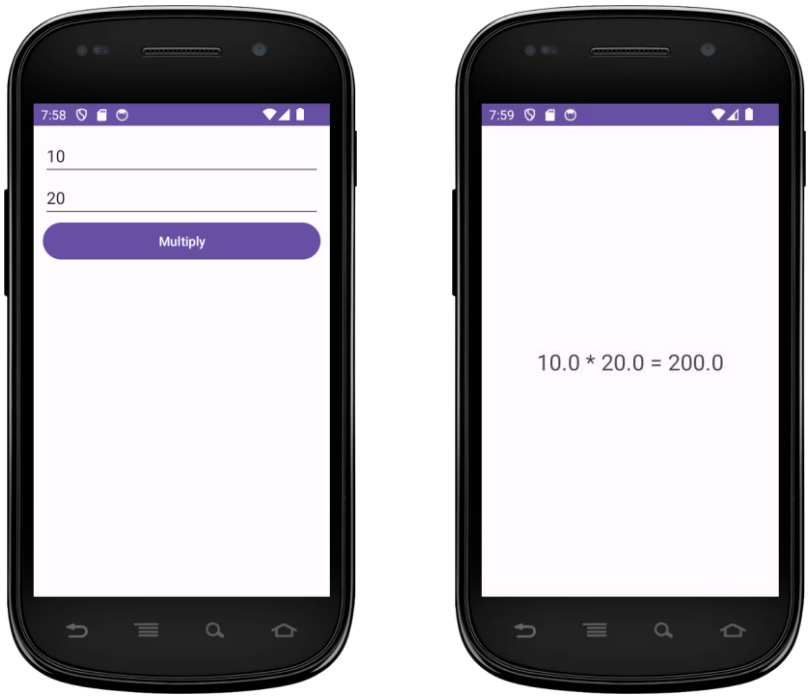
Bài thực hành Fragment – Nhân 2 số

TS. Nguyễn Hồng Quang, Khoa Kỹ thuật máy tính, Trường Công nghệ thông tin và Truyền thông, Đại học Bách Khoa Hà Nội

Đề bài

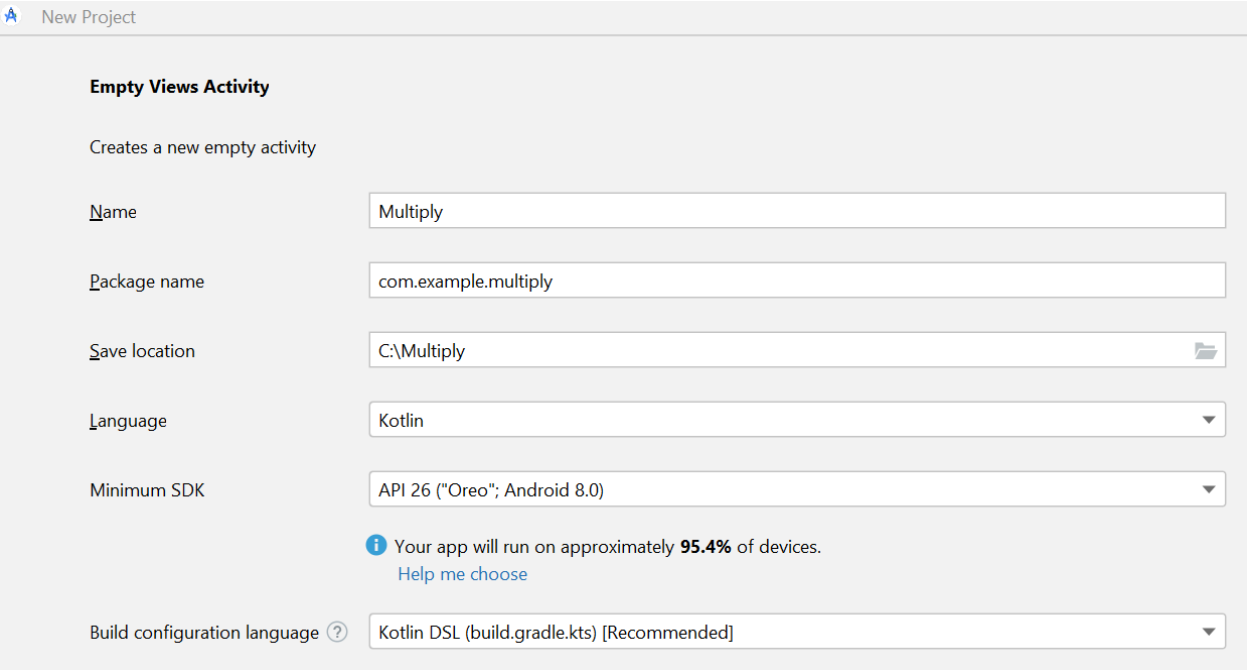
Tạo ứng dụng gồm 2 Fragment: InputFragment và MultiplyFragment. InputFragment gồm 2 ô EditText và một Button "Multiply". Khi ấn vào nút này thì sẽ gửi 2 số sang MultiplyFragment để hiển thị kết quả nhân 2 số.



Bước 1. Tạo Project Android mới

Mở Android Studio, chọn New Project:

Trong cửa sổ tiếp theo, chọn như hình vẽ. Lưu ý: trong mục Build configuration language, chọn "Kotlin DSL(build.gradle.kts)".



Lưu ý: chọn Minimum SDK là API 26.

Cập nhật các thư viện cần thiết cho Navigation UI và Safe Args.

Mở file built.gradle.kts của project, thêm dòng sau vào mục plugins:

```
id("androidx.navigation.safeargs") version "2.8.3" apply false
```

Khi đó file này có dạng như sau:

```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
    id("androidx.navigation.safeargs") version "2.8.3" apply false
}
```

Mở file built.gradle.kts trong thư mục app, thêm dòng sau vào mục plugins:

```
id("androidx.navigation.safeargs.kotlin")
```

Khi đó file này có dạng:

```
plugins {
    alias(libs.plugins.android.application)
```

```
    alias(libs.plugins.kotlin.android)
    id("androidx.navigation.safeargs.kotlin")
}
```

Kiểm tra lại compileSdk là 34 (trong mục android)

```
compileSdk = 34
```

Cũng thêm nội dung sau vào mục android để hỗ trợ cơ chế view binding:

```
buildFeatures {
    dataBinding = true
}
```

Bổ sung thêm hai dòng sau vào mục **dependencies**

```
implementation("androidx.navigation:navigation-fragment-ktx:2.8.3")
implementation("androidx.navigation:navigation-ui-ktx:2.8.3")
```

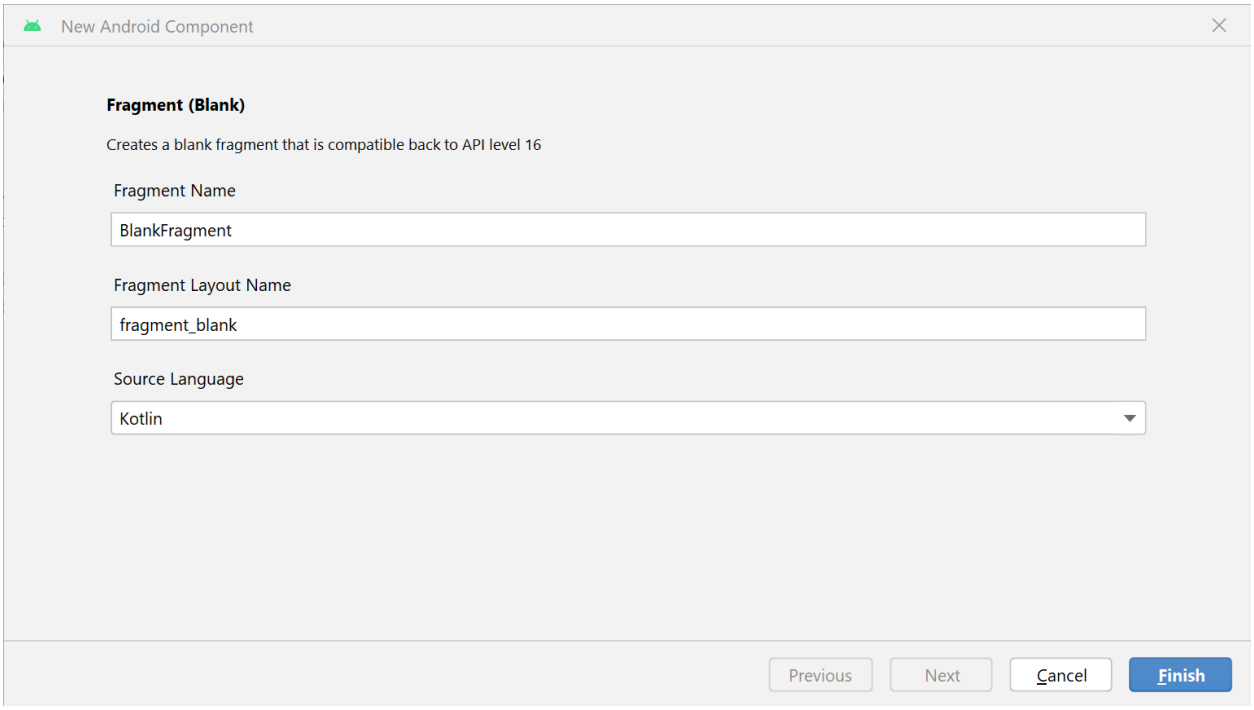
Khi đó mục dependencies có dạng sau:

```
dependencies {
    val core_version = "1.13.1"

    // Java language implementation
    implementation("androidx.core:core:$core_version")
    // Kotlin
    implementation("androidx.core:core-ktx:$core_version")

    //implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.activity)
    implementation(libs.androidx.constraintlayout)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)

    implementation("androidx.navigation:navigation-fragment-ktx:2.8.3")
    implementation("androidx.navigation:navigation-ui-ktx:2.8.3")
}
```

Bước 2. Tạo InputFragment và MultiplyFragment

Tạo InputFragment

Chọn File => New => Fragment => Fragment (Blank)



Nhập Fragment Name: InputFragment

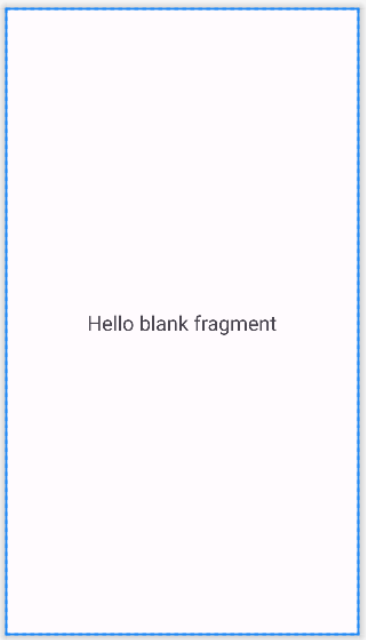
Nhập Fragment Layout Name: fragment_input.xml

Tương tự cho MultiplyFragment với layout: fragment_multiply.xml

Thiết kế giao diện cho fragment_input.xml như sau:


```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="10dp"
    tools:context=".InputFragment">

    <!-- TODO: Update blank fragment layout -->
    <EditText android:id="@+id/number1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="10"
        />
    <EditText android:id="@+id/number2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="20"
        />
    <Button android:id="@+id/bt_multiply"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Multiply"
        />
</LinearLayout>
```

Thiết kế giao diện cho fragment_multiply.xml như sau:




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

    <!-- TODO: Update blank fragment layout -->
    <TextView android:id="@+id/mf_multiply"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_blank_fragment"
        android:textAlignment="center"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

Trong file InputFragment.kt, bổ sung mã cho hàm onCreateView:

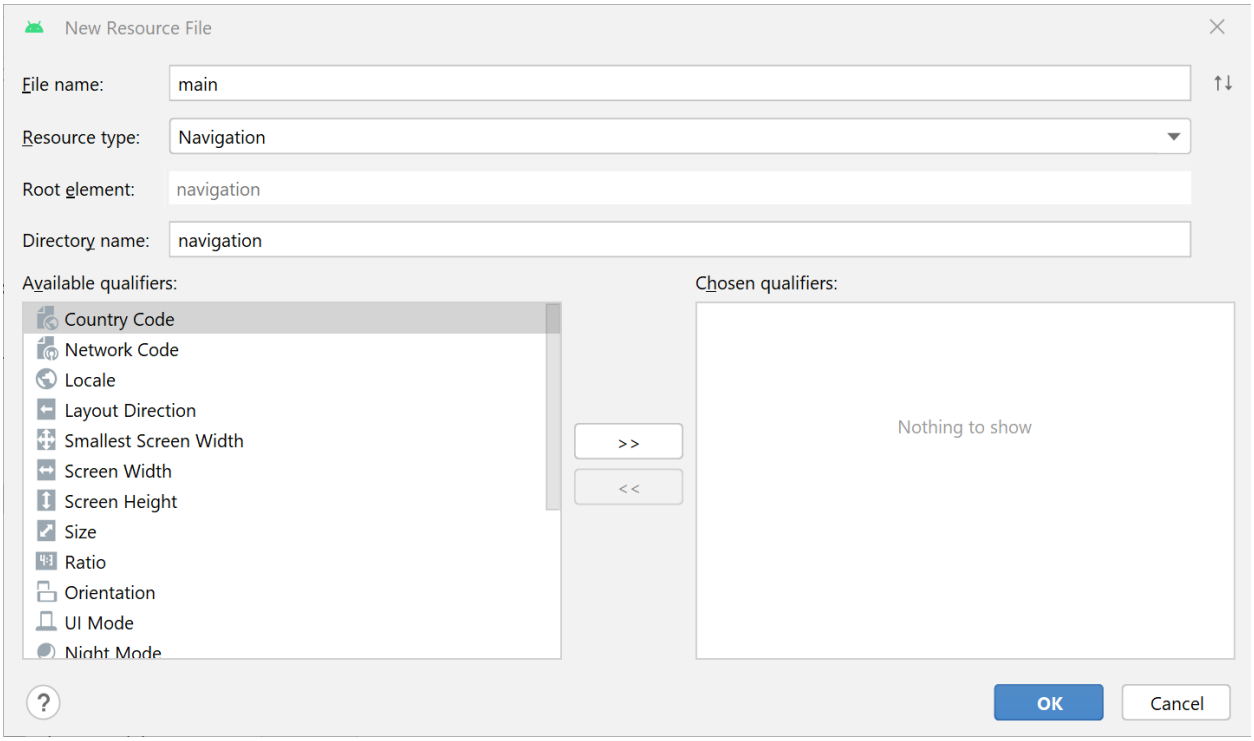
```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    return inflater.inflate(R.layout.fragment_input, container, false)
}
```

Trong file MultiplyFragment.kt, bổ sung mã cho hàm onCreateView:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    return inflater.inflate(R.layout.fragment_multiply, container, false)
}
```

Bước 3. Tạo file Navigation Graph

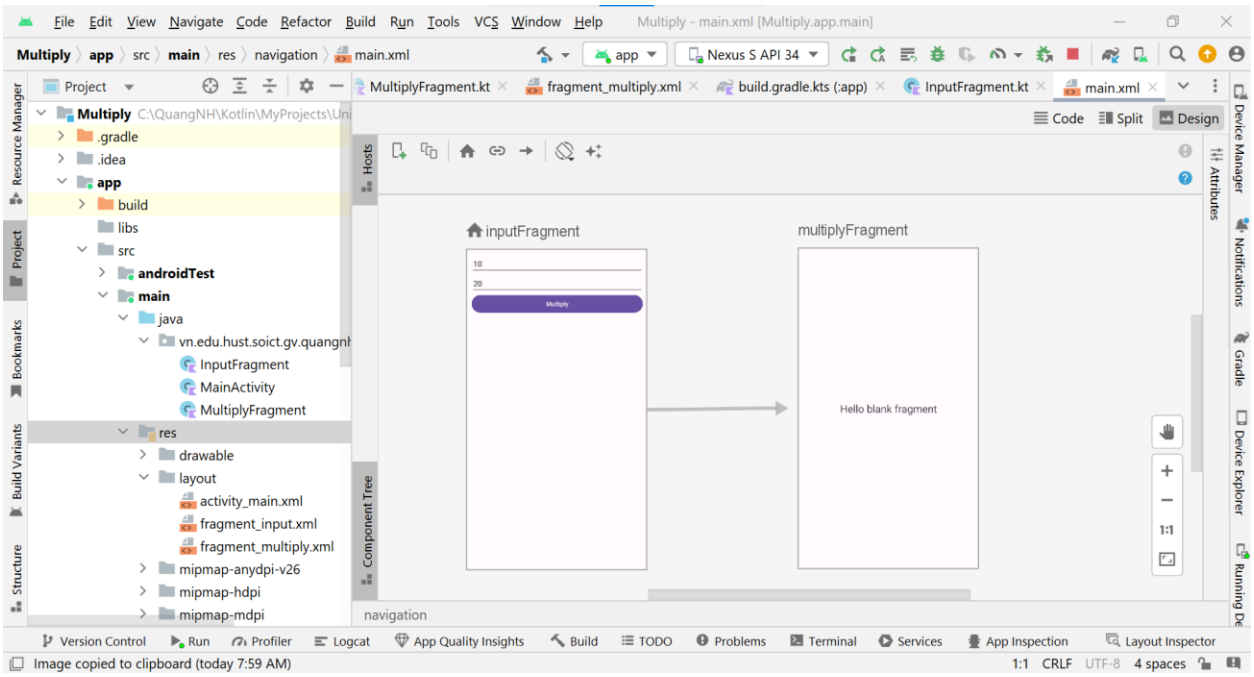
Ấn chuột phải vào thư mục res, chọn New => Android Resource File



Trong mục File name: nhập main

Trong mục Resource type: chọn Navigation

Mở file res/navigation/main.xml:



Dùng nút “Thêm” để thêm InputFragment và MultiplyFragment.

Dùng nút “Home” để chọn InputFragment là Home.

Dùng nút “->” để tạo action từ InputFragment sang MultiplyFragment

Kết quả mã nguồn file main.xml:

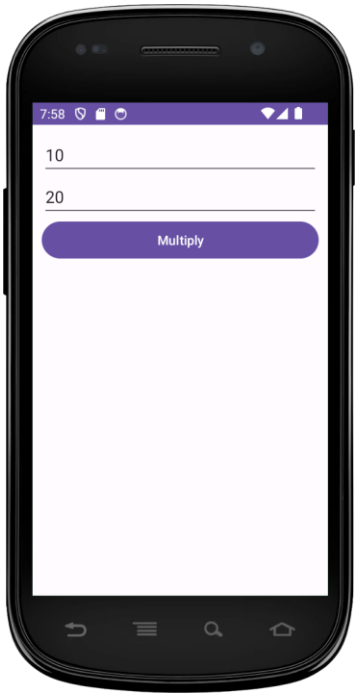
```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    app:startDestination="@id/inputFragment">

    <fragment
        android:id="@+id/inputFragment"
        android:name="vn.edu.hust.soict.gv.quangnh.multiply.InputFragment"
        android:label="fragment_input"
        tools:layout="@layout/fragment_input">
        <action
            android:id="@+id/action_inputFragment_to_multiplyFragment"
            app:destination="@id/multiplyFragment" />
    </fragment>
    <fragment
        android:id="@+id/multiplyFragment"
        android:name="vn.edu.hust.soict.gv.quangnh.multiply.MultiplyFragment"
        android:label="fragment_multiply"
        tools:layout="@layout/fragment_multiply">
    </fragment>
</navigation>
```

Bước 4. Tích hợp Navigation Graph vào MainActivity

Sửa lại file activity_main.xml như sau:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <fragment
        android:id="@+id/nav_host"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/main"/>

</FrameLayout>
```

Biên dịch và chạy ứng dụng, kết quả thu được:



Bước 5. Xử lý sự kiện khi người dùng ấn nút Multiply trên InputFragment

Trong InputFragment.kt, viết chồng hàm onViewCreated:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
```

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

    button.setOnClickListener {
view.findNavController().navigate(R.id.action_inputFragment_to_multiplyFragme
nt)
        }
}
```

Biên dịch và chạy chương trình, khi đó nếu người dùng ấn vào nút Multiply, chương trình sẽ chuyển sang và hiển thị MultiplyFragment.

Bước 6. Truyền / nhận dữ liệu sử dụng Safe Args

Trong file res/navigation/main.xml, bổ sung hai Argument cho fragment multiplyFragment:

```
<fragment
    android:id="@+id/multiplyFragment"
    android:name="vn.edu.hust.soict.gv.quangnh.multiply.MultiplyFragment"
    android:label="fragment_multiply"
    tools:layout="@layout/fragment_multiply">
    <argument
        android:name="number1"
        android:defaultValue="1.0"
        app:argType="float" />
    <argument
        android:name="number2"
        android:defaultValue="1.0"
        app:argType="float" />

</fragment>
```

Để truyền dữ liệu, trong InputFragment.kt, cập nhật hàm onViewCreated:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    val button: Button = view.findViewById(R.id.bt_multiply)
    val number1: EditText = view.findViewById(R.id.number1)
    val number2: EditText = view.findViewById(R.id.number2)

    button.setOnClickListener {
        //Toast.makeText(this.requireContext(), n1.toString(),
Toast.LENGTH_SHORT).show()
        val n1:Float = number1.text.toString().toFloatOrNull() ?: 0.0f
        val n2:Float = number2.text.toString().toFloatOrNull() ?: 0.0f

//view.findNavController().navigate(R.id.action_inputFragment_to_multiplyFrag
ment)
        val action =
InputFragmentDirections.actionInputFragmentToMultiplyFragment(n1, n2)
        view.findNavController().navigate(action)
    }
}
```

Để nhận dữ liệu, trong file MultiplyFragment, viết chồng hàm onViewCreated:

val args: MultiplyFragmentArgs by navArgs()

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    val number1 = args.number1
    val number2 = args.number2
    val result = number1 * number2
    val output:TextView = view.findViewById(R.id.mf_multiply)
    output.text = "${number1} * ${number2} = ${result}"

}
```

Như vậy là hoàn thành bài thực hành. Hãy biên dịch và chạy thử nghiệm lại chương trình.

Collections in Kotlin

Link: https://developer.android.com/codelabs/basic-android-kotlin-training-collections

About this codelab

*subject*Last updated Oct 20, 2023
*account_circle*Written by Google Developers Training team

1. Before you begin

Caution: This codelab is out of date and no longer maintained. Instead, please refer to the Android Basics with Compose course for the latest recommended practices.

In this codelab you will learn more about collections, and about lambdas and higher-order functions in Kotlin.

Prerequisites

- A basic understanding of Kotlin concepts as presented in the prior codelabs.
- Familiar with using the Kotlin Playground for creating and editing Kotlin programs.

What you'll learn

- How to work with collections including sets and maps
- The basics of lambdas
- The basics of higher-order functions

What you need

- A computer with an internet connection to access the Kotlin Playground

2. Learn about collections

A *collection* is a group of related items, like a list of words, or a set of employee records. The collection can have the items ordered or unordered, and the items can be unique or not. You've already learned about one type of collection, lists. Lists have an order to the items, but the items don't have to be unique.

As with lists, Kotlin distinguishes between mutable and immutable collections. Kotlin provides numerous functions for adding or deleting items, viewing, and manipulating collections.

Create a list

In this task you'll review creating a list of numbers and sort them.

1. Open the Kotlin Playground.
2. Replace any code with this code:

```
fun main() {
    val numbers = listOf(0, 3, 8, 4, 0, 5, 5, 8, 9, 2)
    println("list:   ${numbers}")
}
```

3. Run the program by tapping the green arrow, and look at the results that appear:

```
list:   [0, 3, 8, 4, 0, 5, 5, 8, 9, 2]
```

4. The list contains 10 numbers from 0 to 9. Some of the numbers appear more than once while some don't appear at all.
5. The order of the items in the list matters: the first item is `0`, the second item is `3`, and so on. The items will stay in that order unless you change them.
6. Recall from earlier codelabs that lists have many built-in functions, like `sorted()`, which returns a copy of the list sorted in ascending order. After the `println()`, add a line to your program to print a sorted copy of the list:

```
println("sorted: ${numbers.sorted()}")
```

7. Run your program again and look at the results:

```
list:   [0, 3, 8, 4, 0, 5, 5, 8, 9, 2]
sorted: [0, 0, 2, 3, 4, 5, 5, 8, 8, 9]
```

With the numbers sorted, it's easier to see how many times each number appears in your list, or if it doesn't appear at all.

Learn about sets

Another type of collection in Kotlin is a *set*. It's a group of related items, but unlike a list, there can't be any duplicates, and the order doesn't matter. An item can be in the set or not, but if it's in the set, there is only one copy of it. This is similar to the mathematical concept of a set. For example, there is a set of books that you've read. Reading a book multiple times doesn't change the fact it is in the set of books that you've read.

1. Add these lines to your program to convert the list to a set:

```
val setOfNumbers = numbers.toSet()
println("set:    ${setOfNumbers}")
```

2. Run your program and look at the results:

```
list:   [0, 3, 8, 4, 0, 5, 5, 8, 9, 2]
sorted: [0, 0, 2, 3, 4, 5, 5, 8, 8, 9]
set:    [0, 3, 8, 4, 5, 9, 2]
```

The result has all the numbers in the original list, but each only appears once. Note that they are in the same order as in the original list, but that order isn't significant for a set.

3. Define a mutable set and an immutable set, and initialize them with the same set of numbers but in a different order by adding these lines:

```
val set1 = setOf(1,2,3)
val set2 = mutableSetOf(3,2,1)
```

4. Add a line to print whether they are equal:

```
println("$set1 == $set2: ${set1 == set2}")
```

5. Run your program and look at the new results:

```
[1, 2, 3] == [3, 2, 1]: true
```

Even though one is mutable and one isn't, and they have the items in a different order, they're considered equal because they contain exactly the same set of items.

One of the main operations you might perform on a set is checking if a particular item is in the set or not with the `contains()` function. You've seen `contains()` before, but used it on a list.

6. Add this line to your program to print if `7` is in the set:

```
println("contains 7: ${setOfNumbers.contains(7)}")
```

7. Run your program and look at the additional results:

```
contains 7: false
```

You can try testing it with a value that is in the set, too.

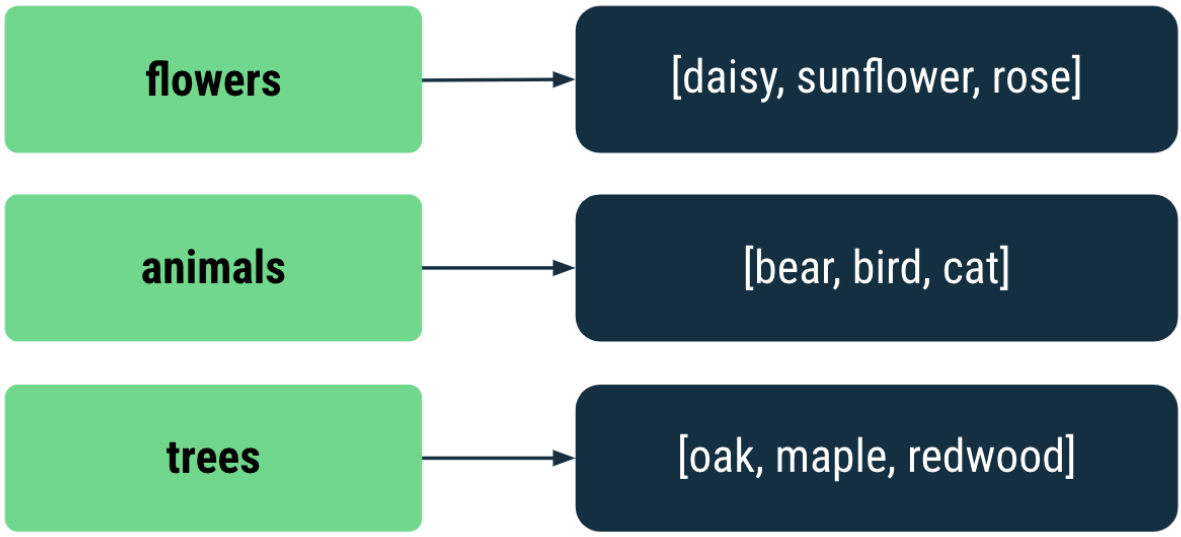
All of the code above:

```
fun main() {
    val numbers = listOf(0, 3, 8, 4, 0, 5, 5, 8, 9, 2)
    println("list:   ${numbers}")
    println("sorted: ${numbers.sorted()}")
    val setOfNumbers = numbers.toSet()
    println("set:    ${setOfNumbers}")
    val set1 = setOf(1,2,3)
    val set2 = mutableSetOf(3,2,1)
    println("$set1 == $set2: ${set1 == set2}")
    println("contains 7: ${setOfNumbers.contains(7)}")
}
```

As with mathematical sets, in Kotlin you can also perform operations like the intersection (∩) or the union (∪) of two sets, using `intersect()` or `union()`.

Learn about maps

The last type of collection you'll learn about in this codelab is a *map* or *dictionary*. A map is a set of *key-value pairs*, designed to make it easy to look up a value given a particular key. Keys are unique, and each key maps to exactly one value, but the values can have duplicates. Values in a map can be strings, numbers, or objects—even another collection like a list or a set.



A map is useful when you have pairs of data, and you can identify each pair based on its key. The key "maps to" the corresponding value.

1. In the Kotlin playground, replace all the code with this code that creates a mutable map to store people's names and their ages:

```
fun main() {
    val peopleAges = mutableMapOf<String, Int>(
        "Fred" to 30,
        "Ann" to 23
    )
    println(peopleAges)
}
```

This creates a mutable map of a `String` (key) to an `Int` (value), initializes the map with two entries, and prints the items.

2. Run your program and look at the results:

```
{Fred=30, Ann=23}
```

3. To add more entries to the map, you can use the `put()` function, passing in the key and the value:

```
peopleAges.put("Barbara", 42)
```

4. You can also use a shorthand notation to add entries:

```
peopleAges["Joe"] = 51
```

Here is all the of the code above:

```
fun main() {
    val peopleAges = mutableMapOf<String, Int>(
        "Fred" to 30,
        "Ann" to 23
    )
    peopleAges.put("Barbara", 42)
    peopleAges["Joe"] = 51
    println(peopleAges)
}
```

5. Run your program, and look at the results:

```
{Fred=30, Ann=23, Barbara=42, Joe=51}
```

As noted above, the keys (names) are unique, but the values (ages) can have duplicates. What do you think happens if you try to add an item using one of the same keys?

6. Before the `println()`, add this line of code:

```
peopleAges["Fred"] = 31
```

7. Run your program, and look at the results:

```
{Fred=31, Ann=23, Barbara=42, Joe=51}
```

The key `"Fred"` doesn't get added again, but the value it maps to is updated to `31`.

As you can see, maps are useful as a quick way to map keys to values in your code!

3. Working with collections

Although they have different qualities, different types of collections have a lot of behavior in common. If they're mutable, you can add or remove items. You can enumerate all the items, find a particular item, or sometimes convert one type of collection to another. You did this earlier where you converted a `List` to a `Set` with `toSet()`. Here are some helpful functions for working with collections.

forEach

Suppose you wanted to print the items in `peopleAges`, and include the person's name and age. For example, `"Fred is 31, Ann is 23,..."` and so on. You learned about `for` loops in an earlier codelab, so you could write a loop with `for (people in peopleAges) { ... }`.

However, enumerating all the objects in a collection is a common operation, so Kotlin provides `forEach()`, which goes through all the items for you and performs an operation on each one.

1. In the playground, add this code after the `println()`:

```
peopleAges.forEach { print("${it.key} is ${it.value}, ") }
```

It's similar to the `for` loop, but a little more compact. Instead of you specifying a variable for the current item, the `forEach` uses the special identifier `it`.

