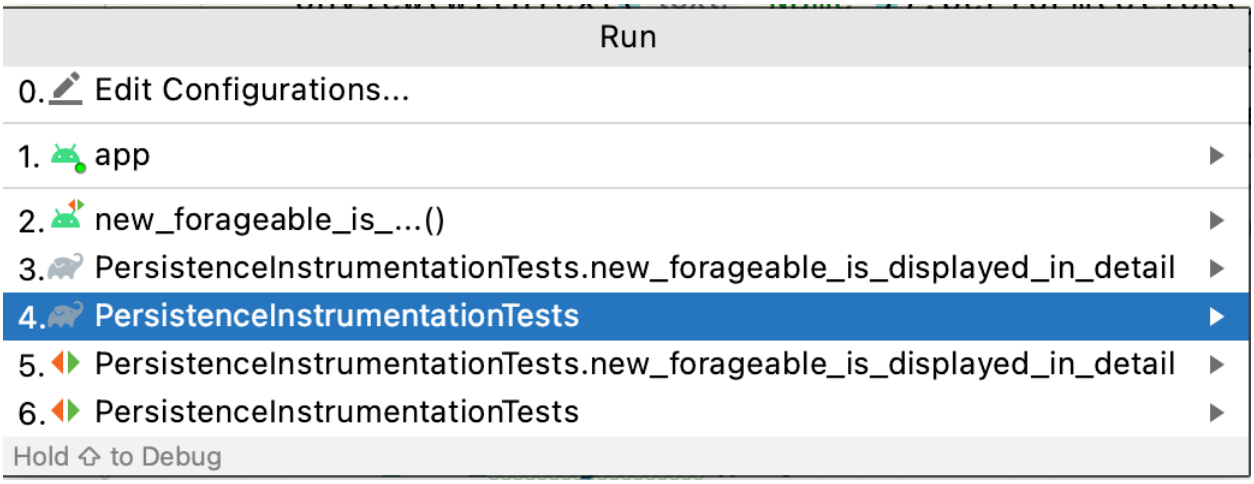
If you have multiple test cases, you can also run the entire test suite. Just like running the app, you can find this option on the **Run** menu.



Note that Android Studio will default to the last target that you ran (app, test targets, etc.) so if the menu still says **Run > Run 'app'**, you can run the test target, by selecting **Run > Run**.



Then choose the test target from the popup menu.



.gitignore file

```
# built application
files
                        *.apk
                        *.ap_
                        # Mac files
                        .DS_Store
                        # files for the dex VM
                        *.dex
                        # Java class files
                        *.class
                        # generated files
                        bin/
                        gen/
                        # Ignore gradle files
                        .gradle/
                        build/
                        # Local configuration file (sdk path,
                        etc)
                        local.properties
                        # Proguard folder generated by Eclipse
                        proguard/
                        proguard-project.txt
                        # Eclipse files
                        .project
                        .classpath
                        .settings/
                        # Android Studio/IDEA
```

```
*.iml
.idea
```

ANDROID BASICS IN KOTLIN

Unit 6: WorkManager

Use Android Jetpack's WorkManager API to schedule necessary background work, like backing up data or downloading fresh content, that keeps running even if the app exits or the device restarts.

Project: Water Me! app

The Water Me! app consists of a list of plants, some information about them, and a description for how often each one should be watered. For each of these plants, the completed app will schedule a reminder for when they should be watered.

All the functionality for the Water Me! app is already implemented, except for the part to schedule and a notification. The code for displaying a notification is in `WaterReminderWorker.kt` (in the **worker** package). This happens in the `doWork()` method of a custom `Worker` class. Because notifications may be a new topic, this code is already implemented.

```
override fun doWork(): Result {
    val intent = Intent(applicationContext, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_NEW_TASK or
Intent.FLAG_ACTIVITY_CLEAR_TASK
    }

    val pendingIntent: PendingIntent = PendingIntent
        .getActivity(applicationContext, 0, intent, 0)

    val plantName = inputData.getString(nameKey)

    val builder = NotificationCompat.Builder(applicationContext,
BaseApplication.CHANNEL_ID)
        .setSmallIcon(R.drawable.ic_android_black_24dp)
        .setContentTitle("Water me!")
        .setContentText("It's time to water your $plantName")
        .setPriority(NotificationCompat.PRIORITY_HIGH)
```

```
        .setContentIntent(pendingIntent)
        .setAutoCancel(true)

    with(NotificationManagerCompat.from(applicationContext)) {
        notify(notificationId, builder.build())
    }

    return Result.success()
}
```

To use WorkManager, an API that handles background work that needs to run regardless of whether the application process is still running.

COMPLETED ALL SELF LEARNING COURSES :)