Note that you didn't need to add parentheses when you called the `forEach()` method, just pass the code in curly braces `{}`.

2. Run your program and look at the additional results:

```
Fred is 31, Ann is 23, Barbara is 42, Joe is 51,
```

That's very close to what you want, but there's an extra comma on the end.

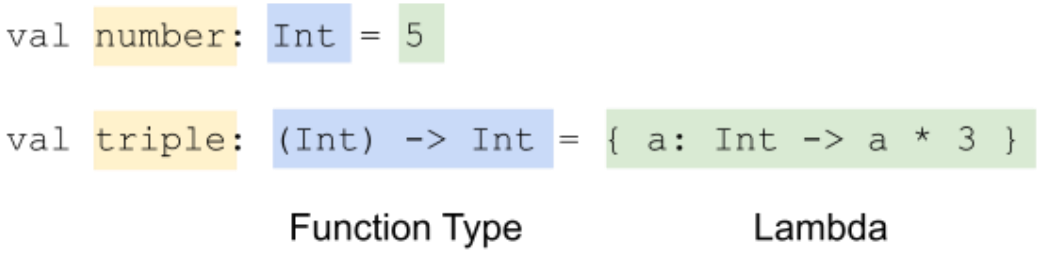
Converting a collection into a string is a common operation, and that extra separator at the end is a common problem, too. You'll learn how to deal with that in the steps ahead.

map

The `map()` function (which shouldn't be confused with a map or dictionary collection above) applies a transformation to each item in a collection.

You could even store a lambda into a variable, as shown in the below diagram. The syntax is similar to how you declare a variable of a basic data type like an `Int`. Observe the variable name (yellow box), variable type (blue box), and variable value (green box). The `triple` variable stores a function. Its type is a function type `(Int) -> Int`, and the value is a lambda expression `{ a: Int -> a * 3}`.

1. Try this code in the playground. Define and call the `triple` function by passing it a number like 5.




```
fun main() {
    val triple: (Int) -> Int = { a: Int -> a * 3 }
    println(triple(5))
}
```

2. The resulting output should be:

```
15
```

Note: It's common to have a lambda that has a single parameter, so Kotlin offers a shorthand. Kotlin implicitly uses the special identifier `it` for the parameter of a lambda with a single parameter.

3. Within the curly braces, you can omit explicitly declaring the parameter (`a: Int`), omit the function arrow (`->`), and just have the function body. Update the `triple` function declared in your `main` function and run the code.

```
val triple: (Int) -> Int = { it * 3 }
```

4. The output should be the same, but now your lambda is written more concisely! For more examples of lambdas, check out this resource.

```
15
```

Higher-order functions

Now that you are starting to see the flexibility of how you can manipulate functions in Kotlin, let's talk about another really powerful idea, a *higher-order function*. This just means passing a function (in this case a lambda) to another function, or returning a function from another function.

It turns out that `map`, `filter`, and `forEach` functions are all examples of higher-order functions because they all took a function as a parameter. (In the lambda passed to this `filter` higher-order function, it's okay to omit the single parameter and arrow symbol, and also use the `it` parameter.)

```
peopleAges.filter { it.key.length < 4 }
```

Here's an example of a new higher-order function: `sortedWith()`.

If you want to sort a list of strings, you can use the built-in `sorted()` method for collections. However, if you wanted to sort the list by the length of the strings, you need to write some code to get the length of two strings and compare them. Kotlin lets you do this by passing a lambda to the `sortedWith()` method.

Note: To compare two objects for sorting, the convention is to return a value less than 0 if the first object is less than the second, 0 if they are equal, and a value greater than 0 if the first object is greater than the second.

1. In the playground, create a list of names and print it sorted by name with this code:

```
fun main() {
    val peopleNames = listOf("Fred", "Ann", "Barbara", "Joe")
    println(peopleNames.sorted())
}
```

2. Now print the list sorted by the length of the names by passing a lambda to the `sortedWith()` function. The lambda should take in two parameters of the same type and return an `Int`. Add this line of code after the `println()` statement in the `main()` function.

```
println(peopleNames.sortedWith { str1: String, str2: String -> str1.length - str2.length })
```

3. Run your program and look at the results.

```
[Ann, Barbara, Fred, Joe]
[Ann, Joe, Fred, Barbara]
```

The lambda passed to `sortedWith()` has two parameters, `str1` which is a `String`, and `str2` which is a `String`. Then you see the function arrow, followed by the function body.

```
{ str1: String, str2: String -> str1.length - str2.length }
```

Remember that the last expression in the lambda is the return value. In this case, it returns the difference between the length of the first string and the length of the second string, which is an `Int`. That matches what is needed for sorting: if `str1` is shorter than `str2`, it will return a value less than 0. If `str1` and `str2` are the same length, it will return 0. If `str1` is longer than `str2`, it

will return a value greater than 0. By doing a series of comparison between two `Strings` at a time, the `sortedWith()` function outputs a list where the names will be in order of increasing length.

OnClickListener and OnKeyListener in Android

Tying this back to what you have learned in Android so far, you have used lambdas in earlier codelabs, such as when you set a click listener for the button in the Tip Calculator app:

```
calculateButton.setOnClickListener{ calculateTip() }
```

Using a lambda to set the click listener is convenient shorthand. The long form way of writing the above code is shown below, and compared against the shortened version. You don't have to understand all the details of the long form version, but notice some patterns between the two versions.

LONG FORM

```
calculateButton.setOnClickListener(object: View.OnClickListener {
    override fun onClick(view: View?) {
        calculateTip()
    }
})
```

SHORT FORM

```
calculateButton.setOnClickListener { view -> calculateTip() }
```

Observe how the lambda has the same function type as the `onClick()` method in `OnClickListener` (takes in one `View` argument and returns `Unit`, which means no return value).

The shortened version of the code is possible because of something called SAM (Single-Abstract-Method) conversion in Kotlin. Kotlin converts the lambda into an `OnClickListener` object which implements the single abstract method `onClick()`. You just need to make sure the lambda function type matches the function type of the abstract function.

Since the `view` parameter is never used in the lambda, the parameter can be omitted. Then we just have the function body in the lambda.

```
calculateButton.setOnClickListener { calculateTip() }
```

These concepts are challenging, so be patient with yourself as it'll take time and experience for these concepts to sink in. Let's look at another example. Recall when you set a key listener on the "Cost of service" text field in the tip calculator, so the onscreen keyboard could be hidden when the Enter key is pressed.

```
costOfServiceEditText.setOnKeyListener { view, keyCode, event ->
handleKeyEvent(view, keyCode) }
```

When you look up `OnKeyListener`, the abstract method has the following parameters `onKey(View v, int keyCode, KeyEvent event)` and returns a `Boolean`. Because of SAM conversions in Kotlin, you can pass in a lambda to `setOnKeyListener()`. Just be sure the lambda has the function type `(View, Int, KeyEvent) -> Boolean`.

Here's a diagram of the lambda expression used above. The parameters are view, keyCode, and event. The function body consists of `handleKeyEvent(view, keyCode)` which uses the parameters passed in and returns a `Boolean`.

```
{ view, keyCode, event -> handleKeyEvent(view, keyCode) }
```

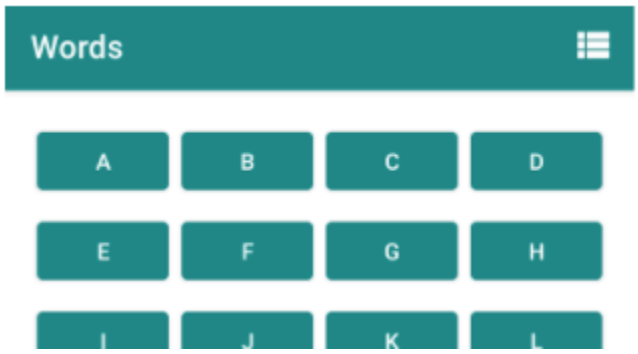
Note: If you don't use a lambda parameter in the function body, you can name it _ to make your code more readable and less cluttered. This code has the same behavior.

```
costOfServiceEditText.setOnKeyListener { view, keyCode, _ ->
handleKeyEvent(view, keyCode) }
```

5. Make word lists

Now let's take everything you learned about collections, lambdas, and higher order functions and apply it to a realistic use case.

Suppose you wanted to create an Android app to play a word game or learn vocabulary words. The app might look something like this, with a button for each letter of the alphabet:



Clicking on the letter **A** would bring up a short list of some words that begin with the letter A, and so on.

You'll need a collection of words, but what kind of collection? If the app is going to include some words that start with each letter of the alphabet, you'll need a way to find or organize all the words that start with a given letter. To make it more challenging, you'll want to choose different words from your collection each time the user runs the app.

First, start with a list of words. For a real app you'd want a longer list of words, and include words that start with all the letters of the alphabet, but a short list is enough to work with for now.

1. Replace the code in the Kotlin playground with this code:

```
fun main() {
    val words = listOf("about", "acute", "awesome", "balloon", "best",
"brief", "class", "coffee", "creative")
}
```

2. To get a collection of the words that start with the letter B, you can use `filter` with a lambda expression. Add these lines:

```
val filteredWords = words.filter { it.startsWith("b", ignoreCase = true) }
println(filteredWords)
```

The `startsWith()` function returns true if a string starts with the specified string. You can also tell it to ignore case, so "b" will match "b" or "B".

3. Run your program and look at the result:

```
[balloon, best, brief]
```

4. Remember that you want the words randomized for your app. With Kotlin collections, you can use the `shuffled()` function to make a copy of a collection with the items randomly shuffled. Change the filtered words to be shuffled, too:

```
val filteredWords = words.filter { it.startsWith("b", ignoreCase = true) }
    .shuffled()
```

5. Run your program and look at the new results:

```
[brief, balloon, best]
```

Because the words are randomly shuffled, you may see the words in a different order.

6. You don't want all the words (especially if your real word list is long), just a few. You can use the `take()` function to get the first items in the collection. Make the filtered words just include the first two shuffled words:

```
val filteredWords = words.filter { it.startsWith("b", ignoreCase = true) }
    .shuffled()
    .take(2)
```

7. Run your program and look at the new results:

```
[brief, balloon]
```

Again because of the random shuffling, you might see different words each time you run it.

8. Finally, for the app you want the random list of words for each letter sorted. As before, you can use the `sorted()` function to return a copy of the collection with the items sorted:

```
val filteredWords = words.filter { it.startsWith("b", ignoreCase = true) }
    .shuffled()
    .take(2)
    .sorted()
```

9. Run your program and look at the new results:

```
[balloon, brief]
```

All the code above:

```
fun main() {
    val words = listOf("about", "acute", "awesome", "balloon", "best",
"brief", "class", "coffee", "creative")
    val filteredWords = words.filter { it.startsWith("b", ignoreCase = true)
}
        .shuffled()
        .take(2)
        .sorted()
    println(filteredWords)
}
```

10. Try changing the code to create a list of one random word that starts with the letter c. What do you have to change in the code above?

```
val filteredWords = words.filter { it.startsWith("c", ignoreCase = true) }
    .shuffled()
    .take(1)
```

In the actual app, you'll need to apply the filter for each letter of the alphabet, but now you know how to generate the word list for each letter!

Collections are powerful and flexible. There's a lot they can do, and there can be more than one way to do something. As you learn more about programming, you'll learn how to figure out which type of collection is right for the problem at hand and the best ways to process it.

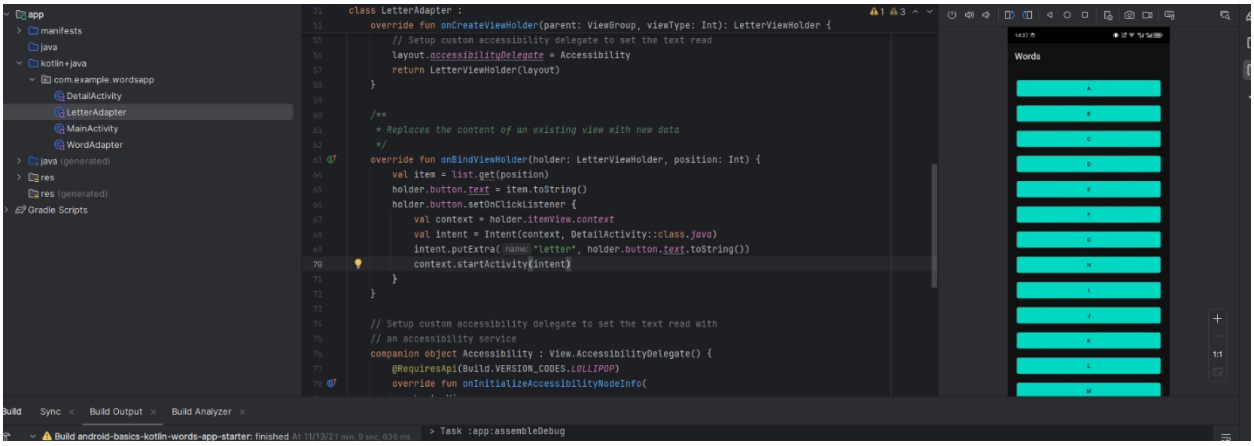
Lambdas and higher-order functions make working with collections easier and more concise. These ideas are very useful, so you'll see them used again and again.

6. Summary

- A collection is a group of related items
- Collections can be mutable or immutable

Lab 6.2. Activities and Intents

5. Set Up Explicit Intent



Việc tạo và sử dụng một `Intent` trong Android chỉ cần thực hiện một vài bước đơn giản:

1. Thiết lập sự kiện click cho button trong `LetterAdapter.kt`:
Trong file `LetterAdapter.kt`, mở phương thức `onBindViewHolder()` và cuộn xuống dưới dòng mã thiết lập văn bản cho nút (button). Sau đó, thiết lập sự kiện `onClickListener` cho `holder.button`.

```
kotlin
holder.button.setOnClickListener {
    // Mã xử lý sự kiện
}
```

2. Lấy tham chiếu đến Context:
Trong phần xử lý sự kiện click, lấy tham chiếu đến `Context`, vì cần sử dụng `Context` để tạo một `Intent` cho việc chuyển đổi giữa các activity.

```
kotlin
val context = holder.itemView.context
```

3. Tạo một Intent:
Sau khi đã có `context`, tiếp theo tạo một đối tượng `Intent`. Truyền vào `context` và tên của lớp `Activity` đích mà bạn muốn chuyển đến. Trong trường hợp này, đó là `DetailActivity`.

```
kotlin
val intent = Intent(context, DetailActivity::class.java)
```

Tên của activity đích được xác định bằng `DetailActivity::class.java`. Đây là cách khai báo tên của activity mà bạn muốn hiển thị, và hệ thống sẽ tự động tạo một đối tượng `DetailActivity` ở phía sau.

4. Truyền dữ liệu vào Intent với `putExtra`:
Tiếp theo, sử dụng phương thức `putExtra` của `Intent` để truyền dữ liệu từ activity này sang activity khác. Dữ liệu có thể là bất kỳ đối tượng nào, nhưng cần phải có tên để có thể lấy lại sau này.

```
kotlin
intent.putExtra("letter", holder.button.text.toString())
```

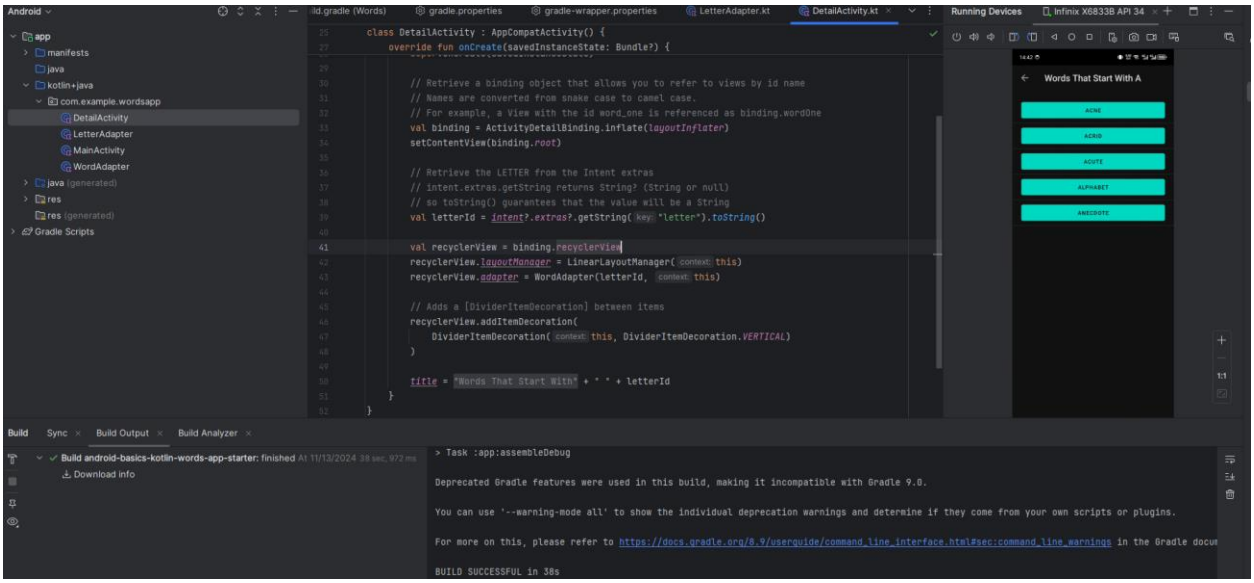
Ở đây, `"letter"` là tên của dữ liệu (extra) mà bạn muốn truyền, và `holder.button.text.toString()` là giá trị của dữ liệu đó. Việc gọi `toString()` là cần thiết vì thuộc tính `text` của button là một `CharSequence` – một kiểu dữ liệu giao diện (interface) đại diện cho chuỗi ký tự, nhưng phương thức `putExtra()` yêu cầu phải truyền vào một đối tượng `String`, không phải `CharSequence`. Vì vậy, cần phải chuyển đổi `CharSequence` sang `String` bằng cách gọi `toString()`.

5. Khởi chạy Activity mới với `startActivity()`:
Cuối cùng, để thực thi `Intent` và chuyển đến màn hình mới, gọi phương thức `startActivity()` trên đối tượng `Context`, truyền vào đối tượng `Intent` mà bạn vừa tạo.

```
kotlin
context.startActivity(intent)
```

Sau khi hoàn thành các bước này, bạn có thể chạy ứng dụng và thử nhấn vào một chữ cái. Màn hình chi tiết sẽ được hiển thị! Tuy nhiên, cho dù người dùng nhấn vào chữ cái nào, màn hình chi tiết vẫn sẽ hiển thị từ của chữ cái "A". Bạn vẫn cần làm thêm một vài thay đổi trong `DetailActivity` để màn hình chi tiết hiển thị từ cho bất kỳ chữ cái nào mà bạn truyền qua Intent.

6. Set Up DetailActivity



Sau khi chuyển từ `MainActivity` sang `DetailActivity`, mục tiêu là lấy dữ liệu chữ cái được truyền qua Intent và hiển thị nó trên màn hình chi tiết. Dưới đây là các bước để thực hiện điều này và cải thiện việc tổ chức mã nguồn bằng cách sử dụng constants.

Bước 1: Lấy Dữ liệu Chữ Cái

Trong `DetailActivity`, trong phương thức `onCreate()`, chữ cái được truyền từ `MainActivity` được lấy từ Intent extras. Intent dùng để mở `DetailActivity` có thể được truy cập qua thuộc tính `intent`. Thuộc tính `extras` của `intent` chứa dữ liệu bổ sung được truyền vào. Trong trường hợp này, chuỗi `"letter"` được truyền từ activity trước và được lấy như sau:

```
kotlin
val letterId = intent?.extras?.getString("letter").toString()
```

Giải thích:

- `intent`: Đây là thuộc tính có sẵn trong mọi Activity, dùng để tham chiếu đến Intent đã khởi tạo activity này. Intent này giữ ngữ cảnh và bất kỳ dữ liệu nào được truyền khi activity được khởi động.
- `extras`: Đây là một `Bundle` chứa tất cả các dữ liệu bổ sung được truyền vào activity. Vì `extras` có thể là null (ví dụ nếu không có dữ liệu nào được truyền qua Intent), ta sử dụng toán tử `?.` để đảm bảo mã không gặp lỗi khi truy cập vào nó.

- `getString("letter")`: Phương thức này lấy giá trị chuỗi tương ứng với khóa `"letter"`.
- `toString()`: Mặc dù `getString()` trả về một giá trị kiểu `String?` (chuỗi có thể là null), ta gọi `toString()` để chuyển nó thành chuỗi không null. Điều này cần thiết để tránh các vấn đề liên quan đến nullability.

Null Safety (An toàn Null):

Kotlin cung cấp các tính năng an toàn với null để đảm bảo rằng một đối tượng có thể tồn tại hoặc có thể là null. Khi truy cập các thuộc tính như `intent` hay `extras`, có thể chúng sẽ là null, vì vậy ta sử dụng toán tử `?.` để xử lý một cách an toàn khi giá trị có thể không có sẵn. Nếu `intent` hoặc `extras` là null, mã sẽ không thực hiện thao tác gì mà không gây lỗi.

Bước 2: Tổ chức Các Constants bằng Companion Objects

Mặc dù việc hardcode chuỗi `"letter"` là hợp lý trong các ứng dụng nhỏ, nhưng nó không phải là cách tốt nhất khi ứng dụng lớn lên với nhiều extras. Để mã nguồn gọn gàng hơn và dễ bảo trì, Kotlin cung cấp **companion objects**. Một companion object là một đối tượng singleton gắn liền với lớp, cho phép bạn định nghĩa các constants hoặc phương thức có thể truy cập mà không cần tạo đối tượng của lớp đó.

Tạo Companion Object trong `DetailActivity`:

1. **Định nghĩa Companion Object**: Trong `DetailActivity`, tạo một companion object để chứa giá trị constant cho `"letter"`.

```
kotlin
companion object {
    const val LETTER = "letter"
}
```

Companion object ở đây được sử dụng để định nghĩa các constant có thể được chia sẻ trong toàn bộ lớp `DetailActivity`.

2. **Cập nhật mã nguồn để sử dụng constant**: Thay vì hardcode `"letter"` trong `onCreate()`, bạn có thể tham chiếu đến constant từ companion object:

```
kotlin
val letterId = intent?.extras?.getString(LETTER).toString()
```

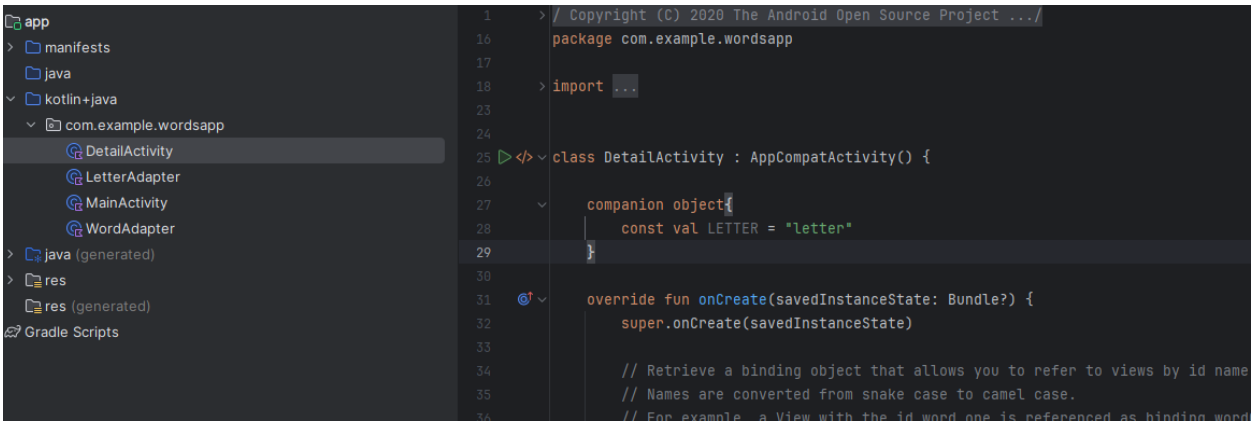
Điều này giúp mã nguồn trở nên sạch sẽ và dễ bảo trì hơn, đặc biệt khi bạn có nhiều extras cần xử lý. Constant `LETTER` giờ đã là một phần của `DetailActivity` và có thể được truy cập thông qua cú pháp dot notation.

Bước 3: Lợi ích của Việc Sử Dụng Companion Objects

- **Tổ chức mã nguồn**: Companion objects giúp bạn tổ chức các constants và các chức năng liên quan đến lớp trong một nơi duy nhất.
- **Tránh lặp lại mã**: Thay vì hardcode các giá trị nhiều lần trong các phần khác nhau của mã nguồn, việc sử dụng constants trong companion objects giúp giảm sự trùng lặp.
- **Khả năng mở rộng**: Khi ứng dụng phát triển và bạn bắt đầu truyền nhiều extras giữa các activity, việc có các constants được định nghĩa trong companion objects giúp duy trì khả năng mở rộng của ứng dụng.

Tóm lại, việc sử dụng intents và constants theo cách này giúp mã nguồn của bạn dễ bảo trì hơn, tổ chức hơn và tránh các vấn đề liên quan đến việc hardcode các giá trị. Cách làm này cũng giúp cải thiện tính dễ đọc và cấu trúc của ứng dụng Android.

7. Set Up Implicit Intent



Trong trường hợp này, bạn sẽ sử dụng một **implicit intent** để mở trình duyệt của người dùng và thực hiện tìm kiếm từ điển trên Google, thay vì thêm một activity mới trong ứng dụng. Đây là một ví dụ điển hình khi bạn không thể chắc chắn ứng dụng nào người dùng sẽ muốn mở để tìm kiếm từ, vì có thể người dùng sử dụng các trình duyệt khác nhau hoặc ứng dụng từ điển của bên thứ ba. Implicit intent cho phép hệ thống chọn ứng dụng phù hợp để xử lý yêu cầu của bạn.

Các Bước Thực Hiện

1. Định nghĩa URL Cơ Sở cho Tìm Kiếm Google:

Để thực hiện tìm kiếm trên Google, bạn cần định nghĩa URL cơ sở cho tìm kiếm. Mỗi lần bạn muốn tìm kiếm một từ, bạn sẽ thêm từ đó vào URL này. Để làm điều này, bạn cần thêm một hằng số `SEARCH_PREFIX` trong `DetailActivity` để sử dụng cho tất cả các tìm kiếm.

```
kotlin
companion object {
    const val LETTER = "letter"
    const val SEARCH_PREFIX = "https://www.google.com/search?q="
}
```

2. Mở `WordAdapter` và Thiết Lập `OnClickListener` cho Button:

Tiếp theo, mở `WordAdapter` và trong phương thức `onBindViewHolder()`, bạn cần thiết lập một sự kiện `setOnClickListener()` cho button. Khi người dùng nhấn vào từ, bạn sẽ tạo một `Uri` cho truy vấn tìm kiếm. Bạn cần nối từ vào `SEARCH_PREFIX` để tạo thành một URL tìm kiếm.

```
kotlin
holder.button.setOnClickListener {
    val queryUrl: Uri =
Uri.parse("${DetailActivity.SEARCH_PREFIX}${item}")
    val intent = Intent(Intent.ACTION_VIEW, queryUrl)
    context.startActivity(intent)
}
```

Trong đoạn mã trên:

- `Uri.parse("${DetailActivity.SEARCH_PREFIX}${item}")`: Tạo một URI cho tìm kiếm từ điển từ `SEARCH_PREFIX` và từ mà người dùng nhấn vào.
- `Intent.ACTION_VIEW`: Đây là một loại implicit intent chung, cho phép hệ thống mở URL trong trình duyệt web của người dùng. Hệ thống sẽ tự động chọn ứng dụng phù hợp (trình duyệt) để xử lý yêu cầu này.

3. Khởi Chạy Activity Bằng `startActivity()`:

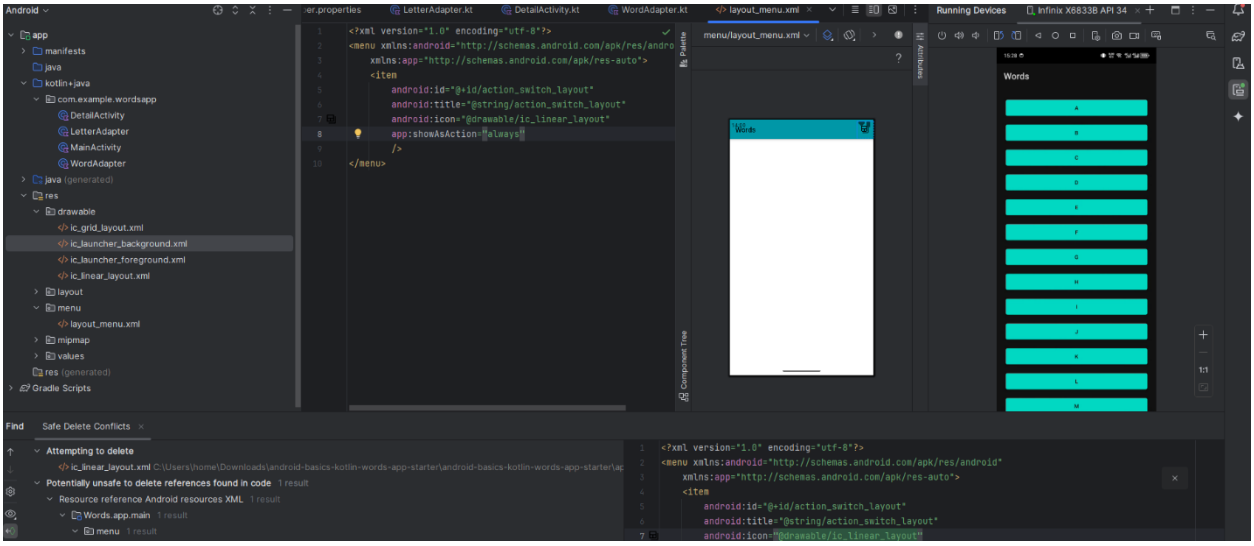
Cuối cùng, bạn gọi `startActivity()` để mở trình duyệt với URL tìm kiếm mà bạn vừa tạo. Hệ thống sẽ mở trình duyệt web và thực hiện tìm kiếm trên Google cho từ mà người dùng đã chọn.

```
kotlin
context.startActivity(intent)
```

Kết Quả

Sau khi thực hiện các bước trên, khi bạn chạy ứng dụng, vào danh sách các từ và nhấn vào một từ, thiết bị của bạn sẽ mở trình duyệt web và thực hiện tìm kiếm từ điển trên Google cho từ đó. Hành vi chính xác có thể khác nhau tùy thuộc vào các ứng dụng trình duyệt hoặc ứng dụng từ điển mà người dùng đã cài đặt. Điều này giúp mang lại một trải nghiệm liền mạch cho người dùng mà không cần phải thêm code phức tạp vào ứng dụng của bạn.

8. Set Up Menu and Icons



Để thêm tính năng chuyển đổi giữa chế độ hiển thị dạng lưới (grid) và danh sách (list) trong ứng dụng Android, bạn cần thực hiện các bước sau:

Thêm Biểu Tượng Cho Chế Độ Lưới và Danh Sách

Trước tiên, bạn cần thêm hai biểu tượng để đại diện cho chế độ hiển thị lưới và danh sách. Để làm điều này, bạn sẽ thêm các vector assets clip art có tên "view module" (đặt tên là `ic_grid_layout`) và "view list" (đặt tên là `ic_linear_layout`).

Để thêm các biểu tượng này:

- Vào **res > drawable** và chọn **New > Vector Asset**.
- Tìm kiếm biểu tượng "view module" cho chế độ lưới và "view list" cho chế độ danh sách.
- Đặt tên các biểu tượng tương ứng là `ic_grid_layout` và `ic_linear_layout`.

Tạo Tập Tin Menu

Tiếp theo, bạn cần chỉ định những gì sẽ được hiển thị trong app bar và biểu tượng nào sẽ được sử dụng. Để làm điều này, bạn tạo một tập tin resource mới trong thư mục `res` của dự án.

Thực hiện như sau:

- Nhấp chuột phải vào thư mục `res`, chọn **New > Android Resource File**.
- Đặt **Resource Type** là **Menu** và **File Name** là `layout_menu`.
- Nhấn **OK** để tạo tệp tin `layout_menu.xml`.

Sửa Nội Dung Tệp `layout_menu.xml`

Mở tệp `res/menu/layout_menu.xml` và thay thế nội dung của nó bằng đoạn mã sau:

```
xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_toggle_view"
        android:title="Toggle View"
        android:icon="@drawable/ic_linear_layout"
        android:showAsAction="always" />
</menu>
```

Cấu trúc của tệp menu khá đơn giản. Giống như việc bạn bắt đầu một layout bằng layout manager để chứa các view con, tệp menu bắt đầu bằng thẻ `<menu>`, chứa các tùy chọn (items) con bên trong. Trong ví dụ trên, chỉ có một tùy chọn, với một số thuộc tính:

- **id**: Được sử dụng để tham chiếu đến menu option trong mã code.
- **title**: Đây là văn bản mô tả cho tùy chọn, tuy nhiên nó không hiển thị trong giao diện người dùng của bạn nhưng có thể hữu ích cho các công cụ hỗ trợ như screen readers.
- **icon**: Đây là biểu tượng mặc định, được đặt là `ic_linear_layout`. Tuy nhiên, nó sẽ được chuyển đổi giữa biểu tượng chế độ lưới và danh sách khi người dùng chọn.
- **showAsAction**: Thuộc tính này xác định cách thức hiển thị nút. Khi giá trị là `always`, nút này sẽ luôn hiển thị trong app bar và không trở thành một phần của menu tràn.

Cập Nhật `MainActivity.kt`

Để menu có thể hoạt động, bạn cần thêm mã trong `MainActivity.kt` để xử lý các sự kiện menu.

Trong `MainActivity.kt`, thực hiện các bước sau:

- **Override `onOptionsItemSelected`** để xử lý sự kiện khi người dùng nhấn vào nút trong app bar.
- Thêm mã để thay đổi biểu tượng của menu khi người dùng chuyển đổi chế độ giữa lưới và danh sách.

Ví dụ:

```
kotlin
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.action_toggle_view -> {
            // Toggle giữa chế độ grid và list
            toggleViewMode()
            return true
        }
        else -> return super.onOptionsItemSelected(item)
    }
}

private fun toggleViewMode() {
    // Thay đổi chế độ hiển thị, có thể dùng một biến để lưu trạng
thái hiện tại
    if (isGridMode) {
        // Chuyển sang chế độ danh sách
        recyclerView.layoutManager = LinearLayoutManager(this)
        item.icon = ContextCompat.getDrawable(this,
R.drawable.ic_linear_layout)
    } else {
        // Chuyển sang chế độ lưới
        recyclerView.layoutManager = GridLayoutManager(this, 2) // 2
cột
        item.icon = ContextCompat.getDrawable(this,
R.drawable.ic_grid_layout)
    }
    isGridMode = !isGridMode // Đổi trạng thái
}
```

Trong đoạn mã trên:

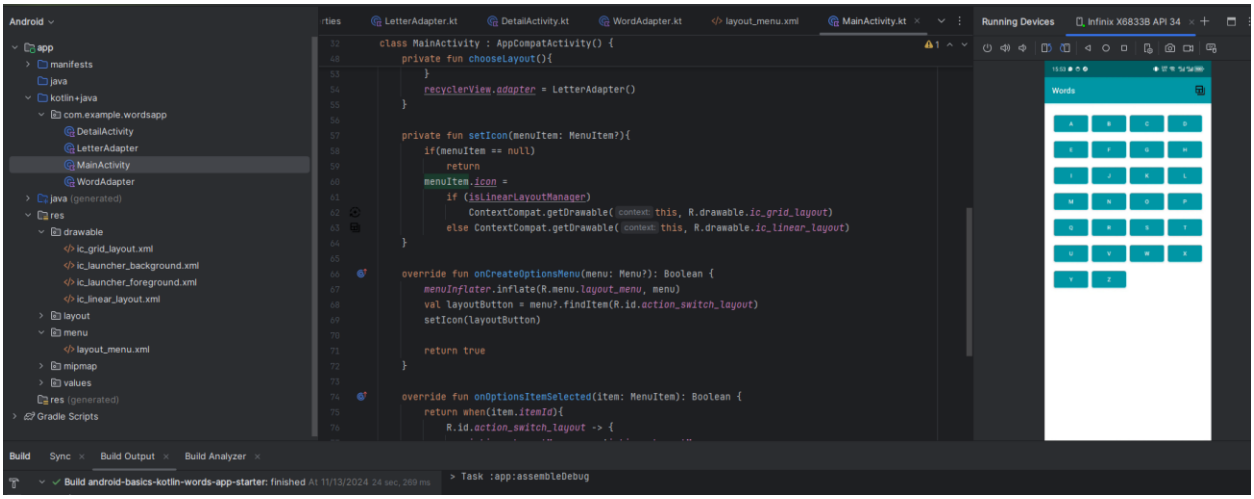
- `onOptionsItemSelected`: Xử lý sự kiện khi người dùng nhấn vào nút trong menu. Khi người dùng chọn `action_toggle_view`, nó sẽ gọi hàm `toggleViewMode`.

- `toggleViewMode`: Hàm này kiểm tra trạng thái hiện tại của chế độ hiển thị và chuyển đổi giữa chế độ danh sách và lưới, đồng thời thay đổi biểu tượng trong menu.

Kết Quả

Sau khi thực hiện các bước trên, khi người dùng nhấn vào biểu tượng trong app bar, ứng dụng của bạn sẽ chuyển đổi giữa chế độ hiển thị lưới và danh sách, đồng thời thay đổi biểu tượng của nút trong app bar.

9. Implement Menu button



Để triển khai tính năng chuyển đổi giữa chế độ hiển thị danh sách (Linear Layout) và chế độ lưới (Grid Layout) trong ứng dụng Android, ta cần thực hiện các bước sau trong `MainActivity.kt`.

Đầu tiên, cần tạo một thuộc tính trong `MainActivity.kt` để theo dõi trạng thái hiện tại của layout. Thuộc tính này sẽ giúp xác định xem ứng dụng đang ở chế độ hiển thị danh sách hay lưới. Giá trị mặc định của thuộc tính này là `true`, vì chế độ Linear Layout sẽ được sử dụng khi ứng dụng khởi động.

```
kotlin
private var isLinearLayoutManager = true
```

Sau khi tạo thuộc tính, ta tiếp tục viết một phương thức `chooseLayout()` để thay đổi cách hiển thị của RecyclerView. Phương thức này sẽ kiểm tra trạng thái của thuộc tính `isLinearLayoutManager` và quyết định sử dụng `LinearLayoutManager`

(dành cho chế độ danh sách) hoặc `GridLayoutManager` (dành cho chế độ lưới với 4 cột).

```
kotlin
private fun chooseLayout() {
    if (isLinearLayoutManager) {
        recyclerView.layoutManager = LinearLayoutManager(this)
    } else {
        recyclerView.layoutManager = GridLayoutManager(this, 4)
    }
    recyclerView.adapter = LetterAdapter()
}
```

Tiếp theo, ta cần thay đổi biểu tượng của nút chuyển đổi layout trong app bar mỗi khi người dùng thay đổi giữa các chế độ. Để làm điều này, ta viết phương thức `setIcon()` để cập nhật biểu tượng của nút tùy theo trạng thái hiện tại của layout. Nếu ứng dụng đang ở chế độ Linear Layout, biểu tượng sẽ là `ic_grid_layout`, ngược lại, nếu đang ở chế độ Grid Layout, biểu tượng sẽ là `ic_linear_layout`.

```
kotlin
private fun setIcon(menuItem: MenuItem?) {
    if (menuItem == null) return

    menuItem.icon =
        if (isLinearLayoutManager)
            ContextCompat.getDrawable(this, R.drawable.ic_grid_layout)
        else
            ContextCompat.getDrawable(this,
R.drawable.ic_linear_layout)
}
```

Sau khi tạo các phương thức hỗ trợ, ta cần ghi đè hai phương thức quan trọng để làm cho menu hoạt động chính xác.

Đầu tiên, ghi đè phương thức `onCreateOptionsMenu()`. Phương thức này sẽ được gọi khi menu của ứng dụng được tạo ra. Trong phương thức này, ta sẽ nạp menu từ tài nguyên `layout_menu.xml` và gọi phương thức `setIcon()` để đảm bảo rằng biểu tượng của nút chuyển đổi layout được cập nhật đúng.

```
kotlin
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.layout_menu, menu)
    val layoutButton = menu?.findItem(R.id.action_switch_layout)
    setIcon(layoutButton)
    return true
}
```

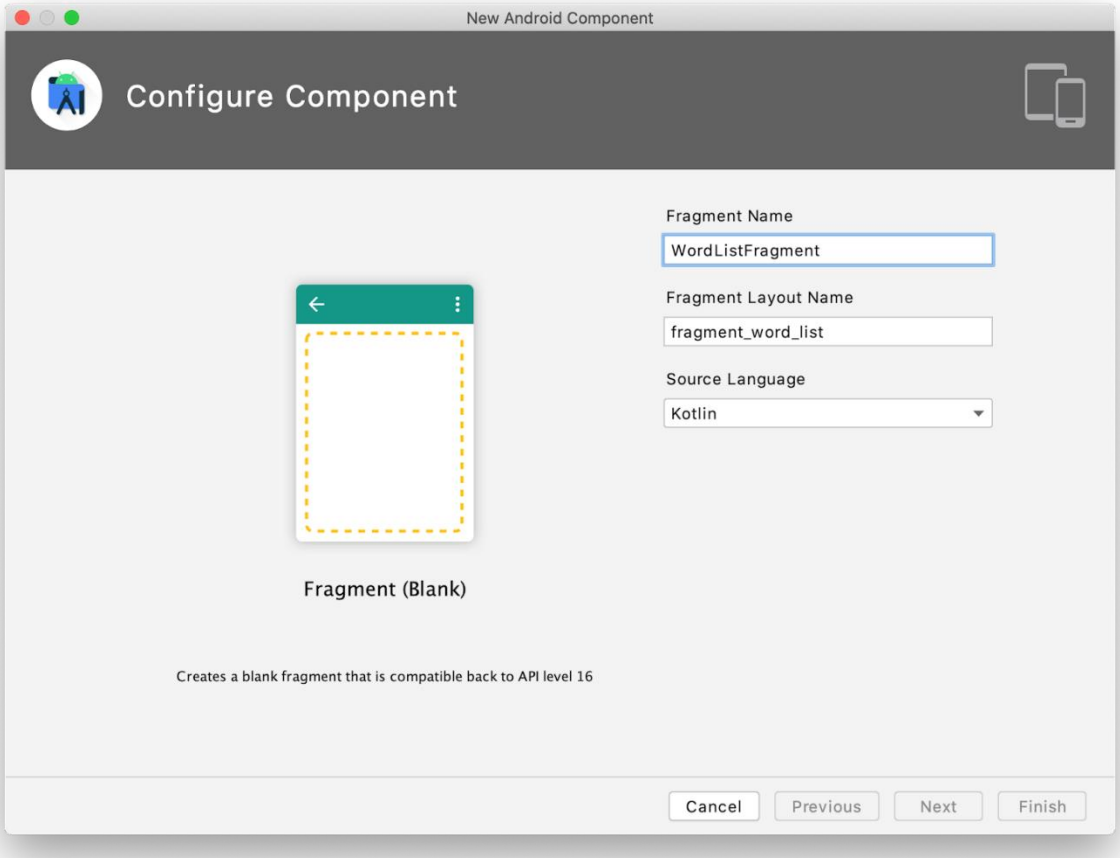
Tiếp theo, ghi đè phương thức `onOptionsItemSelected()`, phương thức này sẽ được gọi khi người dùng chọn một mục trong menu. Trong trường hợp người dùng chọn nút chuyển đổi layout, ta sẽ thay đổi giá trị của thuộc tính `isLinearLayoutManager`, sau đó gọi lại các phương thức `chooseLayout()` và `setIcon()` để cập nhật giao diện người dùng.

```
kotlin
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_switch_layout -> {
            // Đổi trạng thái layout
            isLinearLayoutManager = !isLinearLayoutManager
            // Cập nhật layout và biểu tượng
            chooseLayout()
            setIcon(item)
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Cuối cùng, trong phương thức `onCreate()`, ta sẽ gọi phương thức `chooseLayout()` để thiết lập layout manager cho `RecyclerView`, thay vì thiết lập trực tiếp trong `onCreate()`. Điều này giúp mã nguồn trở nên dễ bảo trì và mở rộng hơn.

```
kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    recyclerView = binding.recyclerView
    // Thiết lập layout manager cho RecyclerView
    chooseLayout()
}
```

Sau khi thực hiện các thay đổi trên, khi chạy ứng dụng, người dùng sẽ có thể chuyển đổi giữa chế độ hiển thị danh sách và lưới bằng cách nhấn vào nút chuyển đổi trong app bar. Biểu tượng của nút này cũng sẽ được cập nhật tự động tùy thuộc vào trạng thái layout hiện tại, giúp người dùng dễ dàng nhận biết chế độ hiển thị của ứng dụng.



2. The generated Kotlin classes for both fragments contain a lot of boilerplate code commonly used when implementing fragments. However, as you're learning about fragments for the first time, go ahead and delete everything except the class declaration for `LetterListFragment` and `WordListFragment` from both files. We'll walk you through implementing the fragments from scratch so that you know how all of the code works. After deleting the boilerplate code, the Kotlin files should look as follows.

LetterListFragment.kt

```
package com.example.wordsapp

import androidx.fragment.app.Fragment

class LetterListFragment : Fragment() {

}
```

WordListFragment.kt

```
package com.example.wordsapp
```

```
import androidx.fragment.app.Fragment

class WordListFragment : Fragment() {

}
```

3. Copy the contents of `activity_main.xml` into `fragment_letter_list.xml` and the contents of `activity_detail.xml` into `fragment_word_list.xml`. Update `tools:context` in `fragment_letter_list.xml` to `.LetterListFragment` and `tools:context` in `fragment_word_list.xml` to `.WordListFragment`.

After the changes, the fragment layout files should look as follows.

fragment_letter_list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".LetterListFragment">

   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/recycler_view"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:clipToPadding="false"
       android:padding="16dp" />

</FrameLayout>
```

fragment_word_list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".WordListFragment">

   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/recycler_view"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:clipToPadding="false"
       android:padding="16dp"
       tools:listitem="@layout/item_view" />

</FrameLayout>
```

5. Implement LetterListFragment

As with activities, you need to inflate the layout and bind individual views. There are just a few minor differences when working with the fragment lifecycle. We'll walk you through the process for setting up the `LetterListFragment`, and then you'll get the chance to do the same for `WordListFragment`.

To implement view binding in `LetterListFragment`, you first need to get a nullable reference to `FragmentLetterListBinding`. Binding classes like this are generated by Android Studio for each layout file, when the `viewBinding` property is enabled under the `buildFeatures` section of the **build.gradle** file. You just need to assign properties in your fragment class for each view in the `FragmentLetterListBinding`.

The type should be `FragmentLetterListBinding?` and it should have an initial value of `null`. Why make it nullable? Because you can't inflate the layout until `onCreateView()` is called. There's a period of time in-between when the instance of `LetterListFragment` is created (when its lifecycle begins with `onCreate()`) and when this property is actually usable. Also keep in mind that fragments' views can be created and destroyed several times throughout the fragment's lifecycle. For this reason you also need to reset the value in another lifecycle method, `onDestroyView()`.

1. In `LetterListFragment.kt`, start by getting a reference to the `FragmentLetterListBinding`, and name the reference `_binding`.

```
private var _binding: FragmentLetterListBinding? = null
```

Because it's nullable, every time you access a property of `_binding`, (e.g. `_binding?.someView`) you need to include the `?` for null safety. However, that doesn't mean you have to litter your code with question marks just because of one null value. If you're certain a value won't be null when you access it, you can append `!!` to its type name. Then you can access it like any other property, without the `?` operator.

NOTE: When making a variable nullable using `!!`, it's a good idea to limit its usage to only one or a few places where you know the value won't be null, just like you know `_binding` will have a value after it is assigned in `onCreateView()`. Accessing a nullable value in this manner is dangerous and can lead to crashes, so use sparingly, if at all.

2. Create a new property, called binding (without the underscore) and set it equal to `_binding!!`.

```
private val binding get() = _binding!!
```

Here, `get()` means this property is "get-only". That means you can **get** the value, but once assigned (as it is here), you can't assign it to something else.

NOTE: In Kotlin, and programming in general, you'll often encounter property names preceded by an underscore. This typically means that the property isn't intended to be accessed directly. In your case, you access the view binding in `LetterListFragment` with the binding property. However, the `_binding` property does not need to be accessed outside of `LetterListFragment`.

3. To display the options menu, override `onCreate()`. Inside `onCreate()` call `setHasOptionsMenu()` passing in `true`.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setHasOptionsMenu(true)
}
```

4. Remember that with fragments, the layout is inflated in `onCreateView()`. Implement `onCreateView()` by inflating the view, setting the value of `_binding`, and returning the root view.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    _binding = FragmentLetterListBinding.inflate(inflater, container, false)
    val view = binding.root
    return view
}
```

5. Below the `binding` property, create a property for the recycler view.

```
private lateinit var recyclerView: RecyclerView
```

6. Then set the value of the `recyclerView` property in `onViewCreated()`, and call `chooseLayout()` like you did in `MainActivity`. You'll move the `chooseLayout()` method into `LetterListFragment` soon, so don't worry that there's an error.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    recyclerView = binding.recyclerView
    chooseLayout()
}
```

Notice how the binding class already created a property for `recyclerView`, and you don't need to call `findViewById()` for each view.

7. Finally, in `onDestroyView()`, reset the `_binding` property to `null`, as the view no longer exists.

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

8. The only other thing to note is there are some subtle differences with the `onCreateOptionsMenu()` method when working with fragments. While the `Activity` class has a global property called `menuInflater`, `Fragment` does not have this property. The menu inflater is instead passed into `onCreateOptionsMenu()`. Also note that the `onCreateOptionsMenu()` method used with fragments doesn't require a return statement. Implement the method as shown:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.layout_menu, menu)

    val layoutButton = menu.findItem(R.id.action_switch_layout)
    setIcon(layoutButton)
}
```

9. Move the remaining code for `chooseLayout()`, `setIcon()`, and `onOptionsItemSelected()` from `MainActivity` as-is. The only other difference to note is that, unlike an activity, a fragment is not a `Context`. You can't pass in `this` (referring to the fragment object) as the layout manager's context. However, fragments provide a `context` property you can use instead. The rest of the code is identical to `MainActivity`.

```
private fun chooseLayout() {
    when (isLinearLayoutManager) {
        true -> {
            recyclerView.layoutManager = LinearLayoutManager(context)
            recyclerView.adapter = LetterAdapter()
        }
        false -> {
            recyclerView.layoutManager = GridLayoutManager(context, 4)
            recyclerView.adapter = LetterAdapter()
        }
    }
}

private fun setIcon(menuItem: MenuItem?) {
    if (menuItem == null)
        return

    menuItem.icon =
        if (isLinearLayoutManager)
            ContextCompat.getDrawable(this.requireContext(),
R.drawable.ic_grid_layout)
        else ContextCompat.getDrawable(this.requireContext(),
R.drawable.ic_linear_layout)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_switch_layout -> {
            isLinearLayoutManager = !isLinearLayoutManager
            chooseLayout()
            setIcon(item)

            return true
```

```
        }

        else -> super.onOptionsItemSelected(item)
    }
}
```

Note: `requireContext()` returns the `Context` this fragment is currently associated with.

10. Finally, copy over the `isLinearLayoutManager` property from `MainActivity`. Put this right below the declaration of the `recyclerView` property.

```
private var isLinearLayoutManager = true
```

11. Now that all the functionality has been moved to `LetterListFragment`, all the `MainActivity` class needs to do is inflate the layout so that the fragment is displayed in the view. Go ahead and delete everything except `onCreate()` from `MainActivity`. After the changes, `MainActivity` should contain only the following.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
}
```

Your turn

That's it for migrating `MainActivity` to `LettersListFragment`. Migrating the `DetailActivity` is almost identical. Perform the following steps to migrate the code to `WordListFragment`.

1. Copy the companion object from `DetailActivity` to `WordListFragment`. Make sure the reference to `SEARCH_PREFIX` in `WordAdapter` is updated to reference `WordListFragment`.
2. Add a `_binding` variable. The variable should be nullable and have `null` as its initial value.
3. Add a get-only variable called binding equal to the `_binding` variable.
4. Inflate the layout in `onCreateView()`, setting the value of `_binding` and returning the root view.
5. Perform any remaining setup in `onViewCreated()`: get a reference to the recycler view, set its layout manager and adapter, and add its item decoration. You'll need to get the letter from the intent. As fragments don't have an `intent` property and shouldn't normally access the intent of the parent activity. For now, you refer to `activity.intent` (rather than `intent` in `DetailActivity`) to get the extras.
6. Reset `_binding` to null in `onDestroyView`.
7. Delete the remaining code from `DetailActivity`, leaving only the `onCreate()` method.

Try to go through the steps on your own before moving on. A detailed walkthrough is available on the next step.

6. Convert DetailActivity to WordListFragment

Hopefully you enjoyed getting the chance to migrate `DetailActivity` to `WordListFragment`. This is almost identical to migrating `MainActivity` to `LetterListFragment`. If you got stuck at any point, the steps are summarized below.

1. First, copy the companion object to `WordListFragment`.

```
companion object {
   val LETTER = "letter"
   val SEARCH_PREFIX = "https://www.google.com/search?q="
}
```

2. Then in `LetterAdapter`, in the `onClickListener()` where you perform the intent, you need to update the call to `putExtra()`, replacing `DetailActivity.LETTER` with `WordListFragment.LETTER`.

```
intent.putExtra(WordListFragment.LETTER, holder.button.text.toString())
```

3. Similarly, in `WordAdapter` you need to update the `onClickListener()` where you navigate to the search results for the word, replacing `DetailActivity.SEARCH_PREFIX` with `WordListFragment.SEARCH_PREFIX`.

```
val queryUrl: Uri = Uri.parse("${WordListFragment.SEARCH_PREFIX}${item}")
```

4. Back in `WordListFragment`, you add a binding variable of type `FragmentWordListBinding?`.

```
private var _binding: FragmentWordListBinding? = null
```

5. You then create a get-only variable so that you can reference views without having to use `?`.

```
private val binding get() = _binding!!
```

6. Then you inflate the layout, assigning the `_binding` variable and returning the root view. Remember that for fragments you do this in `onCreateView()`, not `onCreate()`.

```
override fun onCreateView(
   inflater: LayoutInflater,
   container: ViewGroup?,
   savedInstanceState: Bundle?
): View? {
   _binding = FragmentWordListBinding.inflate(inflater, container, false)
```

```
    return binding.root
}
```

7. Next, you implement `onViewCreated()`. This is almost identical to configuring the `recyclerView` in `onCreate()` in the `DetailActivity`. However, because fragments don't have direct access to the `intent`, you need to reference it with `activity.intent`. You have to do this in `onViewCreated()` however, as there's no guarantee the activity exists earlier in the lifecycle.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    val recyclerView = binding.recyclerView
    recyclerView.layoutManager = LinearLayoutManager(requireContext())
    recyclerView.adapter =
WordAdapter(activity?.intent?.extras?.getString(LETTER).toString(),
requireContext())

    recyclerView.addItemDecoration(
        DividerItemDecoration(context, DividerItemDecoration.VERTICAL)
    )
}
```

8. Finally, you can reset the `_binding` variable in `onDestroyView()`.

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

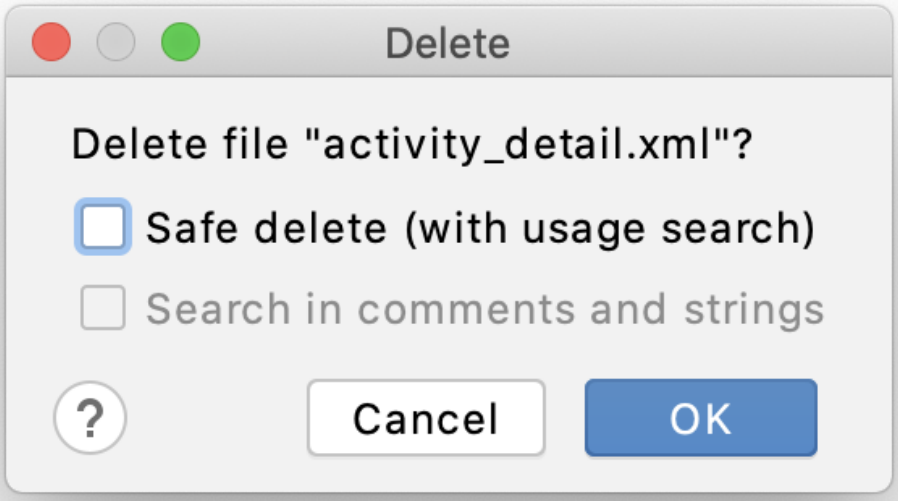
9. With all this functionality moved into WordListFragment, you can now delete the code from DetailActivity. All that should be left is the onCreate() method.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityDetailBinding.inflate(layoutInflater)
    setContentView(binding.root)
}
```

Remove DetailActivity

Now that you've successfully migrated the functionality of `DetailActivity` into `WordListFragment`, you no longer need `DetailActivity`. You can go ahead and delete both the `DetailActivity.kt` and `activity_detail.xml` as well as make a small change to the manifest.

1. First, delete `DetailActivity.kt`



4. Finally, as `DetailActivity` no longer exists, remove the following from `AndroidManifest.xml`.

```
<activity
   android:name=".DetailActivity"
   android:parentActivityName=".MainActivity" />
```

After deleting the detail activity, you're left with two fragments (LetterListFragment and WordListFragment) and a single activity (MainActivity). In the next section, you'll learn about the Jetpack Navigation component and edit `activity_main.xml` so that it can display and navigate between fragments, rather than host a static layout.

7. Jetpack Navigation Component

Android Jetpack provides the *Navigation component* to help you handle any navigation implementation, simple or complex, in your app. The Navigation component has three key parts which you'll use to implement navigation in the **Words** app.

- Navigation Graph: The navigation graph is an XML file that provides a visual representation of navigation in your app. The file consists of *destinations* which correspond to individual activities and fragments as well as actions between them which can be used in code to navigate from one destination to another. Just like with layout

files, Android Studio provides a visual editor to add destinations and actions to the navigation graph.

- `NavHost`: A `NavHost` is used to display destinations from a navigation graph within an activity. When you navigate between fragments, the destination shown in the `NavHost` is updated. You'll use a built-in implementation, called `NavHostFragment`, in your `MainActivity`.
- `NavController`: The `NavController` object lets you control the navigation between destinations displayed in the `NavHost`. When working with intents, you had to call startActivity to navigate to a new screen. With the Navigation component, you can call the `NavController`'s `navigate()` method to swap the fragment that's displayed. The `NavController` also helps you handle common tasks like responding to the system "up" button to navigate back to the previously displayed fragment.

Navigation Dependency

1. In the project-level `build.gradle` file, in **buildscript > ext**, below `material_version` set the `nav_version` equal to `2.5.2`.

```
buildscript {
    ext {
        appcompat_version = "1.5.1"
        constraintlayout_version = "2.1.4"
        core_ktx_version = "1.9.0"
        kotlin_version = "1.7.10"
        material_version = "1.7.0-alpha2"
        nav_version = "2.5.2"
    }

    ...
}
```

2. In the app-level `build.gradle` file, add the following to the dependencies group:

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

Safe Args Plugin

When you first implemented navigation in the **Words** app, you used an explicit intent between the two activities. To pass data between the two activities, you called the `putExtra()` method, passing in the selected letter.

Before you start implementing the Navigation component into the **Words** app, you'll also add something called **Safe Args**—a Gradle plugin that will assist you with type safety when passing data between fragments.

Perform the following steps to integrate SafeArgs into your project.

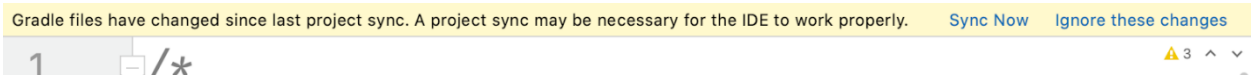
1. In the top-level `build.gradle` file, in **buildscript > dependencies**, add the following classpath.

```
classpath "androidx.navigation:navigation-safe-args-gradle-
plugin:$nav_version"
```

2. In the app-level `build.gradle` file, within `plugins` at the top, add `androidx.navigation.safeargs.kotlin`.

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-kapt'
    id 'androidx.navigation.safeargs.kotlin'
}
```

3. Once you've edited the Gradle files, you may see a yellow banner at the top asking you to sync the project. Click "**Sync Now**" and wait a minute or two while Gradle updates your project's dependencies to reflect your changes.



Once syncing is complete, you're ready to move on to the next step where you'll add a navigation graph.

8. Using the Navigation Graph

Now that you have a basic familiarity with fragments and their lifecycle, it's time for things to get a bit more interesting. The next step is to incorporate the Navigation component. The *navigation component* simply refers to the collection of tools for implementing navigation, particularly between fragments. You'll be working with a new visual editor to help implement navigation between fragments; the Navigation Graph (or NavGraph for short).

What is a Navigation Graph?

The Navigation Graph (or NavGraph for short) is a virtual mapping of your app's navigation. Each screen, or fragment in your case, becomes a possible "destination" that can be navigated to. A `NavGraph` can be represented by an XML file showing how each destination relates to one another.

Behind the scenes, this actually creates a new instance of the `NavGraph` class. However, destinations from the navigation graph are displayed to the user by the `FragmentContainerView`. All you need to do is to create an XML file and define the possible destinations. Then you can use the generated code to navigate between fragments.

Use FragmentContainerView in MainActivity

Because your layouts are now contained in `fragment_letter_list.xml` and `fragment_word_list.xml`, your `activity_main.xml` file no longer needs to contain the layout for the first screen in your app. Instead, you'll repurpose `MainActivity` to contain a `FragmentContainerView` to act as the NavHost for your fragments. From this point forward, all the navigation in the app will take place within the `FragmentContainerView`.

1. Replace the content of the `FrameLayout` in **activity_main.xml** that is `androidx.recyclerview.widget.RecyclerView` with a `FragmentContainerView`. Give it an ID of `nav_host_fragment` and set its height and width to `match_parent` to fill the entire frame layout.

Replace this:

```
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        ...
        android:padding="16dp" />
```

With this:

```
<androidx.fragment.app.FragmentContainerView
   android:id="@+id/nav_host_fragment"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
```

2. Below the id attribute, add a `name` attribute and set it to `androidx.navigation.fragment.NavHostFragment`. While you can specify a specific fragment for this attribute, setting it to `NavHostFragment` allows your `FragmentContainerView` to navigate between fragments.

```
android:name="androidx.navigation.fragment.NavHostFragment"
```

3. Below the layout_height and layout_width attributes, add an attribute called `app:defaultNavHost` and set it equal to `"true"`. This allows the fragment container to interact with the navigation hierarchy. For example, if the system back button is pressed, then the container will navigate back to the previously shown fragment, just like what happens when a new activity is presented.

```
app:defaultNavHost="true"
```

4. Add an attribute called `app:navGraph` and set it equal to `"@navigation/nav_graph"`. This points to an XML file that defines how your app's fragments can navigate to one another. For now, the Android studio will show you an unresolved symbol error. You will address this in the next task.

```
app:navGraph="@navigation/nav_graph"
```

5. Finally, because you added two attributes with the app namespace, be sure to add the `xmlns:app` attribute to the `FrameLayout`.

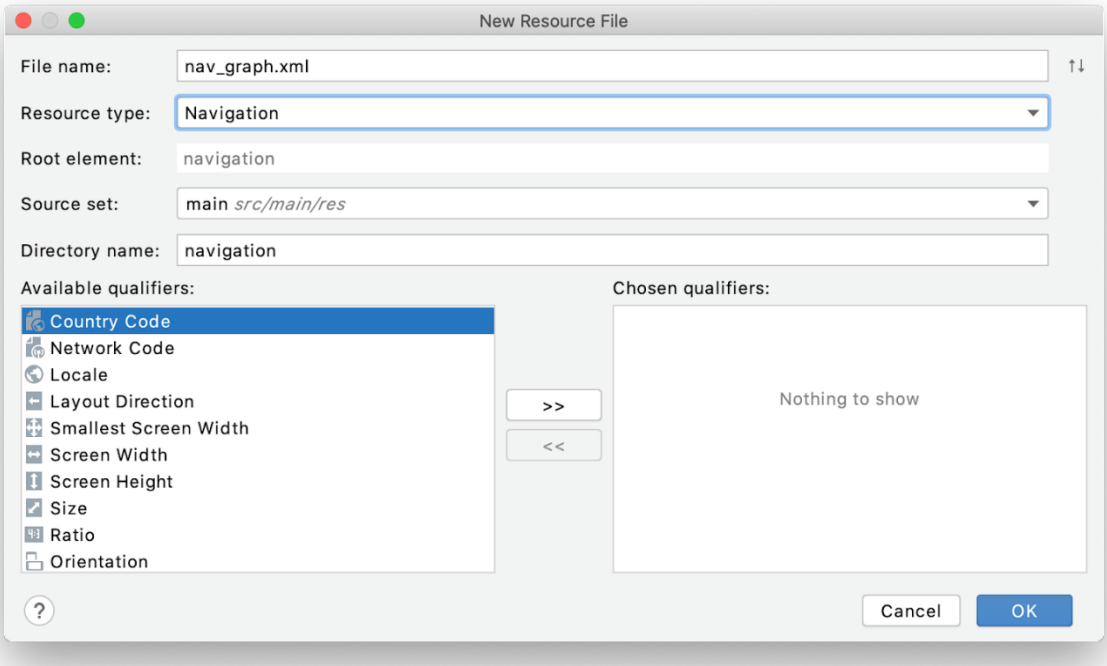
```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

That's all the changes in `activity_main.xml`. Next up, you'll create the `nav_graph` file.

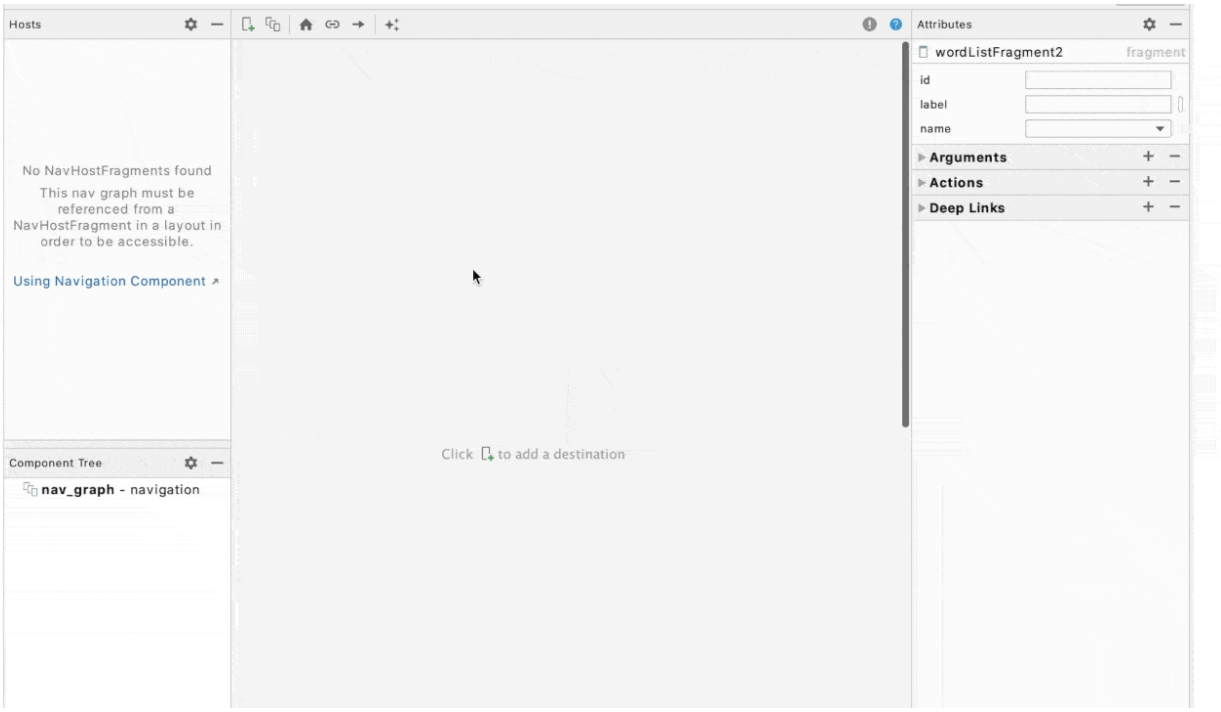
Set Up the Navigation Graph

Add a navigation graph file (**File > New > Android Resource File**) and filling the fields as follows.

- File name: `nav_graph.xml`. This is the same as the name you set for the `app:navGraph` attribute.
- Resource type: **Navigation**. The **Directory name** should then automatically change to `navigation`. A new resource folder called "navigation" will be created.



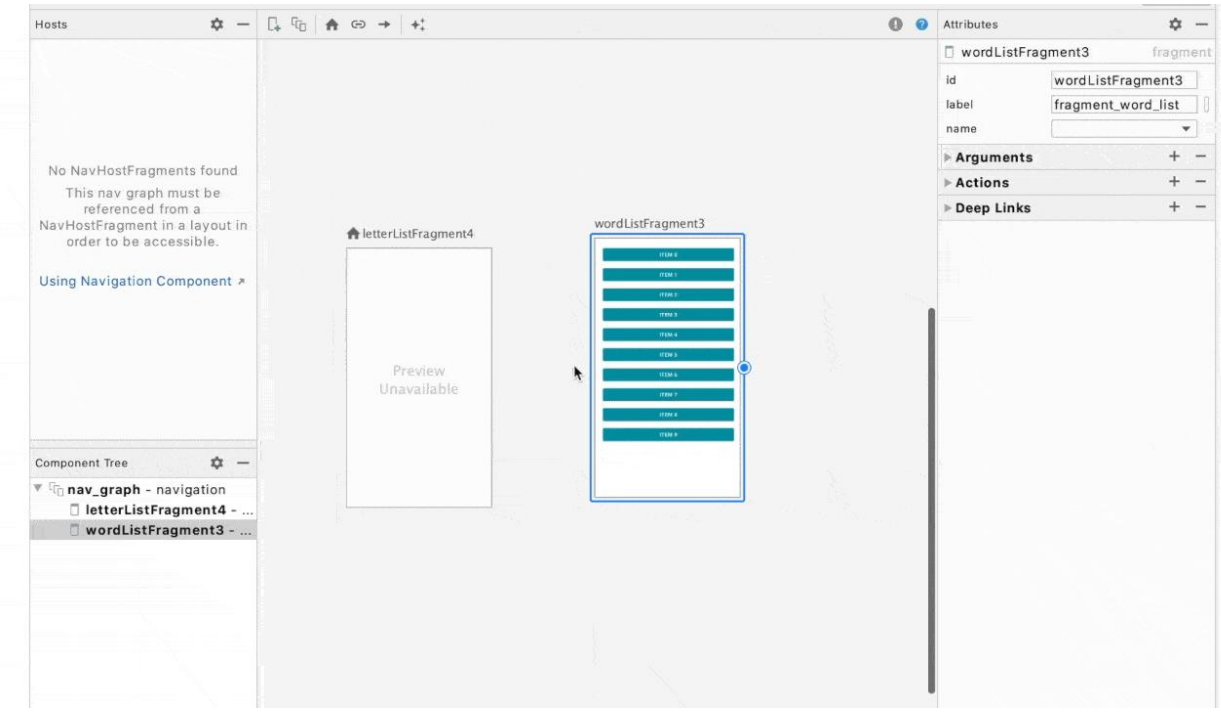
Upon creating the XML file, you're presented with a new visual editor. Because you've already referenced `nav_graph` in the `FragmentContainerView`'s `navGraph` property, to add a new destination, click the new button in the top left of the screen and create a destination for each fragment (one for `fragment_letter_list` and one for `fragment_word_list`).



Once added, these fragments should appear on the navigation graph in the middle of the screen. You can also select a specific destination using the component tree that appears on the left.

Create a navigation action

To create a navigation action between the `letterListFragment` to the `wordListFragment` destinations, hover your mouse over the `letterListFragment` destination and drag from the circle that appears on the right onto the `wordListFragment` destination.



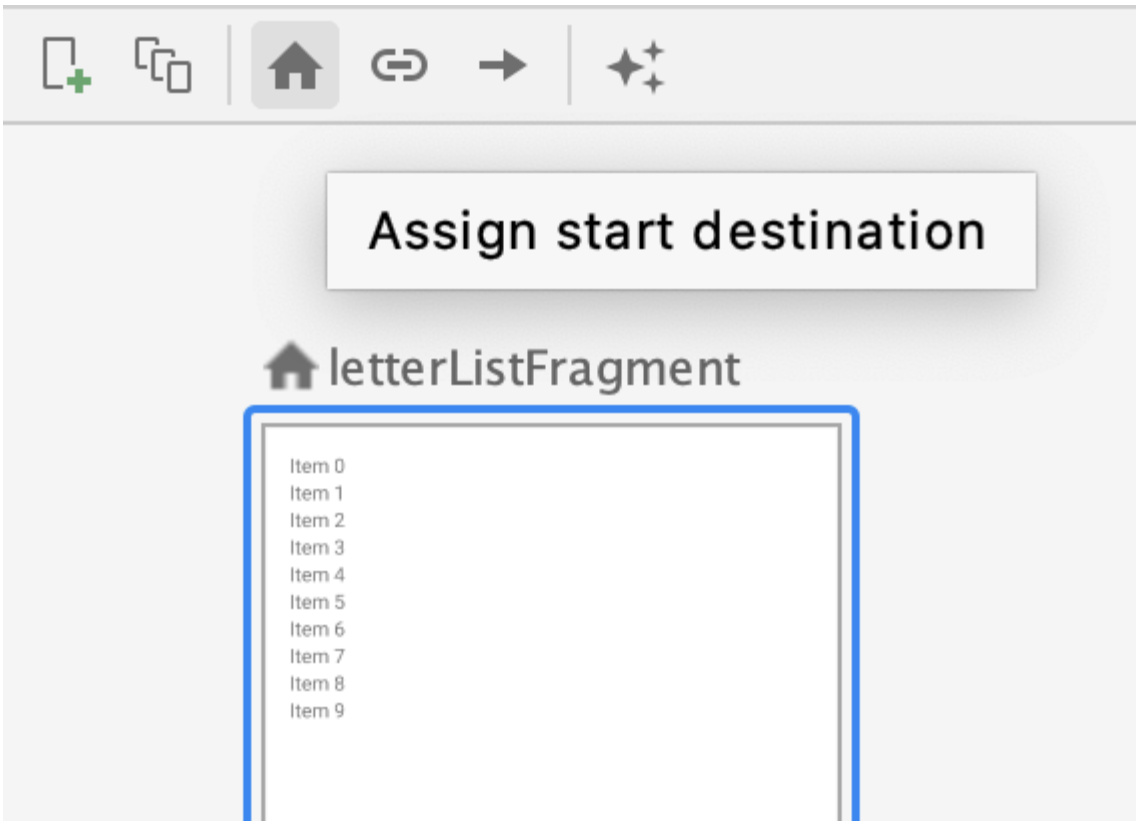
You should now see an arrow has been created to represent the action between the two destinations. Click on the arrow, and you can see in the attributes pane that this action has a name `action_letterListFragment_to_wordListFragment` that can be referenced in code.

Specify Arguments for WordListFragment

When navigating between activities using an intent, you specified an "extra" so that the selected letter could be passed to the `wordListFragment`. Navigation also supports passing parameters between destinations and plus does this in a type safe way.

Select the `wordListFragment` destination and in the attributes pane, under **Arguments**, click the plus button to create a new argument.

The argument should be called `letter` and the type should be `String`. This is where the Safe Args plugin you added earlier comes in. Specifying this argument as a string ensures that a `String` will be expected when your navigation action is performed in code.



1. That's all you need to do with the NavGraph editor for now. At this point, go ahead and build the project. In Android Studio select **Build > Rebuild Project** from the menu bar. This will generate some code based on your navigation graph so that you can use the navigation action you just created.

Perform the Navigation Action

Open up `LetterAdapter.kt` to perform the navigation action. This only requires two steps.

1. Delete the contents of the button's `setOnClickListener()`. Instead, you need to retrieve the navigation action you just created. Add the following to the `setOnClickListener()`.

```
val action =
LetterListFragmentDirections.actionLetterListFragmentToWordListFragment(lette
r = holder.button.text.toString())
```

You probably don't recognize some of these class and function names and that's because they've been automatically generated after you built the project. That's where the Safe Args plugin you added in the first step comes in—the actions created on the NavGraph are turned into code that you can use. The names, however, should be fairly intuitive. `LetterListFragmentDirections` lets you refer to all possible navigation paths starting from the `letterListFragment`.

The function `actionLetterListFragmentToWordListFragment()`

is the specific action to navigate to the `wordListFragment`.

Once you have a reference to your navigation action, simply get a reference to your *NavController* (an object that lets you perform navigation actions) and call `navigate()` passing in the action.

```
holder.view.findNavController().navigate(action)
```

Configure MainActivity

The final piece of setup is in `MainActivity`. There are just a few changes needed in `MainActivity` to get everything working.

1. Create a `navController` property. This is marked as `lateinit` since it will be set in `onCreate`.

```
private lateinit var navController: NavController
```

2. Then, after the call to `setContentView()` in `onCreate()`, get a reference to the `nav_host_fragment` (this is the ID of your `FragmentContainerView`) and assign it to your `navController` property.

```
val navHostFragment = supportFragmentManager
    .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
navController = navHostFragment.navController
```

3. Then in `onCreate()`, call `setupActionBarWithNavController()`, passing in `navController`. This ensures action bar (app bar) buttons, like the menu option in `LetterListFragment` are visible.

```
setupActionBarWithNavController(navController)
```

4. Finally, implement `onSupportNavigateUp()`. Along with setting `defaultNavHost` to `true` in the XML, this method allows you to handle the **up** button. However, your activity needs to provide the implementation.

```
override fun onSupportNavigateUp(): Boolean {
   return navController.navigateUp() || super.onSupportNavigateUp()
}
```

At this point, all the components are in-place to get navigation working with fragments. However, now that navigation is performed using fragments instead of the intent, the intent extra for the letter that you use in `WordListFragment` will no longer work. In the next step, you'll update `WordListFragment`, to get the `letter` argument.

NOTE: Because the `navigateUp()` function might fail, it returns a `Boolean` for whether or not it succeeds. However, you only need to call `super.onSupportNavigateUp()` if `navigateUp()`

returns `false`. This works because of the `||` operator only requires one of the conditions to be true, so if `navigateUp()` returns `true`, the right side of the `||` expression is never executed. If, however, `navigateUp()` is false, then the implementation in the parent class is called. This is called short-circuit evaluation and is a nice little programming trick to know about.

9. Getting Arguments in WordListFragment

Previously, you referenced `activity?.intent` in `WordListFragment` to access the `letter` extra. While this works, this is not a best practice, since fragments can be embedded in other layouts, and in a larger app, it's much harder to assume which activity the fragment belongs to. Furthermore, when navigation is performed using `nav_graph` and safe arguments are used, there are no intents, so trying to access intent extras is simply not going to work.

Thankfully, accessing safe arguments is pretty straightforward, and you don't have to wait until `onViewCreated()` is called either.

1. In `WordListFragment`, create a `letterId` property. You can mark this as lateinit so that you don't have to make it nullable.

```
private lateinit var letterId: String
```

2. Then override `onCreate()` (not `onCreateView()` or `onViewCreated()`!), add the following:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    arguments?.let {
        letterId = it.getString(LETTER).toString()
    }
}
```

Because it's possible for `arguments` to be optional, notice you call `let()` and pass in a lambda. This code will execute assuming `arguments` is not null, passing in the non null arguments for the `it` parameter. If `arguments` is `null`, however, the lambda will not execute.

```
arguments?.let { it: Bundle
    letterId = it.getString(LETTER).toString()
}
```

While not part of the actual code, Android Studio provides a helpful hint to make you aware of the `it` parameter.

What exactly is a `Bundle`? Think of it as a key-value pair used to pass data between classes, such as activities and fragments. Actually, you've already used a bundle when you called `intent?.extras?.getString()` when performing an intent in the first version of this app. Getting the string from arguments when working with fragments works exactly the same way.

3. Finally, you can access the `letterId` when you set the recycler view's adapter. Replace `activity?.intent?.extras?.getString(LETTER).toString()` in `onViewCreated()` with `letterId`.

```
recyclerView.adapter = WordAdapter(letterId, requireContext())
```

You did it! Take a moment to run your app. It's now able to navigate between two screens, without any intents, and all in a single activity.

10. Update Fragment Labels

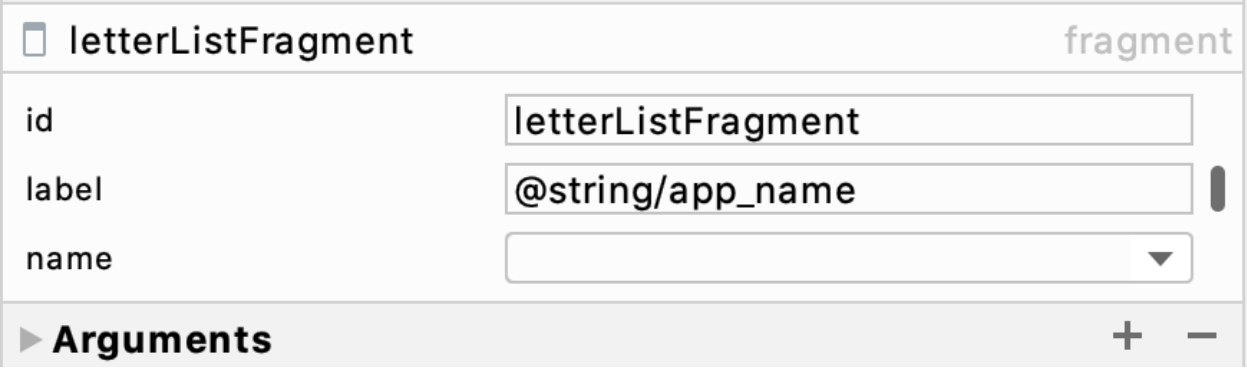
You've successfully converted both screens to use fragments. Before any changes were made, the app bar for each fragment had a descriptive title for each activity contained in the app bar. However, after converting to use fragments, this title is missing from the detail activity.

Fragments have a property called `"label"` where you can set the title which the parent activity will know to use in the app bar.

1. In `strings.xml`, after the app name, add the following constant.

```
<string name="word_list_fragment_label">Words That Start With
{letter}</string>
```

2. You can set the label for each fragment on the navigation graph. Go back into `nav_graph.xml` and select `letterListFragment` in the component tree, and in the attributes pane, set the label to the `app_name` string:



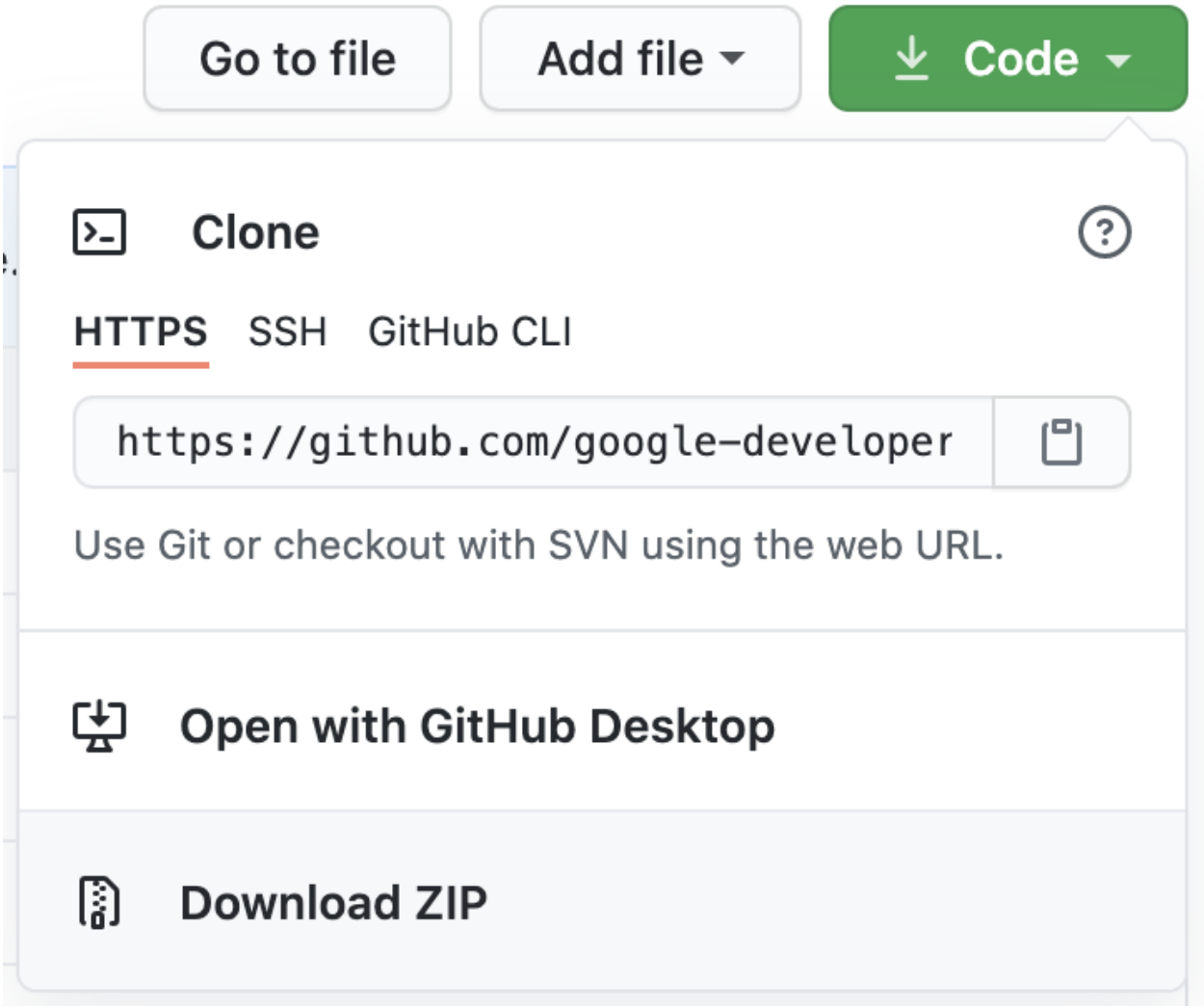
3. Select `wordListFragment` and set the label to `word_list_fragment_label`:

Congratulations on making it this far! Run your app one more time and you should see everything just as it was at the start of the codelab, only now, all your navigation is hosted in a single activity with a separate fragment for each screen.

11. Solution code

The solution code for this codelab is in the project shown below.

Solution Code URL: https://github.com/google-developer-training/android-basics-kotlin-words-app



4. In the popup, click the **Download ZIP** button to save the project to your computer. Wait for the download to complete.
5. Locate the file on your computer (likely in the **Downloads** folder).
6. Double-click the ZIP file to unpack it. This creates a new folder that contains the project files.

Open the project in Android Studio

1. Start Android Studio.
2. In the **Welcome to Android Studio** window, click **Open**.

Lab 7.1. Stages of the activity lifecycle

3. Explore the lifecycle methods and add basic logging

Chu trình sống (Lifecycle) của Activity

Chu trình sống của một Activity tương tự như vòng đời của một sinh vật, như vòng đời của một con bướm. Activity có nhiều trạng thái khác nhau từ khi được khởi tạo, hoạt động cho đến khi bị hủy và bộ nhớ của nó được hệ thống thu hồi.

Vai trò của các phương thức callback trong Activity

Các phương thức callback trong lớp `Activity` (hoặc các lớp con như `AppCompatActivity`) được Android gọi tự động khi Activity thay đổi trạng thái. Bạn có thể ghi đè các phương thức này để chạy mã khi trạng thái của Activity thay đổi.

Quan sát phương thức `onCreate()` trong ứng dụng DessertClicker

Phương thức `onCreate()` là nơi khởi tạo các thành phần ban đầu cho Activity. Sau khi `onCreate()` được thực thi, Activity được coi là đã được tạo. Đây là phương thức **bắt buộc phải ghi đè** và cần gọi `super.onCreate()` để hoàn tất việc khởi tạo Activity.

Ví dụ:

```
kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("MainActivity", "onCreate Called")
}
```

Ghi log bằng `Log.d()` để theo dõi chu trình sống

Mục đích: Xác định khi nào phương thức `onCreate()` được gọi bằng cách ghi thông báo vào Logcat.

Cách thực hiện:

- **Thêm lệnh ghi log:**

  ```
  kotlin
  Log.d("MainActivity", "onCreate Called")
  ```

- **Ý nghĩa:**
  - **TAG:** "MainActivity" giúp dễ dàng lọc log trong Logcat.
  - **MESSAGE:** "onCreate Called" mô tả sự kiện.

- **Tối ưu hóa:** Sử dụng hằng số `TAG`:

```
kotlin
const val TAG = "MainActivity"
Log.d(TAG, "onCreate Called")
```

Kiểm tra log trong Logcat

Bước thực hiện:

1. Chạy ứng dụng DessertClicker.
2. Mở tab **Logcat** trong Android Studio.
3. Gõ `D/MainActivity` vào ô tìm kiếm.

Kết quả: Logcat hiển thị các thông báo ghi lại thời gian, tên package, TAG và thông điệp log. Điều này xác nhận rằng phương thức `onCreate()` đã được gọi thành công.

Triển khai phương thức `onStart()`

Phương thức `onStart()` trong chu trình sống của Activity được gọi ngay sau `onCreate()`. Khi `onStart()` chạy, Activity trở nên **hiển thị trên màn hình**. Không giống như `onCreate()`, phương thức này có thể được gọi **nhiều lần** trong vòng đời của Activity.

Phương thức `onStart()` thường được kết hợp với `onStop()`. Nếu người dùng chuyển về màn hình chính, Activity sẽ dừng lại và không còn hiển thị.

Các bước triển khai `onStart()`

1. **Ghi đè phương thức `onStart()`:**
   - Nhấn **Control+O** (hoặc Command+O trên Mac).
   - Tìm và chọn phương thức `onStart()` từ danh sách.
   - Mã mặc định sau khi chèn:

```
kotlin
override fun onStart() {
    super.onStart()
}
```

2. **Thêm hằng số `TAG`:**
   - Khai báo hằng số `TAG` ở đầu file:

```
kotlin
Sao chép mã
const val TAG = "MainActivity"
```

3. **Thêm log trong `onStart()`:**
    - Sửa đổi phương thức `onStart()`:

```
kotlin
override fun onStart() {
    super.onStart()
    Log.d(TAG, "onStart Called")
}
```

Kiểm tra Logcat

1. **Chạy ứng dụng DessertClicker** và mở **Logcat**.
2. **Lọc log:** Nhập `D/MainActivity` vào ô tìm kiếm.
3. **Kết quả:**
    - Log hiển thị lần lượt các phương thức `onCreate()` và `onStart()` được gọi khi ứng dụng khởi động.
    - Ví dụ log:

```
bash
16:19:59.125 31107-31107/com.example.android.dessertclicker
D/MainActivity: onCreate Called
16:19:59.372 31107-31107/com.example.android.dessertclicker
D/MainActivity: onStart Called
```

4. **Nhấn nút Home:** Trở về màn hình chính của thiết bị, sau đó mở lại ứng dụng từ màn hình đa nhiệm.
    - Kết quả:
        - `onStart()` được gọi lại và log xuất hiện lần thứ hai.
        - `onCreate()` không được gọi lại.
        - Log hiển thị:

```
bash
16:20:11.319 31107-31107/com.example.android.dessertclicker
D/MainActivity: onStart Called
```

Lưu ý khi xoay màn hình

Khi xoay màn hình thiết bị, có thể xảy ra hành vi bất thường trong chu trình sống của Activity. Điều này sẽ được giải thích trong các bước tiếp theo của bài học.

Thêm các log cho các phương thức khác trong chu trình sống

Ở bước này, bạn sẽ triển khai log cho các phương thức chu trình sống khác của Activity để hiểu rõ hơn cách chúng được gọi trong các trạng thái khác nhau của ứng dụng.

Các bước triển khai

1. **Ghi đè các phương thức chu trình sống còn lại:**
   - Trong `MainActivity.kt`, ghi đè các phương thức sau và thêm log cho từng phương thức:

```
kotlin
override fun onResume() {
    super.onResume()
    Log.d(TAG, "onResume Called")
}

override fun onPause() {
    super.onPause()
    Log.d(TAG, "onPause Called")
}

override fun onStop() {
    super.onStop()
    Log.d(TAG, "onStop Called")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d(TAG, "onDestroy Called")
}

override fun onRestart() {
    super.onRestart()
    Log.d(TAG, "onRestart Called")
}
```

2. **Chạy lại ứng dụng DessertClicker** và mở Logcat.
   - Lọc log bằng cách nhập `D/MainActivity` vào ô tìm kiếm.
   - Lưu ý các log mới được thêm vào, đặc biệt là thứ tự gọi các phương thức.

Quan sát thứ tự gọi các phương thức

1. Khi khởi động ứng dụng từ đầu, bạn sẽ thấy các log sau:

```
makefile
10:27:33.244 D/MainActivity: onCreate Called
10:27:33.453 D/MainActivity: onStart Called
10:27:33.454 D/MainActivity: onResume Called
```

   - **`onCreate()`**: Khởi tạo Activity.
   - **`onStart()`**: Bắt đầu Activity và hiển thị trên màn hình.
   - **`onResume()`**: Hoạt động sẵn sàng để người dùng tương tác.
2. Khi rời khỏi ứng dụng bằng nút Home:

```
makefile
10:28:15.521 D/MainActivity: onPause Called
10:28:15.623 D/MainActivity: onStop Called
```

- **`onPause()`**: Tạm dừng tương tác với người dùng (ứng dụng vẫn hiển thị mờ dần).
- **`onStop()`**: Activity bị ẩn hoàn toàn.

3. Khi mở lại ứng dụng từ màn hình đa nhiệm:

```
makefile
10:28:25.745 D/MainActivity: onRestart Called
10:28:25.746 D/MainActivity: onStart Called
10:28:25.747 D/MainActivity: onResume Called
```

- **`onRestart()`**: Chuẩn bị khởi động lại.
- **`onStart()`**: Hoạt động lại và hiển thị.
- **`onResume()`**: Sẵn sàng cho tương tác.

4. Khi thoát ứng dụng hoàn toàn:

```
makefile
10:28:45.812 D/MainActivity: onPause Called
10:28:45.914 D/MainActivity: onStop Called
10:28:46.017 D/MainActivity: onDestroy Called
```

- **`onDestroy()`**: Hoạt động bị hủy hoàn toàn và bộ nhớ được giải phóng.

Lưu ý về `onResume()`

Mặc dù tên là `onResume()`, nhưng phương thức này cũng được gọi khi ứng dụng khởi động từ đầu, ngay cả khi không có gì để "khôi phục". Đây là trạng thái chuẩn bị cuối cùng để ứng dụng sẵn sàng cho người dùng.

4. Explore lifecycle use cases

Sử dụng app và khám phá các callback trong chu trình sống

Trong trường hợp cơ bản nhất, bạn mở ứng dụng lần đầu tiên và sau đó đóng hoàn toàn ứng dụng để quan sát cách các callback trong chu trình sống được kích hoạt.

Khi chạy ứng dụng DessertClicker lần đầu, các callback sau được gọi lần lượt:

```
makefile
D/MainActivity: onCreate Called
D/MainActivity: onStart Called
D/MainActivity: onResume Called
```

Các bước này bao gồm: khởi tạo ứng dụng (`onCreate()`), bắt đầu hiển thị giao diện (`onStart()`), và sẵn sàng tương tác với người dùng (`onResume()`). Sau khi ứng dụng hiển thị, người dùng có thể thực hiện các hành động như nhấn vào chiếc bánh cupcake để tương tác.

Khi nhấn nút **Back** để thoát ứng dụng, các callback sau được gọi:

```
makefile
D/MainActivity: onPause Called
D/MainActivity: onStop Called
D/MainActivity: onDestroy Called
```

Lúc này, ứng dụng đã dừng hoàn toàn và không còn hiển thị trên màn hình. Phương thức `onDestroy()` được gọi để dọn dẹp bộ nhớ và các tài nguyên không cần thiết, giúp hệ thống có thể giải phóng các tài nguyên này. Đây là bước quan trọng để đảm bảo ứng dụng hoạt động hiệu quả trên thiết bị.

Nếu bạn quay lại ứng dụng từ màn hình **Overview** (màn hình Recent Apps), ứng dụng sẽ được khởi động lại từ đầu. Logcat sẽ hiển thị như sau:

```
makefile
D/MainActivity: onCreate Called
D/MainActivity: onStart Called
D/MainActivity: onResume Called
```

Lý do là vì hoạt động (Activity) trước đó đã bị hủy bỏ hoàn toàn trong bước trước. Khi khởi động lại, Android sẽ tạo một phiên bản mới của Activity và gọi lại các phương thức `onCreate()`, `onStart()`, và `onResume()`. Tất cả trạng thái và log từ phiên bản cũ đều không được giữ lại.

Lưu ý quan trọng:

- `onCreate()` chỉ được gọi một lần khi Activity được khởi tạo lần đầu.
- `onDestroy()` chỉ được gọi một lần trước khi Activity bị hủy hoàn toàn.
- Đây là hai điểm mốc quan trọng trong chu trình sống của Activity, nơi bạn cần xử lý việc khởi tạo và dọn dẹp tài nguyên một cách cẩn thận.

Quản lý chu trình sống khi chuyển đổi giữa các ứng dụng

Trong quá trình sử dụng thiết bị Android, người dùng thường xuyên chuyển đổi giữa các ứng dụng, quay lại màn hình chính hoặc bị gián đoạn bởi các hoạt động khác như cuộc gọi. Những tình huống này không khiến ứng dụng bị đóng hoàn toàn, mà thay vào đó ứng dụng chuyển sang trạng thái chạy nền (background) và có thể quay trở lại trạng thái hiển thị (foreground).

Khi ứng dụng DessertClicker đang chạy và người dùng nhấn nút **Home** để quay lại màn hình chính, ứng dụng không bị tắt mà chuyển vào chế độ chạy nền. Logcat ghi nhận:

```
makefile
D/MainActivity: onPause Called
D/MainActivity: onStop Called
```

Callback `onPause()` được gọi khi ứng dụng mất quyền tập trung, và sau đó là `onStop()` khi ứng dụng không còn hiển thị trên màn hình. Tuy nhiên, `onDestroy()` không được gọi, nghĩa là Activity vẫn được giữ trong bộ nhớ. Hệ thống Android giữ lại tài nguyên của Activity để nhanh chóng hiển thị lại nếu người dùng quay trở lại ứng dụng.

Khi người dùng quay lại ứng dụng thông qua màn hình **Recents**, Logcat hiển thị:

```
makefile
D/MainActivity: onRestart Called
D/MainActivity: onStart Called
D/MainActivity: onResume Called
```

Trong trường hợp này, `onCreate()` không được gọi lại vì Activity chưa bị hủy. Thay vào đó, `onRestart()` được gọi trước khi Activity trở lại trạng thái hiển thị. Điều này cho thấy ứng dụng giữ nguyên trạng thái trước đó, bao gồm cả số bánh đã bán trong trường hợp của DessertClicker.

Nếu người dùng mở một ứng dụng khác từ màn hình **Recents** rồi quay lại DessertClicker, các callback được kích hoạt tương tự như khi nhấn nút **Home**:

```
makefile
D/MainActivity: onPause Called
D/MainActivity: onStop Called
D/MainActivity: onRestart Called
D/MainActivity: onStart Called
D/MainActivity: onResume Called
```

Lưu ý quan trọng:

- `onStart()` và `onStop()` có thể được gọi nhiều lần khi ứng dụng chuyển qua lại giữa trạng thái nền và hiển thị.
- Nếu cần thực hiện các công việc khi ứng dụng chuyển trạng thái, bạn nên ghi đè các phương thức chu trình sống liên quan.
- `onRestart()` chỉ được gọi khi Activity được khởi động lại từ trạng thái đã dừng, không phải khi nó được tạo mới. Đây là nơi thích hợp để đặt các đoạn mã cần chạy khi ứng dụng trở lại foreground mà không khởi tạo lại hoàn toàn.

Ẩn một phần Activity và trạng thái chu trình sống

Khi ứng dụng được khởi động và onStart() được gọi, ứng dụng trở nên **hiển thị** trên màn hình. Khi onResume() được gọi, ứng dụng **nhận được tập trung từ người dùng** (user focus), nghĩa là người dùng có thể tương tác với ứng dụng. Trạng thái mà ứng dụng hoàn toàn hiển thị trên màn hình và có thể được tương tác được gọi là **chu trình sống tương tác** (interactive lifecycle).

Khi ứng dụng chuyển sang chạy nền:

- Sau onPause(), ứng dụng mất quyền tập trung (focus).
- Sau onStop(), ứng dụng không còn hiển thị trên màn hình.

Điểm khác biệt giữa **focus** và **visibility** là quan trọng, vì một Activity có thể vẫn hiển thị một phần trên màn hình nhưng không còn quyền tập trung từ người dùng.

Trong trường hợp ứng dụng DessertClicker, khi nhấn nút **Share** ở góc trên bên phải màn hình:

- Một Activity chia sẻ xuất hiện ở nửa dưới của màn hình.
- Mặc dù Activity chính của ứng dụng DessertClicker vẫn **hiển thị** ở nửa trên màn hình, nhưng nó không còn **focus**, vì quyền tập trung đã chuyển sang Activity chia sẻ.

Các trạng thái trong trường hợp này:

1. Khi Activity chia sẻ xuất hiện:
    - Callback onPause() được gọi trên Activity chính của DessertClicker.
    - Điều này cho biết ứng dụng không còn nhận được sự tương tác từ người dùng, nhưng vẫn hiển thị một phần.
2. Khi đóng Activity chia sẻ:
    - Callback onResume() được gọi trên Activity chính, khôi phục focus và đưa ứng dụng trở lại chu trình tương tác.

Lưu ý:

- Đây là ví dụ minh họa rõ ràng về sự khác biệt giữa trạng thái hiển thị và trạng thái tập trung.
- Nếu ứng dụng của bạn cần xử lý các sự kiện khi mất focus nhưng vẫn hiển thị một phần, bạn nên thực hiện trong phương thức onPause().

5. Explore configuration changes

Quản lý chu trình sống khi xảy ra thay đổi cấu hình

Một trường hợp quan trọng trong việc quản lý chu trình sống của Activity là xử lý khi thiết bị thay đổi cấu hình (configuration changes). Thay đổi cấu hình xảy ra khi trạng thái của thiết bị thay đổi đáng kể, khiến hệ thống buộc phải tắt và khởi động lại Activity để thích nghi. Ví dụ:

- Thay đổi ngôn ngữ thiết bị dẫn đến thay đổi bố cục để phù hợp với hướng văn bản và độ dài chuỗi.
- Kết nối thiết bị với dock hoặc bàn phím vật lý có thể yêu cầu sử dụng kích thước hiển thị hoặc bố cục mới.
- Xoay thiết bị giữa chế độ dọc và ngang đòi hỏi bố cục thay đổi để phù hợp.

Khi ứng dụng DessertClicker chạy, xoay thiết bị dẫn đến các callback chu trình sống được gọi theo thứ tự:

1. `onPause()` → Mất quyền tập trung (focus).
2. `onStop()` → Ngừng hiển thị trên màn hình.
3. `onDestroy()` → Hủy Activity hiện tại.
4. Sau đó, Activity được khởi tạo lại thông qua các callback:
    - `onCreate()` → Tạo Activity mới.
    - `onStart()` và `onResume()` → Hiển thị và sẵn sàng tương tác.

Kết quả: Toàn bộ dữ liệu, như số lượng bánh đã bán và doanh thu, bị đặt lại về giá trị mặc định.

Sử dụng onSaveInstanceState() để lưu dữ liệu

Phương thức `onSaveInstanceState()` được sử dụng để lưu các dữ liệu cần thiết khi Activity bị hủy. Hệ thống gọi phương thức này ngay sau khi Activity bị dừng (onStop) và trước khi bị hủy, nhằm đảm bảo dữ liệu được lưu trong trường hợp có sự thay đổi cấu hình hoặc thiết bị gặp áp lực tài nguyên.

Ví dụ: Khi ứng dụng chuyển sang chạy nền, các callback sau được kích hoạt:

1. `onPause()`
2. `onStop()`
3. `onSaveInstanceState()`.

Cách triển khai:

1. Ghi đè `onSaveInstanceState()` và thêm log để theo dõi:

```
kotlin
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.d(TAG, "onSaveInstanceState Called")
}
```

2. Khai báo các khóa để lưu và truy xuất dữ liệu ở đầu tệp:

```
kotlin
const val KEY_REVENUE = "revenue_key"
const val KEY_DESSERT_SOLD = "dessert_sold_key"
```

3. Trong `onSaveInstanceState()`, sử dụng đối tượng `Bundle` để lưu trữ dữ liệu:
    - Lưu doanh thu:

      ```
      kotlin
      outState.putInt(KEY_REVENUE, revenue)
      ```

    - Lưu số lượng bánh đã bán:

      ```
      kotlin
      outState.putInt(KEY_DESSERT_SOLD, dessertsSold)
      ```

Lưu ý:

- Chỉ lưu trữ dữ liệu nhỏ gọn, chẳng hạn như kiểu `Int` hoặc `Boolean`, để tránh gặp lỗi `TransactionTooLargeException`.
- Hệ thống chỉ lưu trữ `Bundle` này trong bộ nhớ, vì vậy dữ liệu cần phải gọn nhẹ và tối ưu hóa để đảm bảo hiệu năng ứng dụng.

Sử dụng onCreate() để khôi phục dữ liệu từ bundle

Khi một Activity được tái tạo do thay đổi cấu hình hoặc tắt đi và khởi động lại, chúng ta có thể khôi phục lại trạng thái của Activity bằng cách sử dụng dữ liệu đã lưu trong `onSaveInstanceState()` vào phương thức `onCreate()` hoặc `onRestoreInstanceState()`. Dữ liệu lưu trong Bundle sẽ được truyền cho cả hai phương thức này.

Khôi phục dữ liệu trong onCreate()

1. **Kiểm tra sự tồn tại của savedInstanceState**: Trong phương thức `onCreate()`, ta có thể kiểm tra xem dữ liệu có tồn tại trong `savedInstanceState` hay không. Nếu có, điều này chứng tỏ Activity đang được tái tạo lại từ một điểm đã lưu trước đó. Dữ liệu này sẽ được lấy ra từ Bundle và được sử dụng để khôi phục lại các giá trị cần thiết trong Activity.

2. **Cách khôi phục dữ liệu**:
    - Trong phương thức `onCreate()`, sau khi gán giá trị cho biến binding, ta kiểm tra xem `savedInstanceState` có khác null không. Nếu khác null, ta sử dụng phương thức `getInt()` để lấy lại giá trị của các biến đã lưu, ví dụ như `revenue` và `dessertsSold`:

      ```
      kotlin
      if (savedInstanceState != null) {
          revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
          dessertsSold = savedInstanceState.getInt(KEY_DESSERT_SOLD, 0)
      }
      ```

3. **Cách sử dụng `getInt()`**:
    - `getInt()` nhận hai tham số: khóa của giá trị trong bundle và giá trị mặc định nếu không tìm thấy khóa đó. Trong trường hợp này, giá trị mặc định là `0`.
4. **Khôi phục hình ảnh của món tráng miệng**: Phương thức `showCurrentDessert()` quyết định hình ảnh món tráng miệng nào sẽ được hiển thị dựa trên số lượng bánh đã bán. Dựa trên số lượng bánh đã bán, chúng ta chọn món tráng miệng thích hợp để hiển thị. Vì giá trị số lượng bánh đã bán đã được lưu trong bundle, chúng ta không cần phải lưu thêm thông tin về hình ảnh của món tráng miệng.
5. **Thực hiện khôi phục trạng thái hình ảnh món tráng miệng**: Sau khi khôi phục giá trị số lượng bánh và doanh thu, ta gọi phương thức `showCurrentDessert()` để đảm bảo món tráng miệng hiển thị đúng với trạng thái của ứng dụng sau khi xoay màn hình:

   ```
   kotlin
   if (savedInstanceState != null) {
       revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
       dessertsSold = savedInstanceState.getInt(KEY_DESSERT_SOLD, 0)
       showCurrentDessert()
   }
   ```

Kết quả và kiểm tra

- Sau khi thực hiện các bước trên, khi quay màn hình, ứng dụng sẽ hiển thị chính xác số lượng bánh đã bán, doanh thu và hình ảnh món tráng miệng đúng như khi người dùng rời khỏi ứng dụng.

6. Summary

Vòng đời Activity

Vòng đời Activity là một chuỗi các trạng thái mà một Activity trải qua. Vòng đời bắt đầu khi Activity được tạo ra lần đầu tiên và kết thúc khi Activity bị hủy.

Khi người dùng chuyển đổi giữa các Activity hoặc chuyển ra ngoài và quay lại ứng dụng, mỗi Activity sẽ di chuyển qua các trạng thái trong vòng đời Activity.

Mỗi trạng thái trong vòng đời Activity có một phương thức callback tương ứng mà bạn có thể ghi đè trong lớp Activity của mình. Các phương thức vòng đời cơ bản bao gồm:

- `onCreate()`
- `onStart()`
- `onPause()`
- `onRestart()`
- `onResume()`
- `onStop()`
- `onDestroy()`

Để thêm hành vi khi Activity chuyển sang một trạng thái vòng đời, bạn có thể ghi đè các phương thức callback của các trạng thái đó.

Ghi Log với Log

API ghi log trong Android, đặc biệt là lớp `Log`, cho phép bạn viết các thông điệp ngắn hiển thị trong Logcat của Android Studio.

- Sử dụng `Log.d()` để ghi một thông điệp debug. Phương thức này nhận hai đối số: thẻ log (thường là tên lớp) và thông điệp log (một chuỗi ngắn).
- Sử dụng cửa sổ Logcat trong Android Studio để xem các log hệ thống, bao gồm các thông điệp bạn ghi.

Bảo vệ trạng thái Activity

Khi ứng dụng của bạn vào nền (background), ngay sau khi `onStop()` được gọi, dữ liệu của ứng dụng có thể được lưu vào một bundle. Một số dữ liệu ứng dụng, như nội dung của `EditText`, sẽ được lưu tự động.

- Bundle là một đối tượng chứa các cặp khóa và giá trị, trong đó khóa luôn là một chuỗi.
- Sử dụng phương thức callback `onSaveInstanceState()` để lưu các dữ liệu khác vào bundle mà bạn muốn giữ lại, ngay cả khi ứng dụng bị đóng tự động. Để đưa dữ liệu vào bundle, sử dụng các phương thức bắt đầu bằng `put`, ví dụ như `putInt()`.
- Bạn có thể lấy lại dữ liệu từ bundle trong phương thức `onRestoreInstanceState()` hoặc thường xuyên trong `onCreate()`. Phương thức `onCreate()` có một tham số `savedInstanceState` chứa bundle.
- Nếu biến `savedInstanceState` là null, điều này có nghĩa là Activity đã được khởi động mà không có bundle trạng thái và không có dữ liệu trạng thái để lấy lại.
- Để lấy dữ liệu từ bundle, bạn sử dụng các phương thức của `Bundle` bắt đầu bằng `get`, ví dụ như `getInt()`.

Thay đổi cấu hình

Thay đổi cấu hình xảy ra khi trạng thái của thiết bị thay đổi quá mạnh mẽ khiến hệ thống phải hủy và tạo lại Activity.

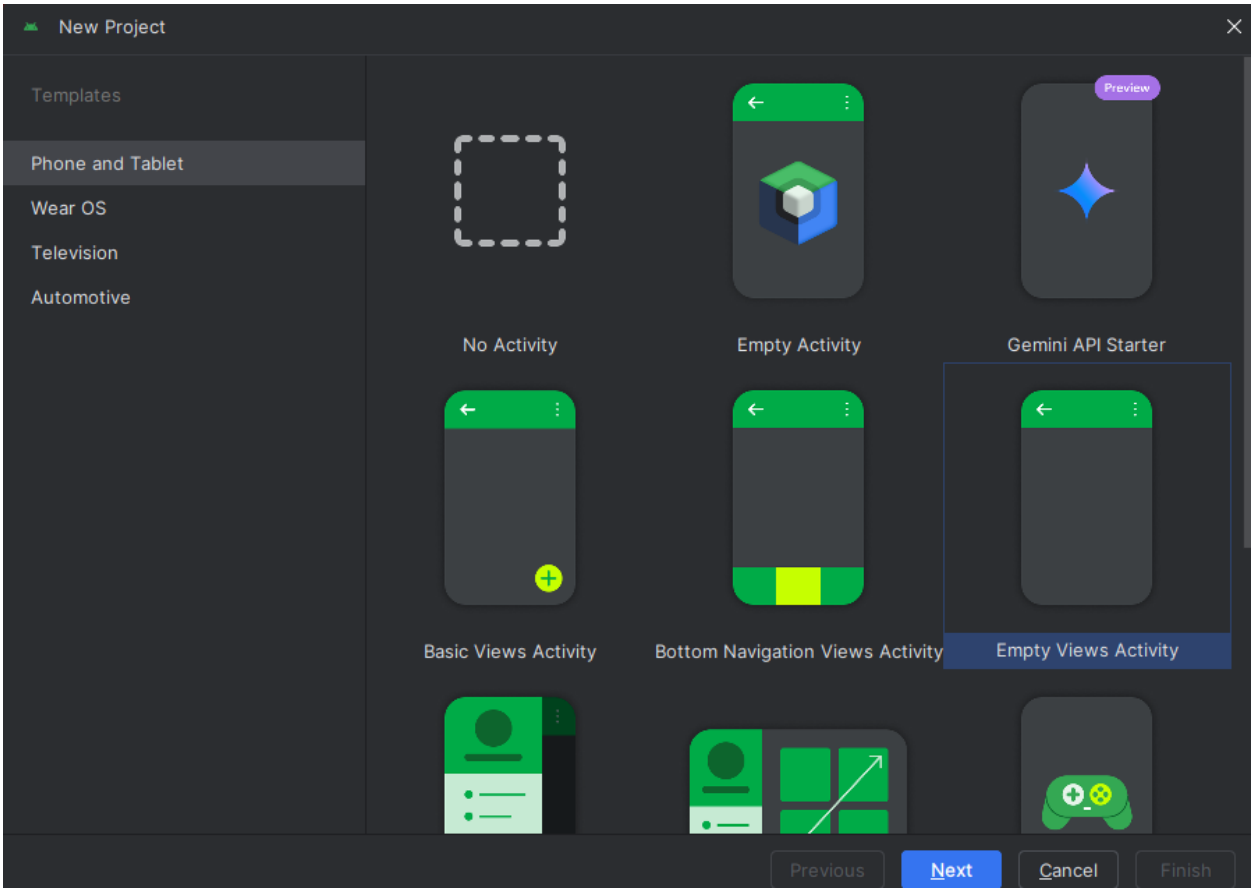
- Ví dụ phổ biến của thay đổi cấu hình là khi người dùng xoay thiết bị từ chế độ dọc sang ngang, hoặc ngược lại. Thay đổi cấu hình cũng có thể xảy ra khi ngôn ngữ của thiết bị thay đổi hoặc khi bàn phím phần cứng được cắm vào.
- Khi thay đổi cấu hình xảy ra, Android sẽ gọi tất cả các phương thức callback shutdown trong vòng đời Activity. Sau đó, Android sẽ khởi động lại Activity từ đầu và gọi tất cả các phương thức callback khởi động của vòng đời Activity.
- Khi Android tắt một ứng dụng vì thay đổi cấu hình, nó sẽ khởi động lại Activity với bundle trạng thái có sẵn được truyền vào `onCreate()`.
- Như trong trường hợp tắt ứng dụng do thay đổi cấu hình, bạn cần lưu trạng thái ứng dụng vào bundle trong `onSaveInstanceState()`.

Báo cáo Life-cycle aware components

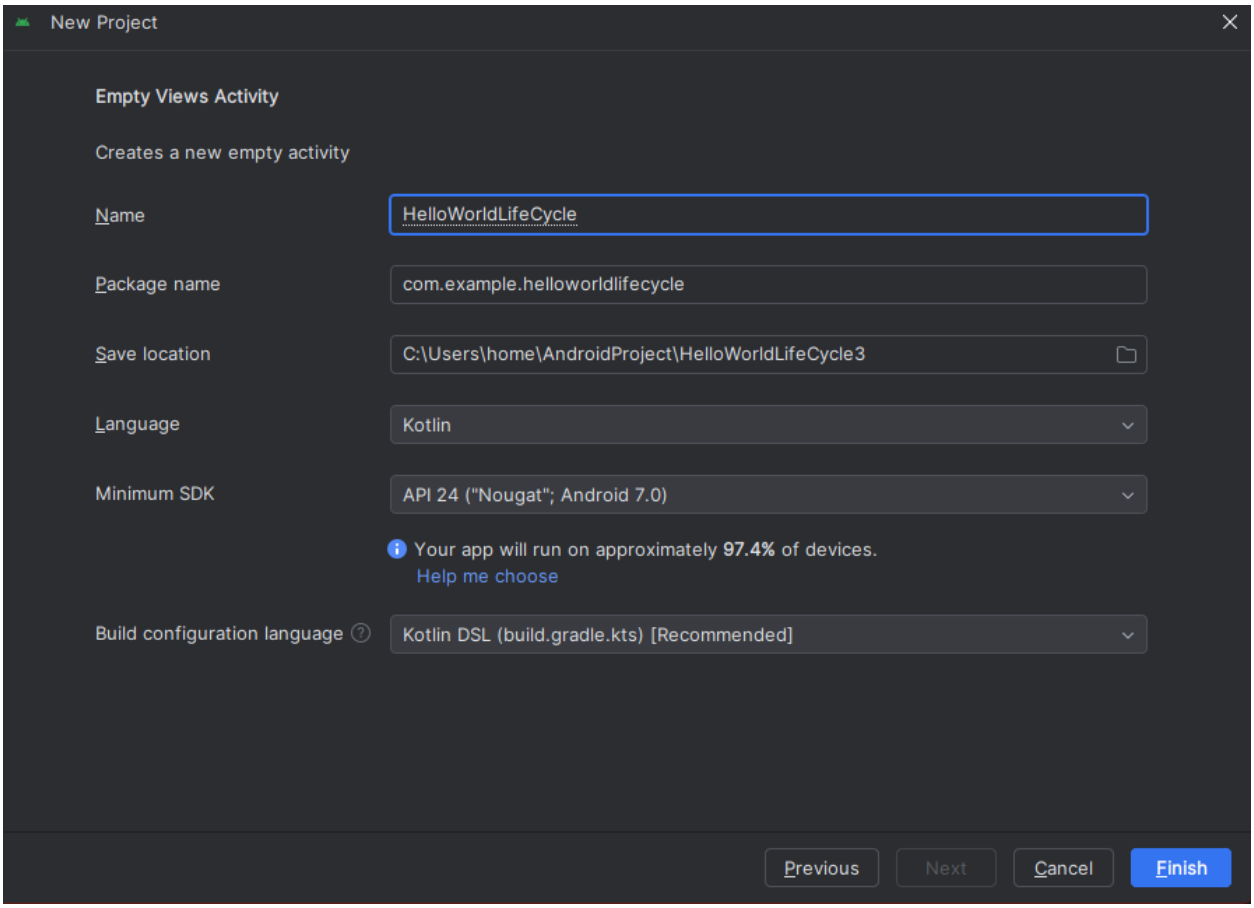
Họ và tên: Nguyễn Văn An

MSSV: 20215520

Bước 1: Tạo Project “HelloWorldLifecycle”



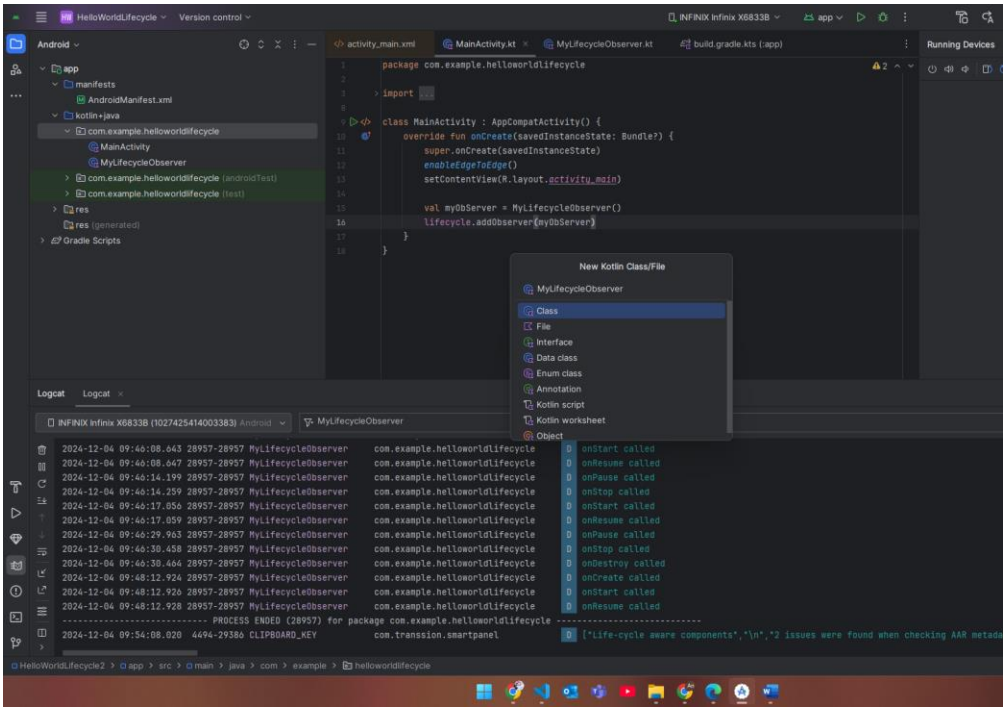
Chọn “Phone and Tablet” và chọn “Empty Views Activity”



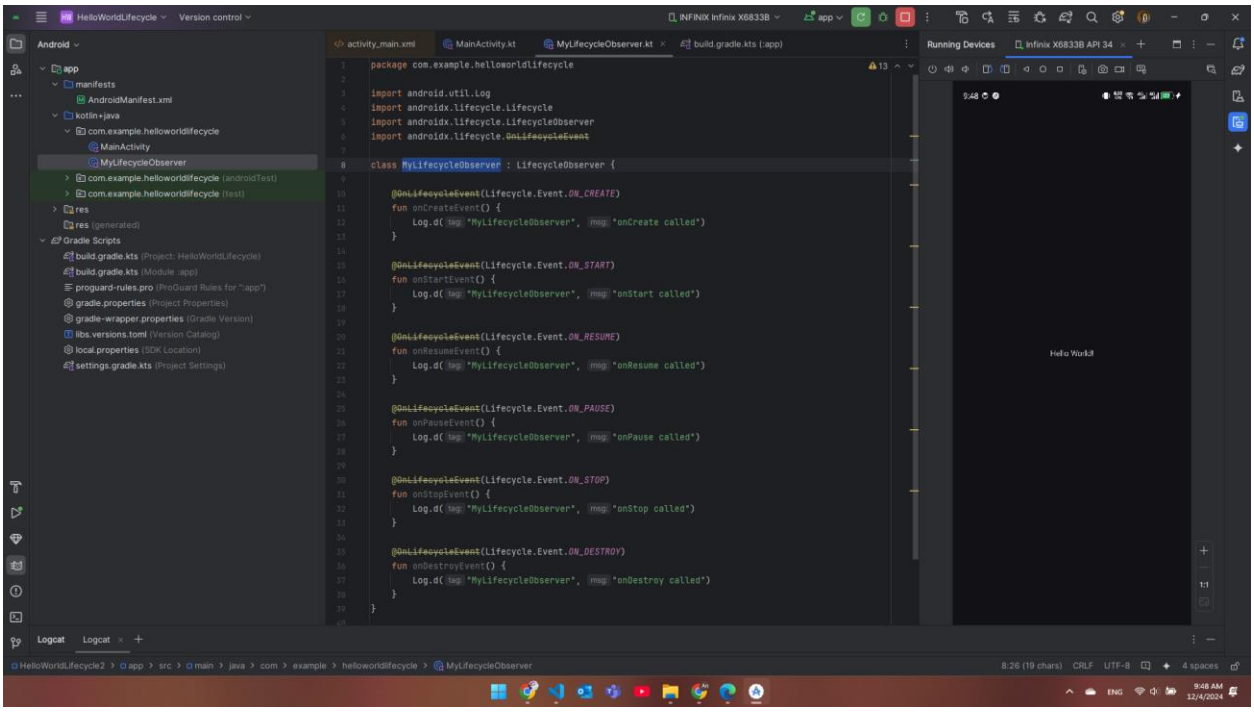
Đặt tên cho dự án là “HelloWorldLifecycle” và các phần như hình ảnh, ấn “Finish để hoàn thành tạo dự án.

Sau khi tạo chờ Android Studio build hoàn tất dự án để thực hiện bước tiếp theo.

Bước 2: Tạo đối tượng MyLifecycleObserver



Trong thư mục “app/src/main/java/com.example.helloworldlifecycle” (với tùy chọn hiển thị “Project”) hoặc “com.example.helloworldlifecycle” (với tùy chọn hiển thị “Android”), tạo class MyLifecycleObserver



2.1. Mã nguồn:

```
package com.example.helloworldlifecycle

import android.util.Log
import androidx.lifecycle.Lifecycle
import androidx.lifecycle.LifecycleObserver
import androidx.lifecycle.OnLifecycleEvent

class MyLifecycleObserver : LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
    fun onCreateEvent() {
        Log.d("MyLifecycleObserver", "onCreate called")
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    fun onStartEvent() {
        Log.d("MyLifecycleObserver", "onStart called")
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    fun onResumeEvent() {
        Log.d("MyLifecycleObserver", "onResume called")
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    fun onPauseEvent() {
        Log.d("MyLifecycleObserver", "onPause called")
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    fun onStopEvent() {
        Log.d("MyLifecycleObserver", "onStop called")
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    fun onDestroyEvent() {
        Log.d("MyLifecycleObserver", "onDestroy called")
    }
}
```

2.2. Phân tích mã nguồn:

Đoạn code trên là lớp MyLifecycleObserver thực hiện theo dõi các sự kiện trong vòng đời của một Activity hoặc Fragment bằng cách sử dụng **LifecycleObserver**.

2.2.1. class MyLifecycleObserver : LifecycleObserver

- MyLifecycleObserver là một lớp triển khai giao diện LifecycleObserver từ thư viện androidx.lifecycle.

- Mục đích của lớp này là để theo dõi các sự kiện của vòng đời ứng dụng như onCreate, onStart, onResume, onPause, onStop, và onDestroy.

2.2.2. onCreateEvent()

- Gọi khi Activity hoặc Fragment được tạo ra (**ON_CREATE**).

- Log.d được sử dụng để ghi lại thông báo vào Logcat với tag "MyLifecycleObserver" và nội dung "onCreate called".

2.2.3. onStartEvent()

- Gọi khi Activity hoặc Fragment bắt đầu hiển thị (**ON_START**).

- Logcat ghi lại "onStart called".

2.2.4. onResumeEvent()

- Gọi khi Activity hoặc Fragment bắt đầu tương tác với người dùng (**ON_RESUME**).

- Logcat ghi lại "onResume called".

2.2.5. onPauseEvent()

- Gọi khi Activity hoặc Fragment tạm dừng (**ON_PAUSE**), thường là khi ứng dụng bị che khuất hoặc mất focus.

- Logcat ghi lại "onPause called".

2.2.6. onStopEvent()

- Gọi khi Activity hoặc Fragment dừng lại hoàn toàn (**ON_STOP**).

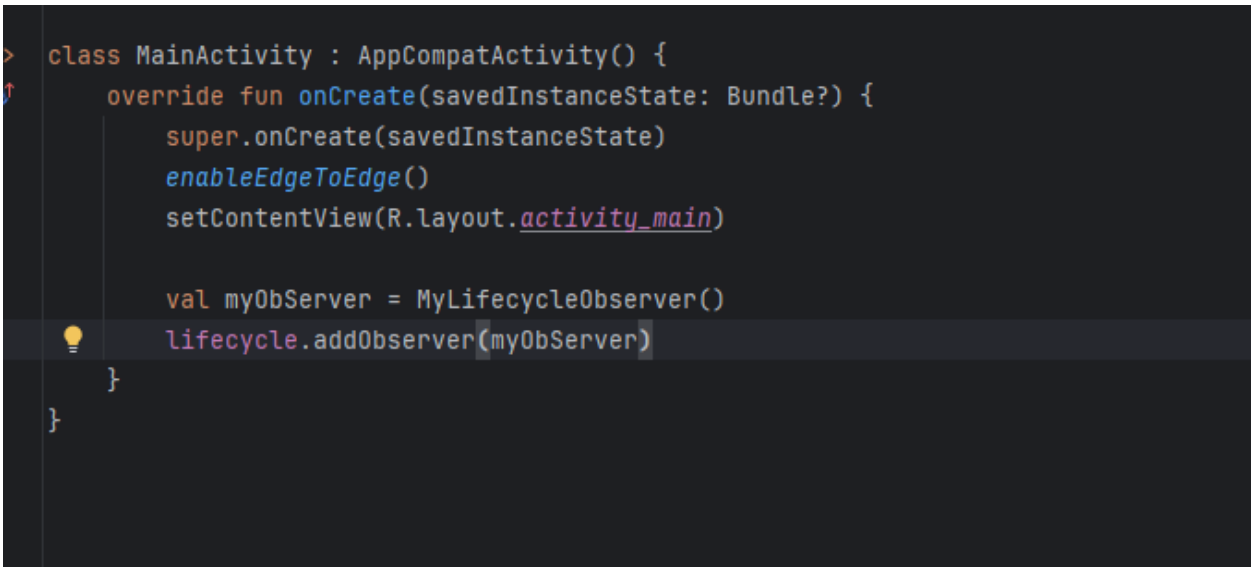
- Logcat ghi lại "onStop called".

2.2.7. onDestroyEvent()

- Gọi khi Activity hoặc Fragment bị hủy (**ON_DESTROY**).

- Logcat ghi lại "onDestroy called".

Bước 3: Đăng ký lớp MyLifecycleObserver với đối tượng lifecycle của MainActivity



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

        val myObServer = MyLifecycleObserver()
        lifecycle.addObserver(myObServer)
    }
}
```

Mã nguồn:

val myObServer = MyLifecycleObserver()
lifecycle.addObserver(myObServer)

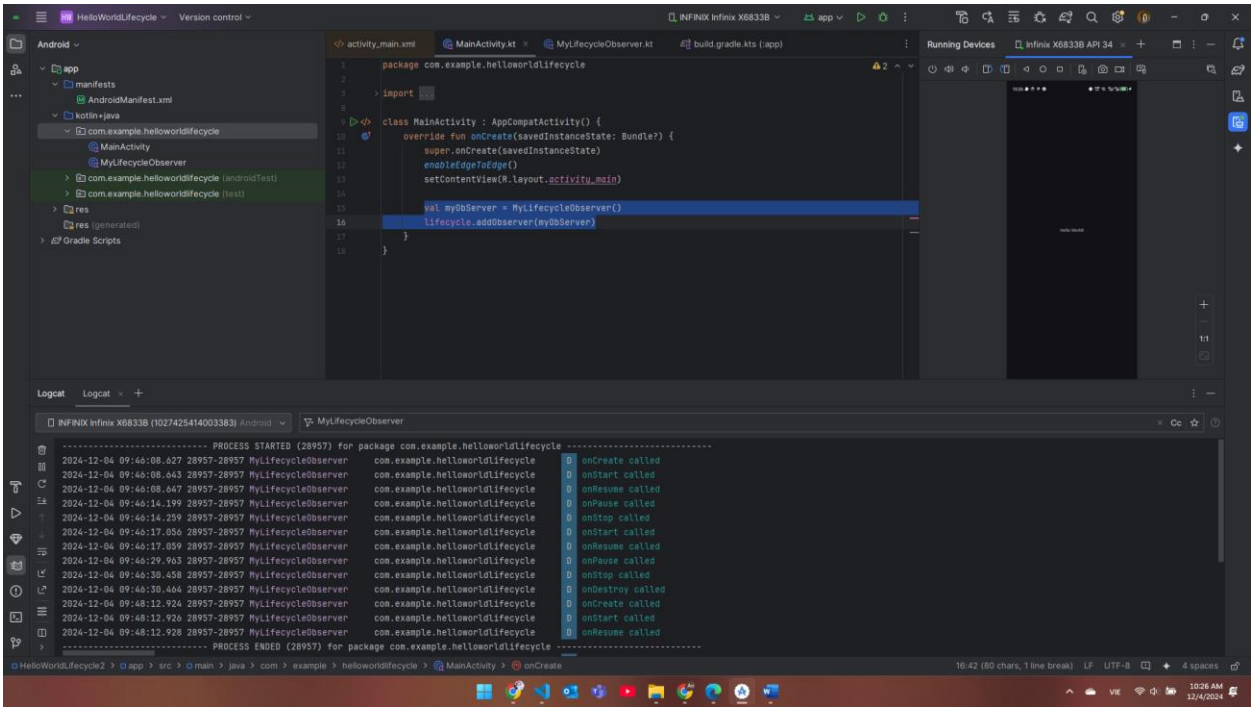
- Đối tượng myLifecycleObserver sẽ được sử dụng để theo dõi các sự kiện trong vòng đời của Activity.

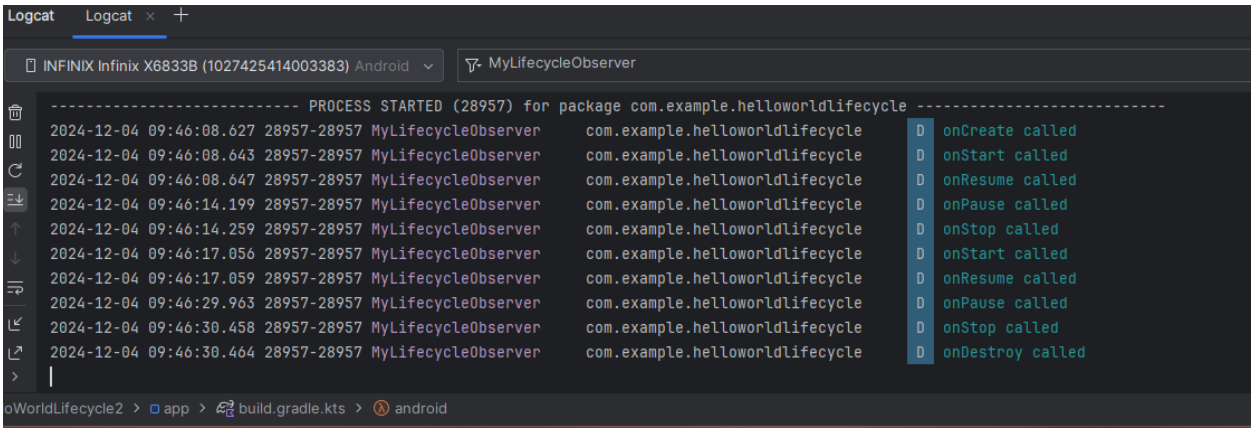
- lifecycle: Đây là một thuộc tính của lớp AppCompatActivity (hoặc ComponentActivity), đại diện cho vòng đời của Activity. Nó cung cấp khả năng theo dõi trạng thái hiện tại của Activity thông qua LifecycleOwner.

- addObserver(myLifecycleObserver): Đăng ký đối tượng myLifecycleObserver làm observer (người quan sát) vòng đời của Activity.

- Sau khi đăng ký, bất cứ khi nào Activity chuyển đổi giữa các trạng thái vòng đời như onCreate, onStart, onResume, v.v., đối tượng myLifecycleObserver sẽ được thông báo và các phương thức tương ứng trong MyLifecycleObserver sẽ được gọi.

Bước 4: Thử nghiệm ứng dụng, quan sát log





4.1. Phân tích các log:

1. onCreate called: sau khi chạy thành công ứng dụng.
2. onStart called: sau khi chạy thành công ứng dụng, app được tự động mở, người dùng truy cập vào app và app được hiển thị.
3. onResume called: sau khi người dùng truy cập vào giao diện app và có thể tương tác.
4. onPause called: sau khi thử thoát ra ngoài và không xóa đa nhiệm, một hoạt động khác đã có thể làm mờ và che giao diện.
5. onStop called: bị dừng hoàn toàn và không hiển thị trên màn hình.

6. onStart called: app được khởi động lại một lần nữa, tương tự như trên.
7. onResume called: giao diện của app được hiển thị một lần nữa, tương tự như trên.
8. onPause called: một lần nữa bị tạm dừng.
9. onStop called: một lần nữa bị dừng hoàn toàn.
10. onDestroy called: bị hủy hoàn toàn, giải phóng tài nguyên.

4.2. Luồng hoạt động

- Khởi động ứng dụng lần đầu tiên:

- onCreate → onStart → onResume
- Ứng dụng được khởi động và người dùng bắt đầu tương tác với MainActivity.

- Ứng dụng bị đưa vào nền:

- onPause → onStop
- MainActivity không còn hiển thị khi người dùng chuyển sang ứng dụng khác hoặc màn hình chính.

- Quay trở lại ứng dụng:

- onStart → onResume
- Người dùng quay lại ứng dụng, và MainActivity lại sẵn sàng tương tác.

- Ứng dụng bị đóng hoàn toàn:

- onPause → onStop → onDestroy
- MainActivity bị hủy hoàn toàn, và tài nguyên được giải phóng.

Bước 5: Kết luận và báo cáo

- Ứng dụng hoạt động đúng theo vòng đời chuẩn của Activity trong Android.
- Các trạng thái vòng đời của Activity như onCreate, onStart, onResume, onPause, onStop, và onDestroy được gọi đúng theo trình tự và được ghi lại chính xác trong log.
- Ứng dụng phản hồi chính xác khi người dùng đưa ứng dụng vào nền, quay trở lại hoặc khởi động lại.

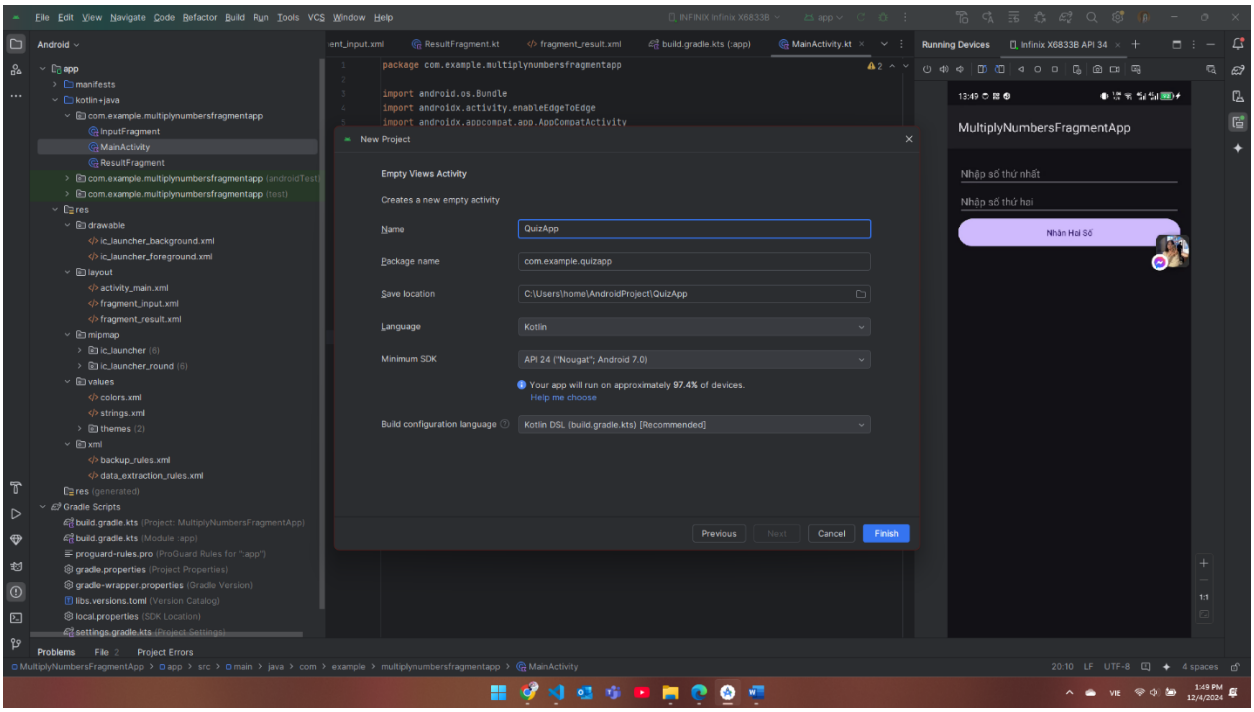
Báo cáo Tùy biến back stack với Fragment

Họ và tên: Nguyễn Văn An

MSSV: 20215520

Bước 1: Tạo dự án “QuizApp”

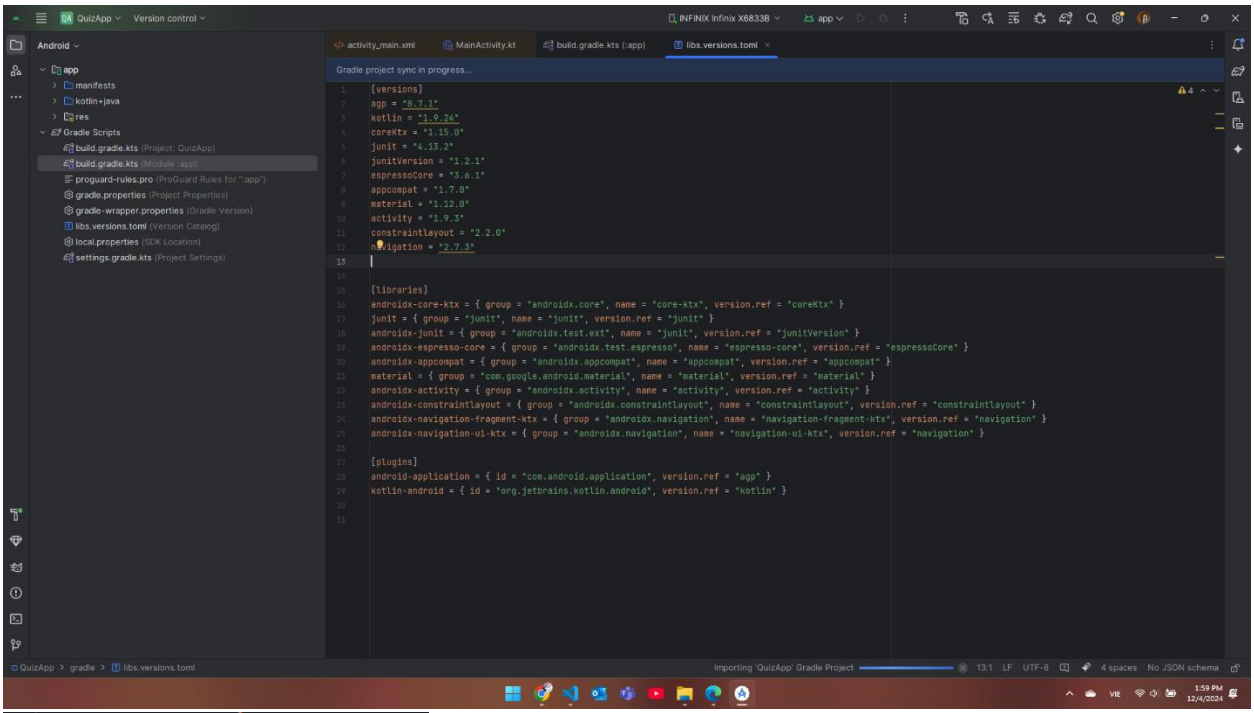
1.1.Khởi tạo dự án Android với Kotlin



Mở Android Studio và tạo một dự án mới:

- Tên dự án: QuizApp
- Ngôn ngữ: Kotlin
- Template: Empty Activity
- Minimum SDK: API 21 (Android 5.0)

1.2.Cấu hình Navigation Component

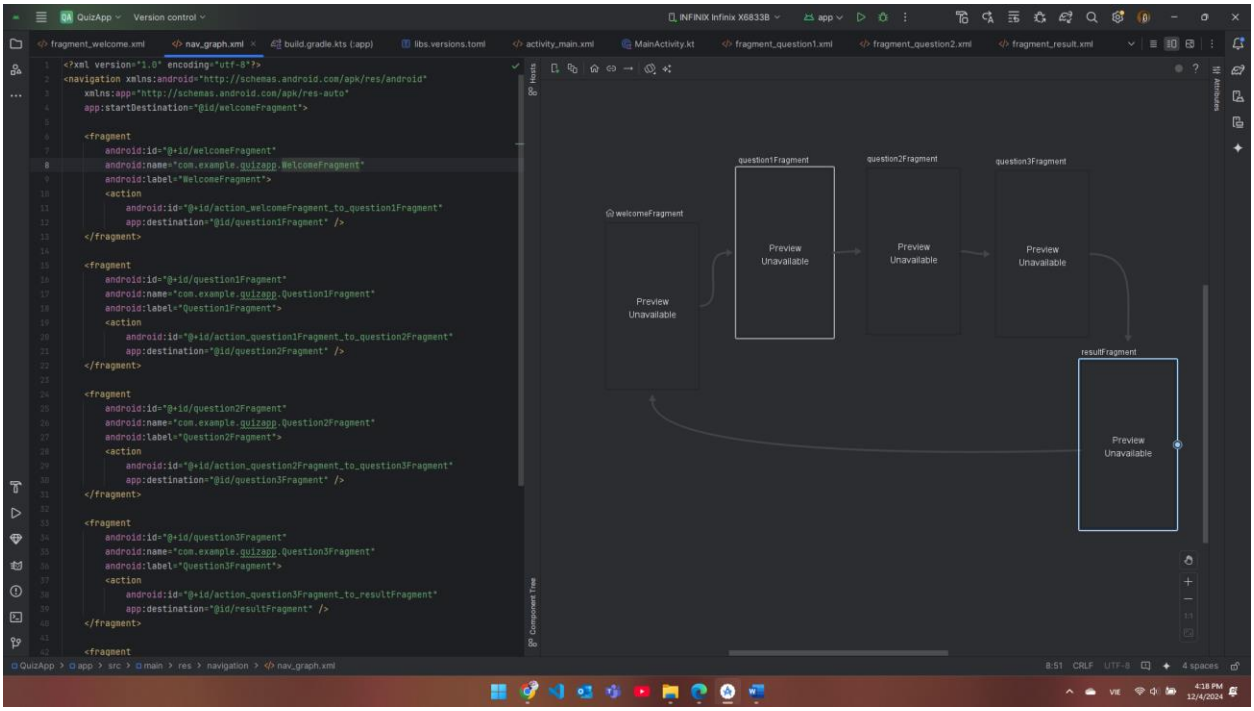


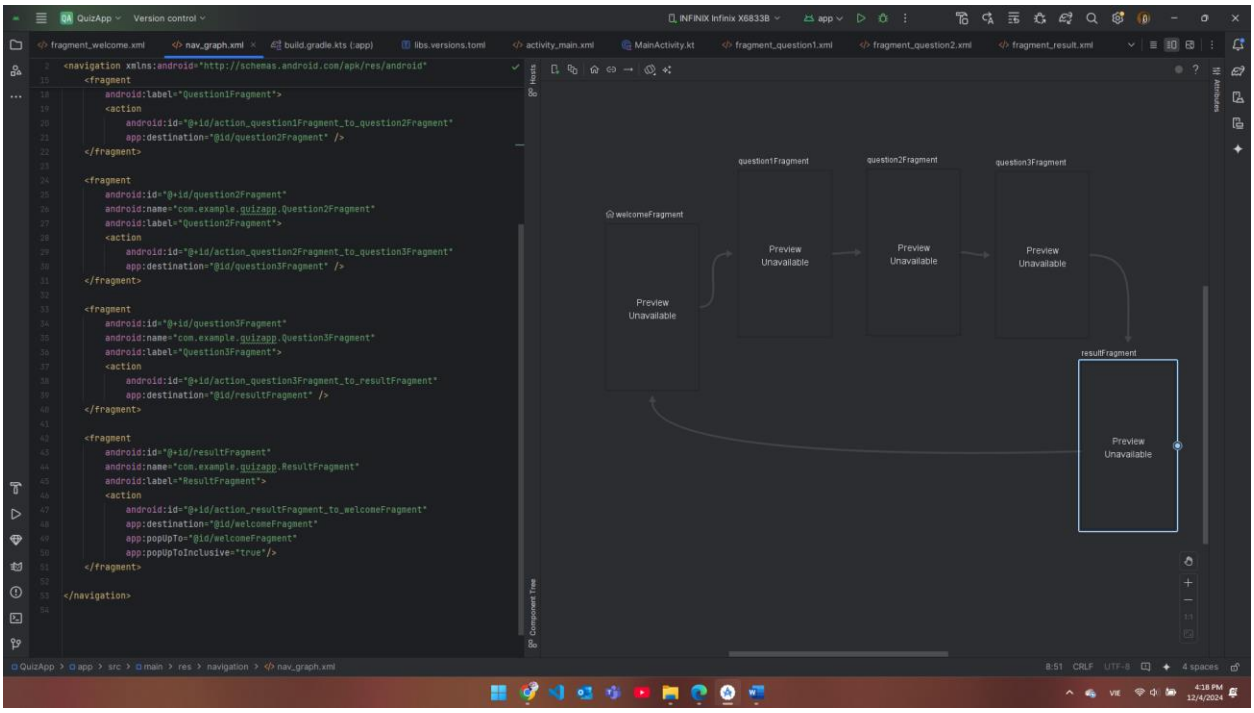

```
navigation = "2.7.3"
```

```
androidx-navigation-fragment-ktx = { group = "androidx.navigation", name = "navigation-fragment-ktx", version.ref = "navigation" }
androidx-navigation-ui-ktx = { group = "androidx.navigation", name = "navigation-ui-ktx", version.ref = "navigation" }
```

Để sử dụng Navigation Component trong dự án Android, chúng ta cần thêm các dependencies vào file build.gradle của module ứng dụng. Sau khi thêm dependencies vào file build.gradle, tiến hành Sync Gradle để Gradle tải về các thư viện cần thiết. Bạn có thể thực hiện điều này bằng cách nhấn vào nút Sync Now ở góc trên bên phải của Android Studio.

Bước 2: Tạo Navigation Graph





2.1. Tạo Navigation Graph (file nav_graph.xml)

File nav_graph.xml được sử dụng để định nghĩa các Fragment và các hành động chuyển đổi giữa chúng.

- Khởi tạo startDestination: welcomeFragment là Fragment đầu tiên khi ứng dụng bắt đầu.
- Tạo các Fragment và hành động chuyển đổi:
    - welcomeFragment: Đây là Fragment chào mừng, và từ đây người dùng có thể chuyển đến question1Fragment thông qua hành động action_welcomeFragment_to_question1Fragment.
    - question1Fragment, question2Fragment, và question3Fragment: Mỗi Fragment này đều có hành động chuyển tiếp đến Fragment kế tiếp.
    - resultFragment: Sau khi người dùng hoàn thành các câu hỏi, ứng dụng sẽ chuyển đến resultFragment để hiển thị kết quả. Từ đây, có thể quay lại welcomeFragment thông qua hành động action_resultFragment_to_welcomeFragment.
- Cấu hình Back Stack:
    - popUpTo: Để khi người dùng quay lại từ resultFragment, nó sẽ đưa người dùng trở lại welcomeFragment và xóa tất cả các Fragment ở phía trên của nó trong back stack. Cụ thể, app:popUpTo="@id/welcomeFragment" và app:popUpToInclusive="true" đảm bảo rằng welcomeFragment sẽ là Fragment duy nhất còn lại trong back stack khi người dùng quay lại.

2.2. Mô tả các Fragment:

- WelcomeFragment: Là màn hình chào mừng, nơi người dùng bắt đầu hành trình.
- Question1Fragment, Question2Fragment, Question3Fragment: Mỗi Fragment chứa một câu hỏi trắc nghiệm. Người dùng sẽ trả lời các câu hỏi và chuyển tiếp từ câu hỏi này sang câu hỏi tiếp theo.
- ResultFragment: Sau khi người dùng hoàn thành các câu hỏi, ứng dụng sẽ hiển thị kết quả của bài thi trong Fragment này.

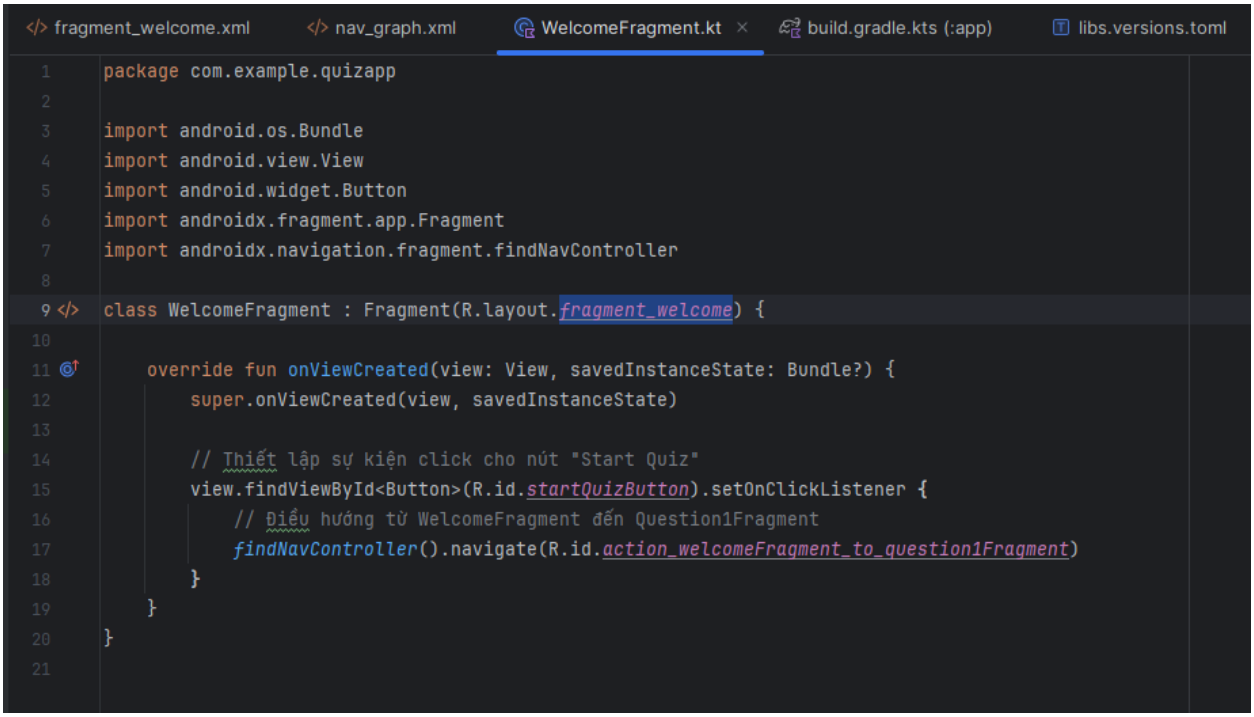
2.3. Quản lý dữ liệu giữa các Fragment:

- Để truyền dữ liệu từ các câu hỏi đến resultFragment, có thể sử dụng Bundle hoặc ViewModel để lưu trữ và truyền thông tin trả lời giữa các Fragment

Bước 3: Cấu hình các Fragment

3.1. Màn hình mở đầu

3.1.1. WelcomeFragment Class



Trong ứng dụng Quiz, WelcomeFragment là Fragment đầu tiên mà người dùng sẽ nhìn thấy khi mở ứng dụng. Đây là nơi bắt đầu của bài trắc nghiệm. Dưới đây là mô tả chi tiết về cách hoạt động của WelcomeFragment:

1. Mô tả chung:

- Mục đích: WelcomeFragment hiển thị màn hình chào mừng với một nút "Start Quiz", nơi người dùng có thể bắt đầu bài kiểm tra.
- Chức năng chính: Khi người dùng nhấn nút "Start Quiz", ứng dụng sẽ điều hướng đến Question1Fragment để bắt đầu chuỗi câu hỏi.

2. Mã nguồn và sự kiện:

Trong WelcomeFragment, ta sử dụng onViewCreated để thiết lập sự kiện click cho nút "Start Quiz":

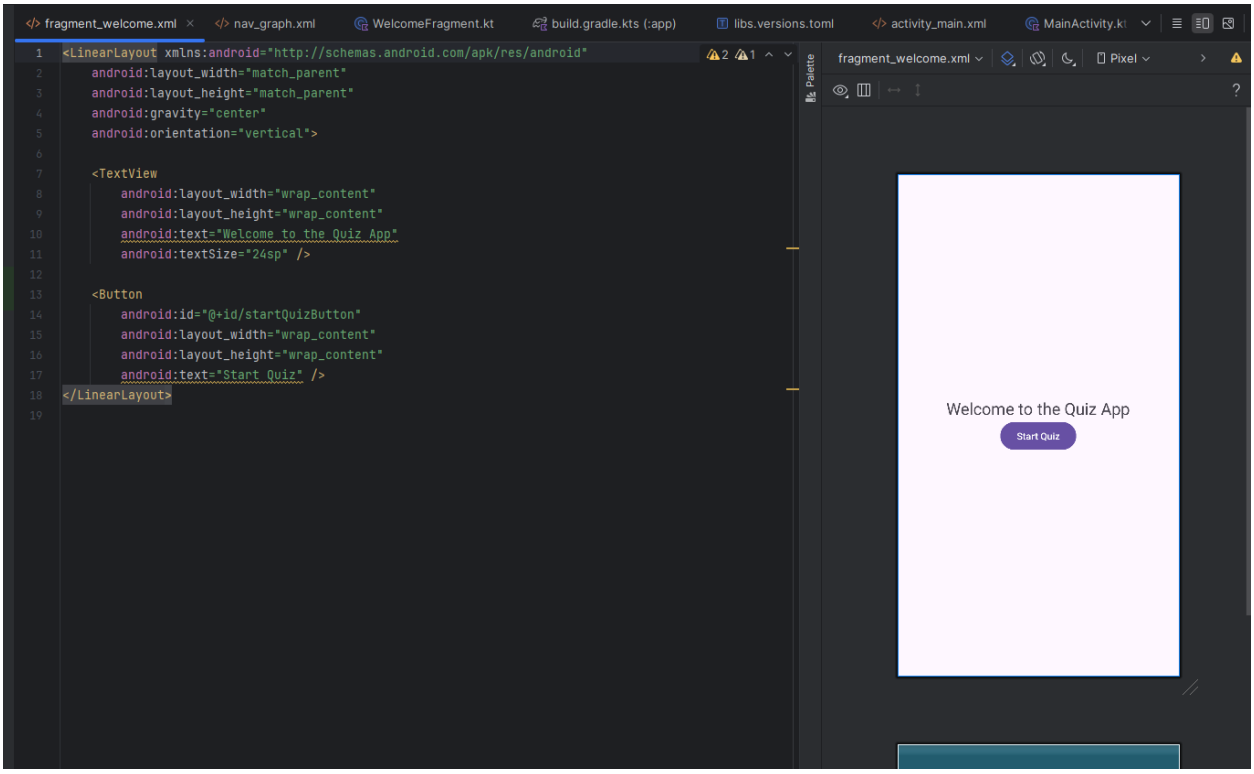
```
view.findViewById<Button>(R.id.startQuizButton).setOnClickListener {
    // Điều hướng từ WelcomeFragment đến Question1Fragment

findNavController().navigate(R.id.action_welcomeFragment_to_question1Fragment)
}
```

- Sự kiện click: Khi người dùng nhấn nút "Start Quiz", sự kiện click được kích hoạt.
- Điều hướng: Sử dụng findNavController().navigate() để chuyển từ WelcomeFragment đến Question1Fragment. Hành động này được định nghĩa trong nav_graph.xml với ID action_welcomeFragment_to_question1Fragment.

3. Kết luận:

- Điều hướng dễ dàng: Việc sử dụng Navigation Component giúp đơn giản hóa việc chuyển đổi giữa các Fragment mà không cần phải quản lý FragmentTransaction thủ công.
- Giao diện người dùng: WelcomeFragment cung cấp một giao diện người dùng đơn giản và dễ sử dụng với nút bắt đầu, giúp người dùng dễ dàng bắt đầu bài trắc nghiệm.

3.1.2. fragment_welcome xml



- Layout: Sử dụng LinearLayout với hướng dọc (vertical) để căn chỉnh các phần tử trong giao diện.

- Định vị các thành phần: TextView và Button được căn giữa theo chiều dọc và ngang nhờ thuộc tính android:gravity="center".

- Giao diện đơn giản: Layout này có giao diện đơn giản với một dòng văn bản chào mừng và một nút bắt đầu để người dùng khởi động bài kiểm tra.

- Tính năng: Người dùng có thể bắt đầu bài trắc nghiệm khi nhấn vào nút "Start Quiz", chuyển tiếp đến các câu hỏi trong ứng dụng.

3.2. Màn hình các câu hỏi

3.2.1. QuestionFragment Class

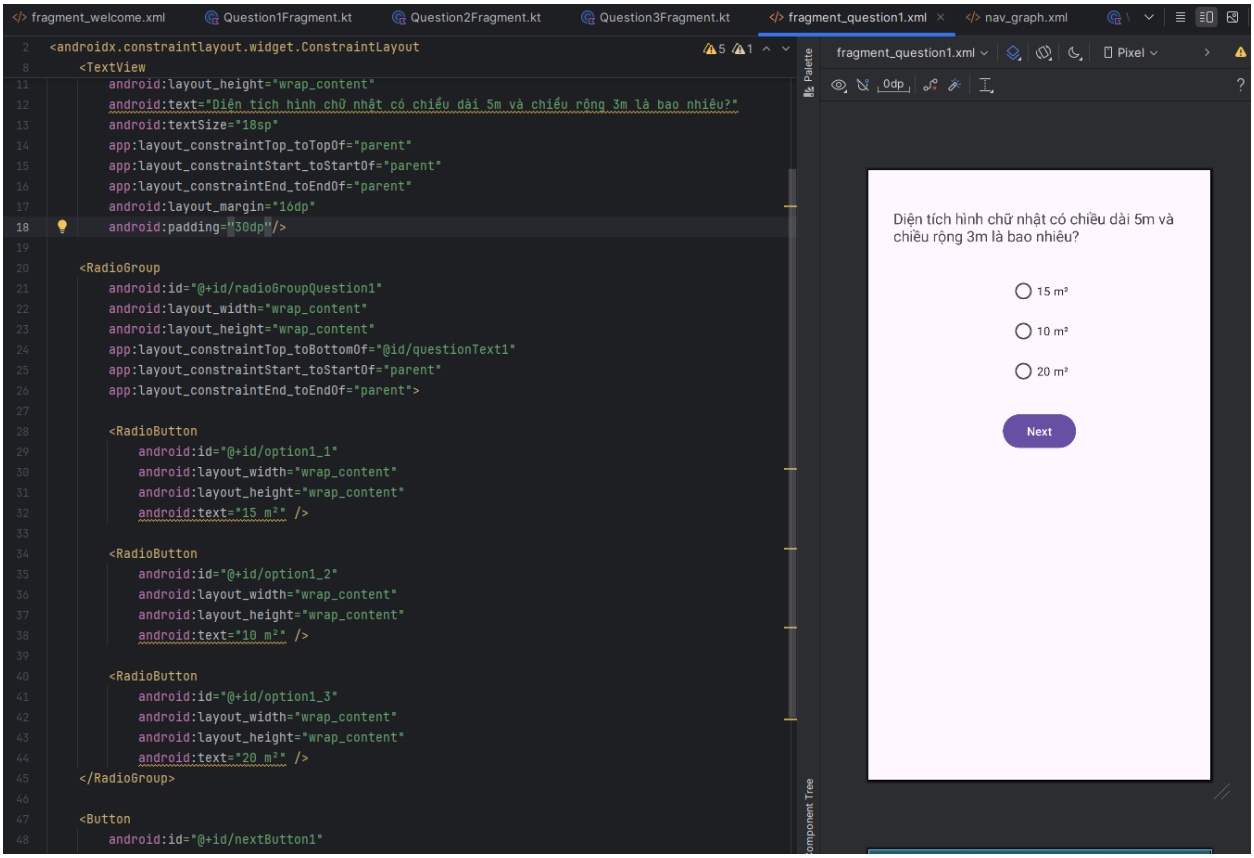
```
class Question1Fragment : Fragment(R.layout.fragment_question1) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // Lắng nghe sự kiện click của nút "Next"
        view.findViewById<Button>(R.id.nextButton1).setOnClickListener {
            // Lấy ID của RadioButton được chọn
            val selectedAnswer = view.findViewById<RadioGroup>(R.id.radioGroupQuestion1)
                .checkedRadioButtonId

            if (selectedAnswer != -1) {
                // Lấy text của câu trả lời
                val answer = view.findViewById<RadioButton>(selectedAnswer).text.toString()

                // Đưa câu trả lời vào Bundle để truyền sang Fragment tiếp theo
                val bundle = bundleOf( ...pairs: "answer1" to answer)

                // Điều hướng sang Question2Fragment và truyền Bundle
                findNavController().navigate(
                    R.id.action_question1Fragment_to_question2Fragment,
                    bundle
                )
            }
        }
    }
}
```

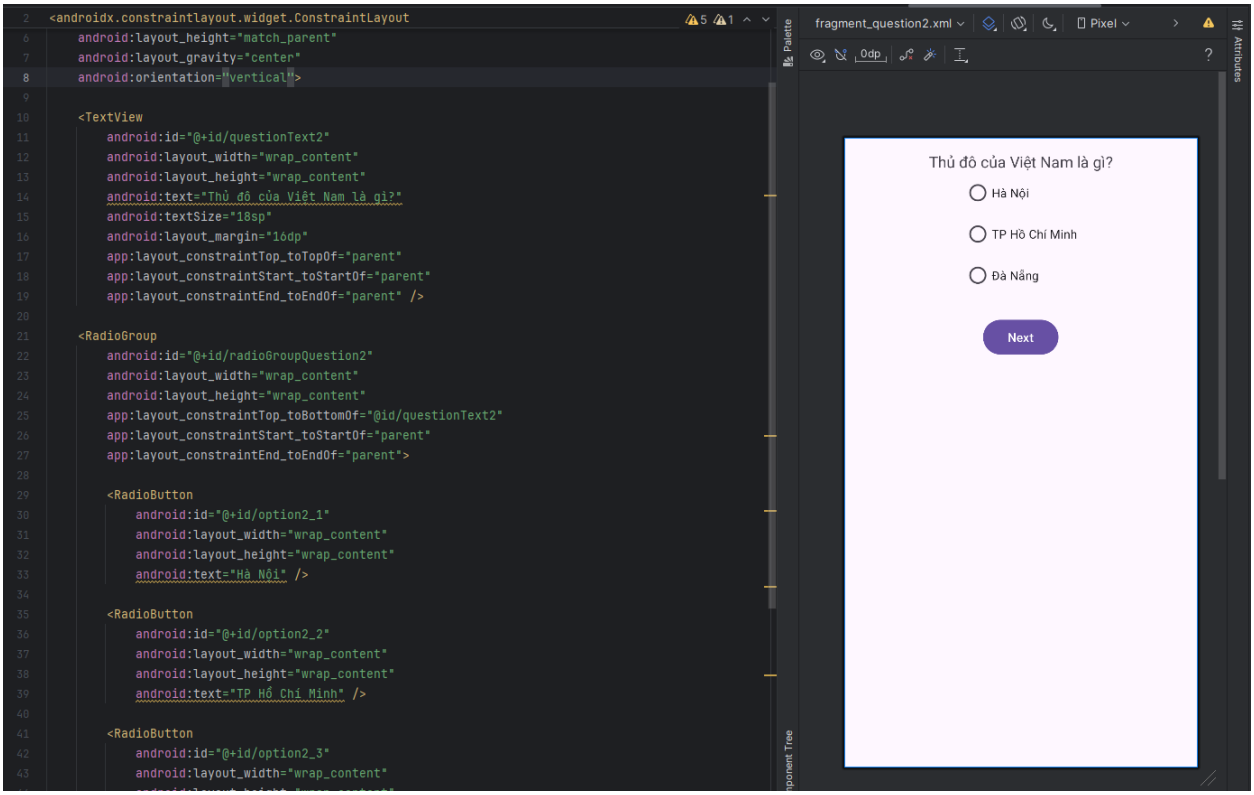
```kotlin
class Question2Fragment : Fragment(R.layout.fragment_question2) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // Lấy câu trả lời của Question 1 từ arguments
        val answer1 = arguments?.getString( key: "answer1")

        // Xử lý sự kiện khi nhấn nút "Next"
        view.findViewById<Button>(R.id.nextButton2).setOnClickListener {
            // Kiểm tra xem người dùng đã chọn câu trả lời chưa
            val selectedAnswer = view.findViewById<RadioGroup>(R.id.radioGroupQuestion2)
                .checkedRadioButtonId

            // Nếu chưa chọn đáp án, hiển thị thông báo
            if (selectedAnswer == -1) {
                Toast.makeText(context, text: "Vui lòng chọn một câu trả lời!", Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }

            // Lấy câu trả lời đã chọn
            val answer2 = view.findViewById<RadioButton>(selectedAnswer).text.toString()

            // Tạo bundle chứa câu trả lời từ cả hai câu hỏi
            val bundle = bundleOf( ...pairs: "answer1" to answer1, "answer2" to answer2)

            // Chuyển sang Question3Fragment và truyền dữ liệu
            findNavController().navigate(R.id.action_question2Fragment_to_question3Fragment, bundle)
        }
    }
}
```

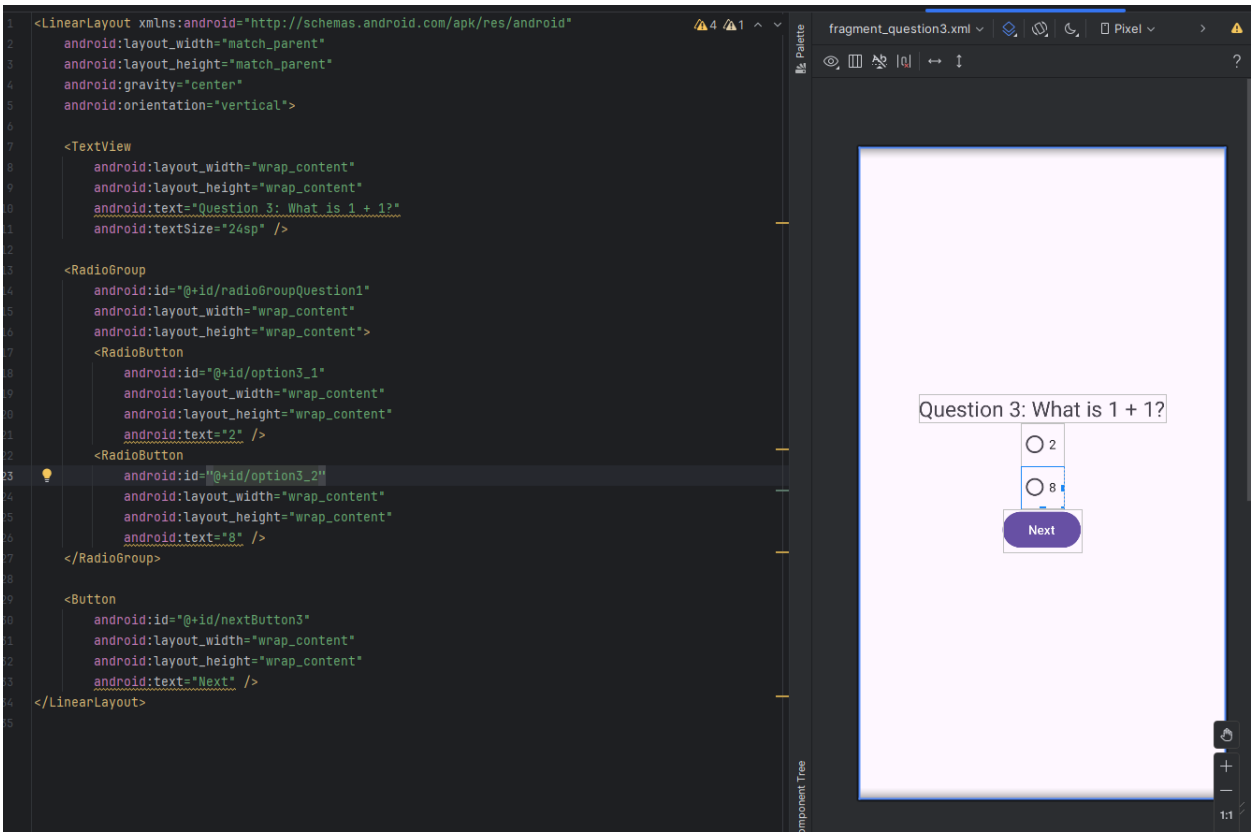
```kotlin
class Question3Fragment : Fragment(R.layout.fragment_question2) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // Lấy câu trả lời của Question 1 từ arguments
        val answer1 = arguments?.getString( key: "answer1")
        val answer2 = arguments?.getString( key: "answer2")

        // Xử lý sự kiện khi nhấn nút "Next"
        view.findViewById<Button>(R.id.nextButton3).setOnClickListener {
            // Kiểm tra xem người dùng đã chọn câu trả lời chưa
            val selectedAnswer = view.findViewById<RadioGroup>(R.id.radioGroupQuestion2)
                .checkedRadioButtonId

            // Nếu chưa chọn đáp án, hiển thị thông báo
            if (selectedAnswer == -1) {
                Toast.makeText(context, text: "Vui lòng chọn một câu trả lời!", Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }

            // Lấy câu trả lời đã chọn
            val answer3 = view.findViewById<RadioButton>(selectedAnswer).text.toString()

            // Tạo bundle chứa câu trả lời từ cả hai câu hỏi
            val bundle = bundleOf( ...pairs: "answer1" to answer1, "answer2" to answer2, "answer3" to answer3)

            // Chuyển sang Question3Fragment và truyền dữ liệu
            findNavController().navigate(R.id.action_question3Fragment_to_resultFragment, bundle)
        }
    }
}
```

- **Chức năng:** Question1Fragment hiển thị câu hỏi đầu tiên và cung cấp các lựa chọn dưới dạng RadioButton. Người dùng chọn một câu trả lời và nhấn nút "Next" để chuyển sang câu hỏi tiếp theo.
- **Điều hướng:** Khi người dùng nhấn nút "Next", câu trả lời của họ sẽ được lấy và truyền sang Question2Fragment.
- **Chức năng chính:** Question1Fragment giúp người dùng chọn một câu trả lời cho câu hỏi đầu tiên và chuyển câu trả lời này sang Fragment tiếp theo.
- **Dễ dàng chuyển tiếp dữ liệu:** Sử dụng NavController để điều hướng giữa các Fragment và Bundle để truyền dữ liệu giữa chúng.

3.2.2. fragment_question xml







- Layout: Sử dụng ConstraintLayout để dễ dàng căn chỉnh các phần tử theo các ràng buộc (constraints), giúp tạo giao diện linh hoạt và dễ dàng thích ứng với nhiều kích thước màn hình.

- Các thành phần chính:

  - Một TextView hiển thị câu hỏi.
  - Một RadioGroup chứa các lựa chọn câu trả lời (3 RadioButton).
  - Một Button có nhãn "Next" để người dùng chuyển sang câu hỏi tiếp theo.

- Chức năng chính: Giao diện của Question1Fragment cung cấp cho người dùng câu hỏi đầu tiên cùng với các lựa chọn để trả lời. Sau khi người dùng chọn một câu trả lời, họ có thể nhấn nút "Next" để chuyển sang câu hỏi tiếp theo.

- Điều chỉnh linh hoạt: Sử dụng ConstraintLayout để đảm bảo giao diện này có thể thích ứng với nhiều kích thước màn hình khác nhau mà không gặp vấn đề về bố cục.

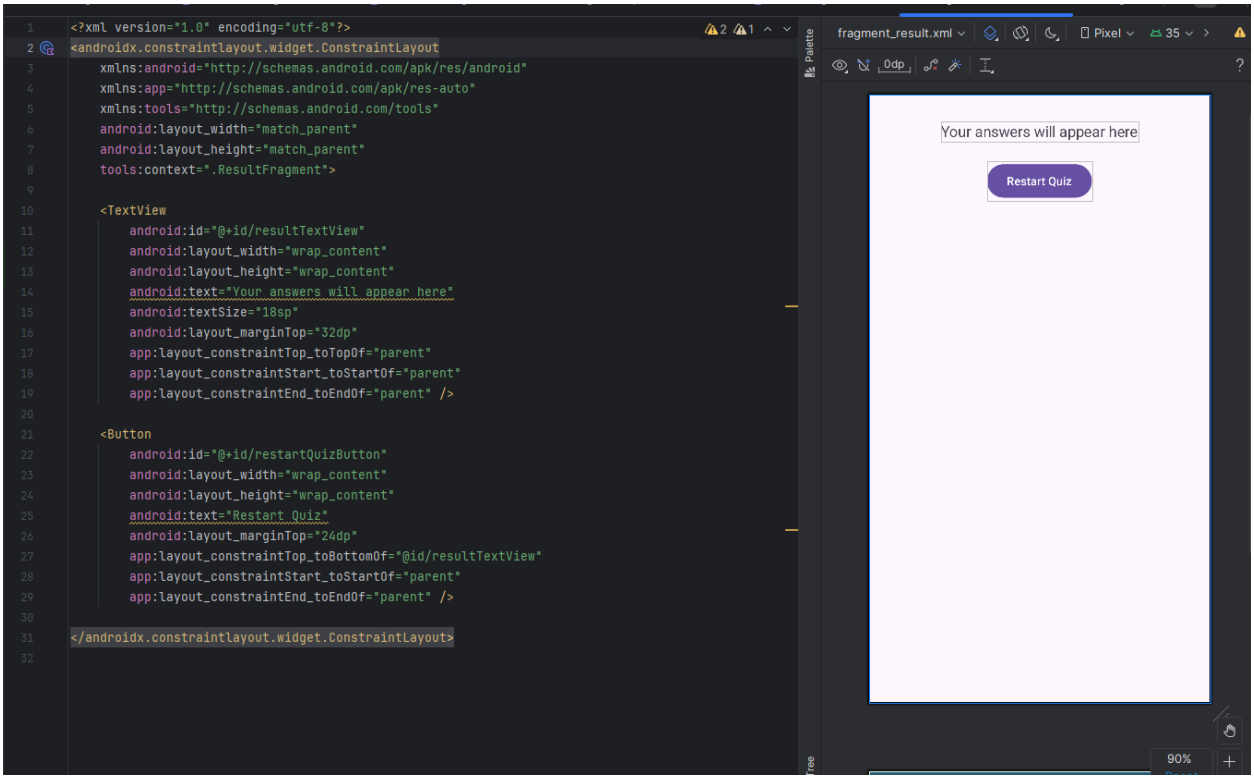
3.3. Màn hình kết thúc

3.3.1. ResultFragment Class

```
class ResultFragment : Fragment(R.layout.fragment_result) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // Lấy các câu trả lời từ các Fragment trước đó
        val answer1 = arguments?.getString( key: "answer1")
        val answer2 = arguments?.getString( key: "answer2")
        val answer3 = arguments?.getString( key: "answer3")

        // Hiển thị kết quả
        val resultTextView = view.findViewById<TextView>(R.id.resultTextView)
        resultTextView.text = """
            Your answers:
            1. $answer1
            2. $answer2
            3. $answer3
        """.trimIndent()

        // Nút quay lại WelcomeFragment
        view.findViewById<Button>(R.id.restartQuizButton).setOnClickListener {
            findNavController().navigate(R.id.action_resultFragment_to_welcomeFragment)
        }
    }
}
```

- Layout: Sử dụng fragment_result.xml (chưa cung cấp) để hiển thị các câu trả lời đã chọn và cung cấp nút "Restart Quiz" để quay lại WelcomeFragment.
- Chức năng chính:
    - Hiển thị kết quả của người dùng.
    - Cho phép người dùng bắt đầu lại bài quiz.
- Chức năng: ResultFragment cung cấp cho người dùng cái nhìn tổng quan về các câu trả lời mà họ đã chọn trong quiz. Nó cũng cho phép người dùng quay lại màn hình chào mừng để bắt đầu lại quiz.
- Điều hướng dễ dàng: Sử dụng Navigation Component để điều hướng giữa các fragment, giúp quản lý luồng người dùng một cách mượt mà và dễ dàng.

3.3.2. fragment_result xml




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

    <TextView
        android:id="@+id/resultTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Your answers will appear here"
        android:textSize="18sp"
        android:layout_marginTop="32dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

    <Button
        android:id="@+id/restartQuizButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Restart Quiz"
        android:layout_marginTop="24dp"
        app:layout_constraintTop_toBottomOf="@id/resultTextView"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

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

- Layout: Sử dụng ConstraintLayout để bố trí các phần tử UI.
- Chức năng:
    - Hiển thị các câu trả lời của người dùng.
    - Cung cấp nút "Restart Quiz" để người dùng quay lại WelcomeFragment và bắt đầu lại quiz.

Bài thực hành Lab 8.1. Bản cập nhật tháng 11 năm 2024

Mã khởi đầu: https://github.com/google-developer-training/android-basics-kotlin-unscramble-app/tree/starter

Bước 1. Tạo project với tên App unscramble

Cập nhật API 35

Trong file app/build.gradle.kts

```
android {
    namespace = "com.example.unscramble"
    compileSdk = 35

defaultConfig {
    applicationId = "com.example.unscramble"
    minSdk = 26
    targetSdk = 35
```

Ấn Sync để đồng bộ thư viện.

Bước 2. Cài đặt thư viện SafeArgs và Fragment

Trong file build.gradle.kts của Project

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
    id("androidx.navigation.safeargs") version "2.8.3" apply false
}
```

Trong file app/build.gradle.kts: bổ sung các phần bôi vàng

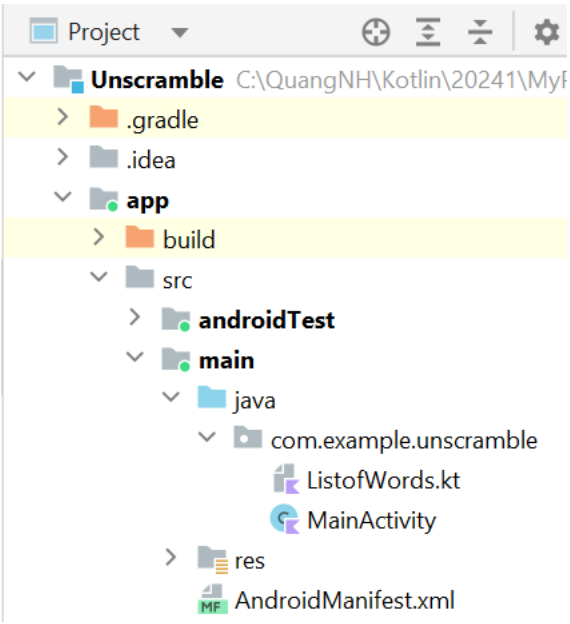
```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("androidx.navigation.safeargs.kotlin")
}

android {

        buildFeatures {
            dataBinding = true
        }

}
```

dependencies **{**

    implementation("androidx.navigation:navigation-fragment-ktx:2.8.3")
    implementation("androidx.navigation:navigation-ui-ktx:2.8.3")
}

Bước 3. Copy file ListofWords.kt vào project

https://github.com/google-developer-training/android-basics-kotlin-unscramble-app/blob/starter/app/src/main/java/com/example/android/unscramble/ui/game/ListofWords.kt

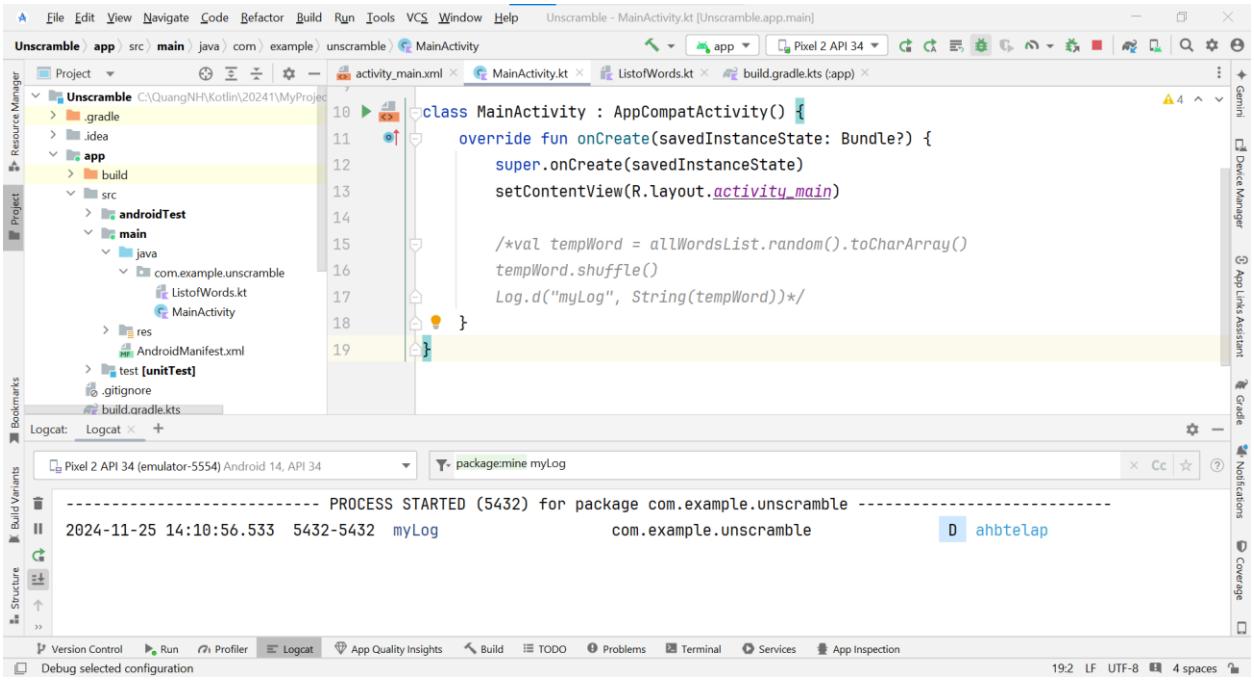


Cập nhật lại tên gói:

package com.example.unscramble

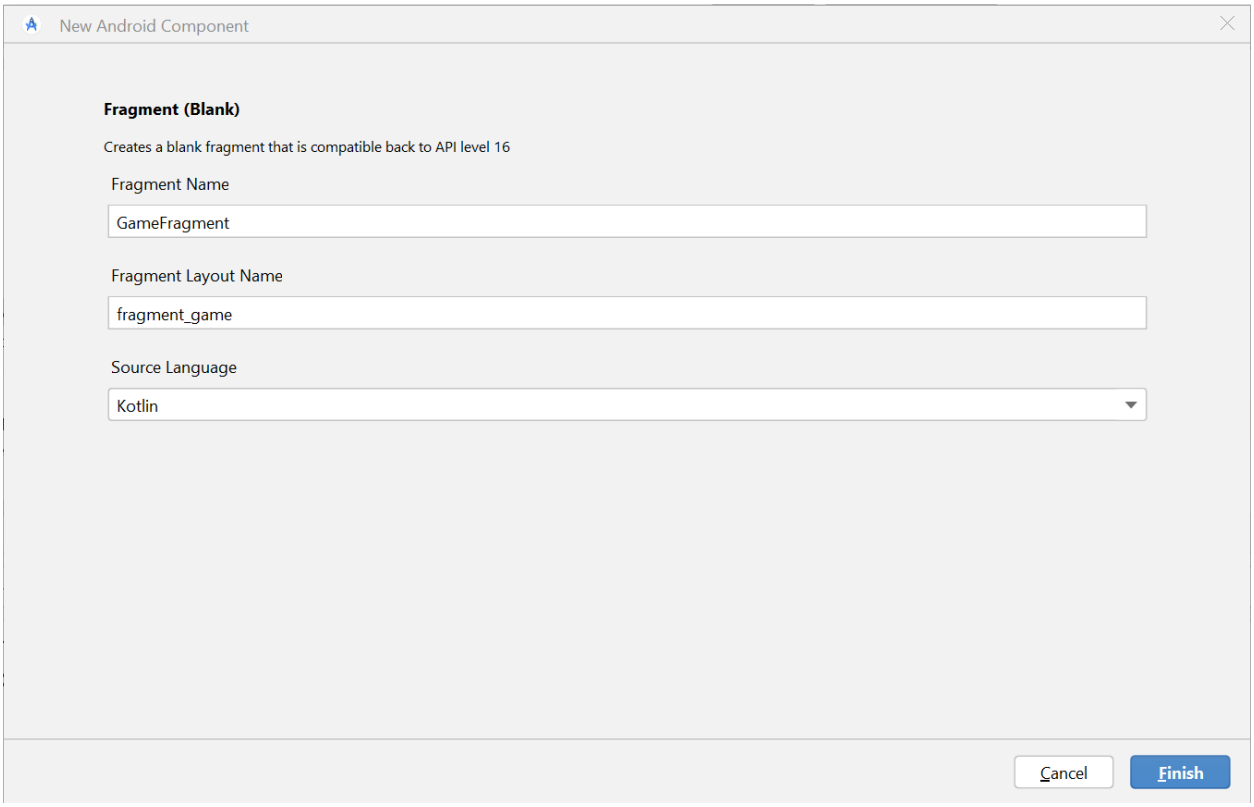
Thử nghiệm đối tượng *allWordsList khai báo trong file ListofWords.kt*

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

        /*val tempWord = allWordsList.random().toCharArray()
        tempWord.shuffle()
        Log.d("myLog", String(tempWord))*/
    }
}
```



Bước 4. Bổ sung GameFragment

Chọn File => New => Fragment => Fragment (Blank)



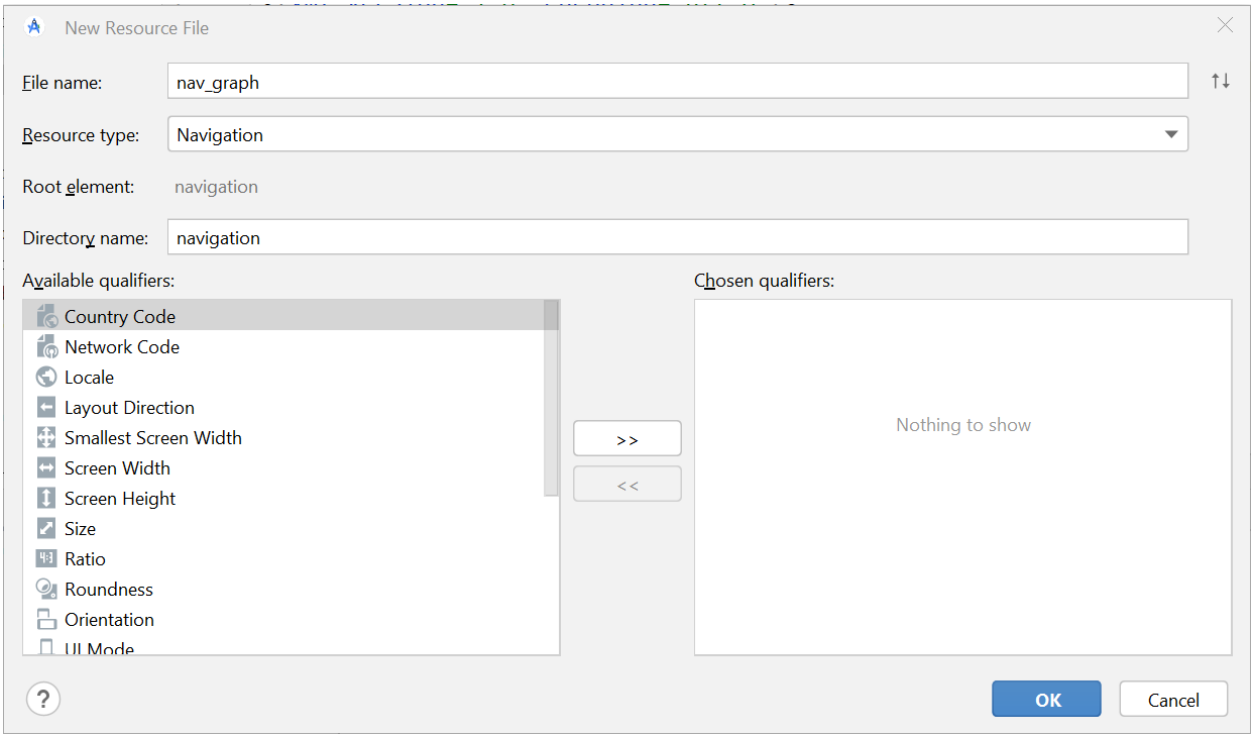
```
package com.example.unscramble

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class GameFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_game, container, false)
    }
}
```

Bước 5. Thiết lập cơ chế View Binding cho GameFragment

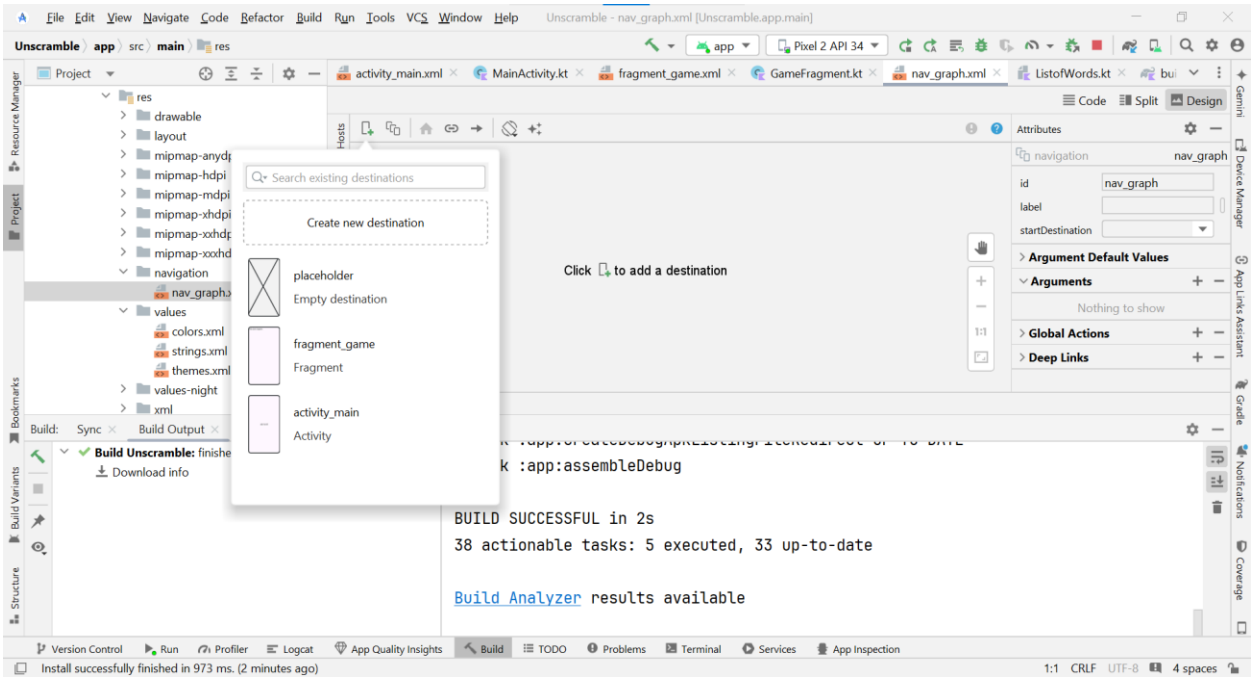
```
package com.example.unscramble

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.example.unscramble.databinding.FragmentGameBinding

class GameFragment : Fragment() {
    private lateinit var binding: FragmentGameBinding
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        //return inflater.inflate(R.layout.fragment_game,
container, false)
        binding = FragmentGameBinding.inflate(inflater,
container, false)
        return binding.root
    }
}
```

Bước 6. Tạo file Navigation Graph

Ấn chuột phải vào thư mục res, chọn New => Android Resource File



Bước 7. Thêm GameFragment vào nav_graph.xml



Bước 8. Tích hợp Navigation Graph vào MainActivity

Sửa lại file activity_main.xml như sau:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <fragment
        android:id="@+id/nav_host"

android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />
</FrameLayout>
```

Chạy chương trình:

Bước 9. Tạo file dimens.xml trong thư mục values

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="default_padding">16dp</dimen>
    <dimen name="default_margin">32dp</dimen>
</resources>
```

Bước 10. Bổ sung các xâu ký tự trong file strings.xml

```
<resources>
    <string name="app_name">Unscramble</string>
    <!-- TODO: Remove or change this placeholder text -->
    <string name="hello_blank_fragment">Hello blank
fragment</string>
    <string name="instructions">Unscramble the word using all
the letters.</string>
    <string name="skip">Skip</string>
    <string name="submit">Submit</string>
    <string name="score">Score: %d</string>
    <string name="word_count">%d of %d words</string>
    <string name="enter_your_word">Enter your word</string>
    <string name="congratulations">Congratulations!</string>
    <string name="you_scored">"You scored: %d"</string>
    <string name="exit">Exit</string>
    <string name="play_again">Play Again</string>
    <string name="try_again">Try again!</string>
</resources>
```

Bước 11. Bổ sung file styles.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Widget.Unscramble.TextInputLayout.OutlinedBox"
parent="Widget.MaterialComponents.TextInputLayout.OutlinedBox">
        <item name="boxStrokeErrorColor">@color/red_700</item>
        <item name="errorIconTint">@color/red_700</item>
        <item name="errorTextColor">@color/red_700</item>
        <item name="errorIconDrawable">@drawable/ic_error</item>
        <item
name="helperTextTextAppearance">@style/TextAppearance.MaterialCo
mponents.Subtitle1</item>
    </style>
</resources>
```

Bước 12. Bổ sung file colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="indigo_200">#FF9FA8DA</color>
    <color name="indigo_500">#FF3F51B5</color>
    <color name="indigo_800">#FF283593</color>
    <color name="light_blue_200">#FF81D4FA</color>
    <color name="light_blue_700">#FF0288D1</color>
    <color name="red_700">#FFD32F2F</color>
    <color name="red_400">#FFEF5350</color>
</resources>
```

Bước 13. Tạo file ic_error.xml trong thư mục drawable

```
<vector android:height="24dp" android:tint="#C62828"
    android:viewportHeight="24" android:viewportWidth="24"
    android:width="24dp"
xmlns:android="http://schemas.android.com/apk/res/android">
    <path android:fillColor="@android:color/white"
android:pathData="M11,15h2v2h-2zM11,7h2v6h-2zM11.99,2C6.47,2
2,6.48 2,12s4.47,10 9.99,10C17.52,22 22,17.52 22,12S17.52,2
11.99,2zM12,20c-4.42,0 -8,-3.58 -8,-8s3.58,-8 8,-8 8,3.58 8,8 -
3.58,8 -8,8z"/>
</vector>
```

Bước 14. Tạo giao diện cho GameFragment

```
<?xml version="1.0" encoding="utf-8"?>

<ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="@dimen/default_padding"
        tools:context=".ui.game.GameFragment">
```

```
        <Button
            android:id="@+id/skip"
            style="?attr/materialButtonOutlinedStyle"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginStart="@dimen/default_padding"
            android:layout_marginEnd="@dimen/default_padding"
            android:text="@string/skip"

app:layout_constraintBaseline_toBaselineOf="@+id/submit"
            app:layout_constraintEnd_toStartOf="@+id/submit"
            app:layout_constraintStart_toStartOf="parent" />

        <Button
            android:id="@+id/submit"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginTop="@dimen/default_margin"
            android:text="@string/submit"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/skip"
            app:layout_constraintTop_toBottomOf="@+id/textField"
/>

        <TextView
            android:id="@+id/textView_instructions"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/instructions"
            android:textSize="17sp"
            app:layout_constraintBottom_toTopOf="@+id/textField"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toBottomOf="@+id/textView_unscrambled_w
ord" />

        <TextView
            android:id="@+id/textView_unscrambled_word"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="@dimen/default_margin"
            android:layout_marginBottom="@dimen/default_margin"

android:textAppearance="@style/TextAppearance.MaterialComponents
.Headline3"
```

```
app:layout_constraintBottom_toTopOf="@+id/textView_instructions"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toBottomOf="@+id/word_count"
            tools:text="Scramble word" />

        <TextView
            android:id="@+id/word_count"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/word_count"

android:textAppearance="@style/TextAppearance.MaterialComponents
.Headline6"

app:layout_constraintBottom_toTopOf="@+id/textView_unscrambled_w
ord"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="3 of 10 words" />

        <TextView
            android:id="@+id/score"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/score"
            android:textAllCaps="true"

android:textAppearance="@style/TextAppearance.MaterialComponents
.Headline6"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="Score: 20" />

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/textField"

style="@style/Widget.Unscramble.TextInputLayout.OutlinedBox"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginTop="@dimen/default_margin"
            android:hint="@string/enter_your_word"
            app:errorIconDrawable="@drawable/ic_error"

app:helperTextTextAppearance="@style/TextAppearance.MaterialComp
onents.Subtitle1"
```

```
            app:layout_constraintBottom_toTopOf="@+id/submit"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"

app:layout_constraintTop_toBottomOf="@+id/textView_instructions"
>


<com.google.android.material.textfield.TextInputEditText
                android:id="@+id/text_input_edit_text"
                android:layout_width="match_parent"
                android:layout_height="match_parent"

android:inputType="textPersonName|textNoSuggestions"
                android:maxLines="1" />
        </com.google.android.material.textfield.TextInputLayout>

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

https://github.com/google-developer-training/android-basics-kotlin-unscramble-app/blob/starter/app/src/main/res/layout/game_fragment.xml

Bước 15. Khai báo các biến trạng thái trong GameFragment

```
private var score = 0
private var currentWordCount = 0
private var currentScrambledWord = "test"
```

Bước 16. Bổ sung trong GameFragment các hàm sau

```
/*
 * Gets a random word for the list of words and shuffles the
letters in it.
 */
private fun getNextScrambledWord(): String {
    val tempWord = allWordsList.random().toCharArray()
    tempWord.shuffle()
    return String(tempWord)
}
```

Hàm này để tạo một từ mới với các ký tự được trộn từ các ký tự lấy của một từ trong từ điển.

```
/*
* Checks the user's word, and updates the score accordingly.
* Displays the next scrambled word.
*/
private fun onSubmitWord() {
```

```
        currentScrambledWord = getNextScrambledWord()
        currentWordCount++
        score += SCORE_INCREASE
        binding.wordCount.text = getString(R.string.word_count,
currentWordCount, MAX_NO_OF_WORDS)
        binding.score.text = getString(R.string.score, score)
        setErrorTextField(false)
        updateNextWordOnScreen()
    }

    /*
     * Skips the current word without changing the score.
     * Increases the word count.
     */
    private fun onSkipWord() {
        currentScrambledWord = getNextScrambledWord()
        currentWordCount++
        binding.wordCount.text = getString(R.string.word_count,
currentWordCount, MAX_NO_OF_WORDS)
        setErrorTextField(false)
        updateNextWordOnScreen()
    }

    /*
     * Re-initializes the data in the ViewModel and updates the
views with the new data, to
     * restart the game.
     */
    private fun restartGame() {
        setErrorTextField(false)
        updateNextWordOnScreen()
    }

    /*
     * Exits the game.
     */
    private fun exitGame() {
        activity?.finish()
    }

    /*
    * Sets and resets the text field error status.
    */
    private fun setErrorTextField(error: Boolean) {
        if (error) {
            binding.textField.isErrorEnabled = true
            binding.textField.error = getString(R.string.try_again)
```

```
        } else {
            binding.textField.isErrorEnabled = false
            binding.textInputEditText.text = null
        }
    }

    /*
     * Displays the next scrambled word on screen.
     */
    private fun updateNextWordOnScreen() {
        binding.textViewUnscrambledWord.text = currentScrambledWord
    }
```

Bước 18. Bổ sung trong GameFragment hàm onViewCreated

```
override fun onViewCreated(view: View, savedInstanceState:
Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    // Setup a click listener for the Submit and Skip buttons.
    binding.submit.setOnClickListener { onSubmitWord() }
    binding.skip.setOnClickListener { onSkipWord() }
    // Update the UI
    updateNextWordOnScreen()
    binding.score.text = getString(R.string.score, 0)
    binding.wordCount.text = getString(
        R.string.word_count, 0, MAX_NO_OF_WORDS)
}
```

Các vấn đề với Starter code

As you played the game, you may have observed the following bugs:

1. On clicking the Submit button, the app does not check the player's word. The player always scores points.

2. There is no way to end the game. The app lets you play beyond 10 words.

3. The game screen shows a scrambled word, player's score, and word count. Change the screen orientation by rotating the device or emulator. Notice that the current word, score, and word count are lost and the game restarts from the beginning.

Các vấn đề này sẽ được giải quyết trong bài thực hành Lab 8.1. Store data in ViewModel.

Lab 8.1 Store data in ViewModel

Họ và tên: Nguyễn Văn An

MSSV: 20215520

3. Learn about App Architecture

3.1. Giới thiệu

Kiến trúc ứng dụng cung cấp bộ hướng dẫn nhằm phân chia hợp lý trách nhiệm giữa các lớp trong một ứng dụng. Kiến trúc được thiết kế tốt giúp 1ứng dụng có thể mở rộng, bổ sung tính năng và đảm bảo hợp tác nhóm hiệu quả. Báo cáo này đề cập đến nguyên tắc chính là tách biệt trách nhiệm và xây dựng giao diện từ model. Ngoài ra, báo cáo giới thiệu các thành phần chính trong kiến trúc Android: UI Controller (Activity/Fragment), ViewModel, LiveData và Room, nhấn mạnh vai trò và trách nhiệm của chúng.

3.2. Nguyên tắc kiến trúc chính

3.2.1. Tách biệt trách nhiệm

Nguyên tắc tách biệt trách nhiệm khuyên cáo rằng một ứng dụng nên được chia thành các lớp khác nhau, mỗi lớp đảm nhận một trách nhiệm riêng. Sự phân chia này giúp:

- Tăng tính bảo trì bằng cách tách biệt chức năng.
- Dễ dàng gắp lỗi và kiểm tra các thành phần riêng lẻ.
- Tăng cường hợp tác nhóm khi mỗi nhóm làm việc trên một lớp riêng.

3.2.2. Xây dựng giao diện từ Model

Nguyên tắc quan trọng khác là giao diện người dùng (UI) nên được xây dựng dựa trên Model, tốt nhất là model lưu trữ. Model là các thành phần quản lý và xử lý dữ liệu cho ứng dụng. Bằng cách tách biệt Views và các thành phần khác trong ứng dụng, Model không bị ảnh hưởng bởi chu kỳ sống của ứng dụng, đảm bảo quản lý dữ liệu một cách nhất quán.

3.3. Các thành phần chính trong Kiến trúc Android

3.3.1. UI Controller (Activity/Fragment)

UI Controller bao gồm Activities và Fragments, đảm nhận quản lý giao diện người dùng. Chúng:

- Hiển thị Views lên màn hình.
- Xử lý các sự kiện tương tác người dùng.
- Quản lý các nhiệm vụ liên quan đến giao diện như hiệu ứng hoặc cập nhật giao diện.

Lưu ý quan trọng:

- **Không chứa dữ liệu hoặc logic ra quyết định:** Logic xử lý dữ liệu và quản lý trạng thái nên được xử lý trong ViewModel, để tránh các vấn đề liên quan đến chu kỳ sống.
- **Nhạy cảm với chu kỳ sống:** Hệ thống Android có thể hủy UI Controller do tình trạng hệ thống như hết bộ nhớ hoặc do tương tác người dùng. Do đó, trạng thái hoặc dữ liệu được khuyến cáo không lưu trực tiếp trong Activity hoặc Fragment.

Ví dụ: Trong ứng dụng Unscramble, từ ngữa xố, điểm số và số từ hiện tại được hiển thị trong Fragment. Tuy nhiên, logic để xác định từ ngữa tiếp theo, tính điểm và số từ được đặt trong ViewModel.

3.3.2. ViewModel

ViewModel đóng vai trò là cầu nối giữa UI và dữ liệu ứng dụng, tuân thủ nguyên tắc kiến trúc xây dựng UI từ model. ViewModel:

- **Quản lý dữ liệu ứng dụng:** Xử lý và quản lý tất cả dữ liệu cần thiết cho UI.
- **Độc lập với UI Controller:** Tránh tham chiếu trực tiếp đến

4. Add a ViewModel

4.1. Giới Thiệu

Trong bài học này, bạn sẽ thêm ViewModel vào ứng dụng để quản lý dữ liệu của ứng dụng như từng tự (scrambled word), số lượng từ (word count), và điểm số (score). Kiến trúc ứng dụng sẽ được thiết kế như sau:

- **MainActivity** chứa **GameFragment**.
- **GameFragment** sẽ truy cập thông tin về trò chơi từ **GameViewModel**.

4.2. Thực Hiện

Bước 1: Kiểm tra ViewModel Library

1. Trong cửa sổ Android Studio, mở mục **Gradle Scripts** và chọn tệp **build.gradle (Module: Unscramble.app)**.
2. Xác minh rằng đã có ViewModel library ở trong khối **dependencies**. Phần này đã được thiết lập sẵn. Ví dụ:

```
// ViewModel
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
```

Luôn khuyên dùng phiên bản mới nhất của thư viện ViewModel.

Bước 2: Tạo GameViewModel

1. Trong cửa sổ Android Studio, nhấp chuột phải vào mục **ui.game**, sau đó chọn **New > Kotlin File/Class**.
2. Đặt tên tệp là **GameViewModel** và chọn loại **Class**.
3. Sửa lớp **GameViewModel** để kế thừa từ lớp trửu tượng **ViewModel**. Ví dụ:

```
class GameViewModel : ViewModel() {
}
```

Bước 3: Gán ViewModel cho Fragment

1. Trong lớp **GameFragment**, khai báo thuộc tính để tham chiếu đến GameViewModel:

```
private val viewModel: GameViewModel by viewModels()
```

2. Nhập thư viện sau nếu được Android Studio nhắc nhở:

```
import androidx.fragment.app.viewModels
```

4.3. Khái Niệm Quan Trọng

Kotlin Property Delegate

- Trong Kotlin, thuộc tính có hai hàm getter và setter mặc định cho thuộc tính mutable (**var**). Riêng thuộc tính immutable (**val**) chỉ có getter.
- Property delegation giúp giao trách nhiệm getter-và-setter cho một lớp khác.

Cú pháp:

```
var <property-name> : <property-type> by <delegate-class>()
```

Trong trường hợp ViewModel:

- Nếu khởi tạo trực tiếp ViewModel như sau:

```
private val viewModel = GameViewModel()
```

Thì khi xoay màn hình (configuration change), ứng dụng sẽ tạo mới ViewModel và làm mất dữ liệu của ViewModel.

- Thay vì đó, sử dụng property delegation như sau:

```
private val viewModel: GameViewModel by viewModels()
```

Lúc này, việc khởi tạo và duy trì ViewModel được xử lý bởi class **viewModels**. ViewModel sẽ được giữ nguyên khi có thay đổi về cấu hình.

5. Move data to the ViewModel

5.1. Giới Thiệu

Trong bài học này, bạn sẽ thêm ViewModel vào ứng dụng để quản lý dữ liệu của ứng dụng như từng tự (scrambled word), số lượng từ (word count), và điểm số (score). Kiến trúc ứng dụng sẽ được thiết kế như sau:

- **MainActivity** chứa **GameFragment**.
- **GameFragment** sẽ truy cập thông tin về trò chơi từ **GameViewModel**.

5.2. Thực Hiện

Bước 1: Kiểm tra ViewModel Library

1. Trong cửa sổ Android Studio, mở mục **Gradle Scripts** và chọn tệp **build.gradle (Module: Unscramble.app)**.
2. Xác minh rằng đã có ViewModel library ở trong khối **dependencies**. Phần này đã được thiết lập sẵn. Ví dụ:

```
// ViewModel
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
```

Luôn khuyên dùng phiên bản mới nhất của thư viện ViewModel.

Bước 2: Tạo GameViewModel

1. Trong cửa sổ Android Studio, nhấp chuột phải vào mục **ui.game**, sau đó chọn **New > Kotlin File/Class**.
2. Đặt tên tệp là **GameViewModel** và chọn loại **Class**.
3. Sửa lớp **GameViewModel** để kế thừa từ lớp trửu tượng **ViewModel**. Ví dụ:

```
class GameViewModel : ViewModel() {
}
```

Bước 3: Gán ViewModel cho Fragment

1. Trong lớp **GameFragment**, khai báo thuộc tính để tham chiếu đến GameViewModel:

```
private val viewModel: GameViewModel by viewModels()
```

2. Nhập thư viện sau nếu được Android Studio nhắc nhở:

```
import androidx.fragment.app.viewModels
```

5.3. Khái Niệm Quan Trọng

Kotlin Property Delegate

- Trong Kotlin, thuộc tính có hai hàm getter và setter mặc định cho thuộc tính mutable (**var**). Riêng thuộc tính immutable (**val**) chỉ có getter.
- Property delegation giúp giao trách nhiệm getter-và-setter cho một lớp khác.

Cú pháp:

```
var <property-name> : <property-type> by <delegate-class>()
```

Trong trường hợp ViewModel:

- Nếu khởi tạo trực tiếp ViewModel như sau:

```
private val viewModel = GameViewModel()
```

Thì khi xoay màn hình (configuration change), ứng dụng sẽ tạo mới ViewModel và làm mất dữ liệu của ViewModel.

- Thay vì đó, sử dụng property delegation như sau:

```
private val viewModel: GameViewModel by viewModels()
```

Lúc này, việc khởi tạo và duy trì ViewModel được xử lý bởi class **viewModels**. ViewModel sẽ được giữ nguyên khi có thay đổi về cấu hình.

6. The lifecycle of a ViewModel

ViewModel trong Android được thiết kế để duy trì dữ liệu giao diện (UI data) ngay cả khi Fragment hoặc Activity bị phá hủ vì thay đổi cấu hình (như xoay màn hình). Khi một Fragment hoặc Activity mới được tạo, nó sẽ kết nối lại với ViewModel đã tồn tại, thay vì tạo mới ViewModel.

Bài viết này sẽ giúp bạn hiểu rõ chu kỳ sống của ViewModel và cách nó được duy trì trong ứng dụng Android.

6.1. Tìm hiểu vòng đời ViewModel

Đặc điểm

1. ViewModel được duy trì trong suốt vòng đời của Activity hoặc Fragment.
2. Khi Activity hoặc Fragment bị phá hủ vì thay đổi cấu hình, ViewModel không bị phá hủ.

3. Khi Activity hoặc Fragment mới được tạo, nó kết nối với instance đã tồn tại của ViewModel.
4. ViewModel chỉ bị phá hủ khi vòng đời của Fragment hoặc Activity kết thúc hoàn toàn.

Biểu đồ chu kỳ sống

- Khi Fragment hoặc Activity được tạo: ViewModel được khởi tạo lần đầu.
- Khi Fragment hoặc Activity bị phá hủ vì thay đổi cấu hình: ViewModel được duy trì.
- Khi Fragment hoặc Activity kết thúc hoàn toàn: ViewModel bị phá hủ.

6.2. Thêm logging để theo dõi chu kỳ sống ViewModel

6.2.1. Logging trong GameViewModel

Thêm khối init:

```
class GameViewModel : ViewModel() {
    init {
        Log.d("GameFragment", "GameViewModel created!")
    }
}
```

- Khối `init` được gọ i khi ViewModel được tạo lần đầu.

Ghi đè onCleared():

```
override fun onCleared() {
    super.onCleared()
    Log.d("GameFragment", "GameViewModel destroyed!")
}
```

- Phương thức `onCleared` được gọ i trước khi ViewModel bị phá hủ.

6.2.2. Logging trong GameFragment

Ghi log khi tạo Fragment:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    binding = GameFragmentBinding.inflate(inflater, container, false)
    Log.d("GameFragment", "GameFragment created/re-created!")
    return binding.root
}
```

Ghi log khi Fragment bị phá hủ:

```
override fun onDetach() {
```

```
    super.onDetach()
    Log.d("GameFragment", "GameFragment destroyed!")
}
```

6.3. Kết quả trong Logcat

1. **Khi tạo Fragment và ViewModel:**

```
D/GameFragment: GameFragment created/re-created!
D/GameFragment: GameViewModel created!
```

2. **Khi xoay màn hình (thay đổi cấu hình):**

```
D/GameFragment: GameFragment destroyed!
D/GameFragment: GameFragment created/re-created!
```

(ViewModel không bị tạo lại.)

3. **Khi thoát khỏi app hoặc Fragment bị phá hủ hoàn toàn:**

```
D/GameFragment: GameViewModel destroyed!
D/GameFragment: GameFragment destroyed!
```

Store data in ViewModel

1. Before you begin

You have learned in the previous codelabs the lifecycle of activities and fragments and the related lifecycle issues with configuration changes. To save the app data, saving the instance state is one option, but it comes with its own limitations. In this codelab you learn about a robust way to design your app and preserve app data during configuration changes, by taking advantage of Android Jetpack libraries.

Android Jetpack libraries are a collection of libraries to make it easier for you to develop great Android apps. These libraries help you follow best practices, free you from writing boilerplate code, and simplify complex tasks, so you can focus on the code you care about, like the app logic.

Android Architecture Components are part of Android Jetpack libraries, to help you design apps with good architecture. Architecture Components provide guidance on app architecture, and it is the recommended best practice.

App architecture is a set of design rules. Much like the blueprint of a house, your architecture provides the structure for your app. A good app architecture can make your code robust, flexible, scalable and maintainable for years to come.

- This layout contains a text field for the player's word, along with `TextViews` to display score and word count. It also has instructions and buttons (**Submit** and **Skip**) to play the game.

main_activity.xml

Defines the main activity layout with a single game fragment.

res/values folder

You are familiar with the resource files in this folder.

- `colors.xml` contains the theme colors used in the app
- `strings.xml` contains all the strings your app needs
- `themes` and `styles` folders contain the UI customization done for your app

MainActivity.kt

Contains the default template generated code to set the activity's content view as `main_activity.xml`.

ListOfWords.kt

This file contains a list of the words used in the game, as well as constants for the maximum number of words per game and the number of points the player scores for every correct word.

WARNING: It is not a recommended practice to hardcode strings in the code, strings should be in `strings.xml` for easier localization. To keep things simple, and to focus on Architecture Components, strings are hardcoded in this app.

GameFragment.kt

This is the only fragment in your app, where most of the game's action takes place:

- Variables are defined for the current scrambled word (`currentScrambledWord`), word count (`currentWordCount`), and the score (`score`).
- Binding object instance with access to the `game_fragment` views called `binding` is defined.
- `onCreateView()` function inflates the `game_fragment` layout XML using the binding object.
- `onViewCreated()` function sets up the button click listeners and updates the UI.
- `onSubmitWord()` is the click listener for the **Submit** button, this function displays the next scrambled word, clears the text field, and increases the score and word count without validating the player's word.
- `onSkipWord()` is the click listener for the **Skip** button, this function updates the UI similar to `onSubmitWord()` except the score.

- `getNextScrambledWord()` is a helper function that picks a random word from the list of words and shuffles the letters in it.
- `restartGame()` and `exitGame()` functions are used to restart and end the game respectively, you will use these functions later.
- `setErrorTextField()` clears the text field content and resets the error status.
- `updateNextWordOnScreen()` function displays the new scrambled word.

3. Learn about App Architecture

Architecture provides you with the guidelines to help you allocate responsibilities in your app, between the classes. A well-designed app architecture helps you scale your app and extend it with additional features in the future. It also makes team collaboration easier.

The most common architectural principles are: separation of concerns and driving UI from a model.

Separation of concerns

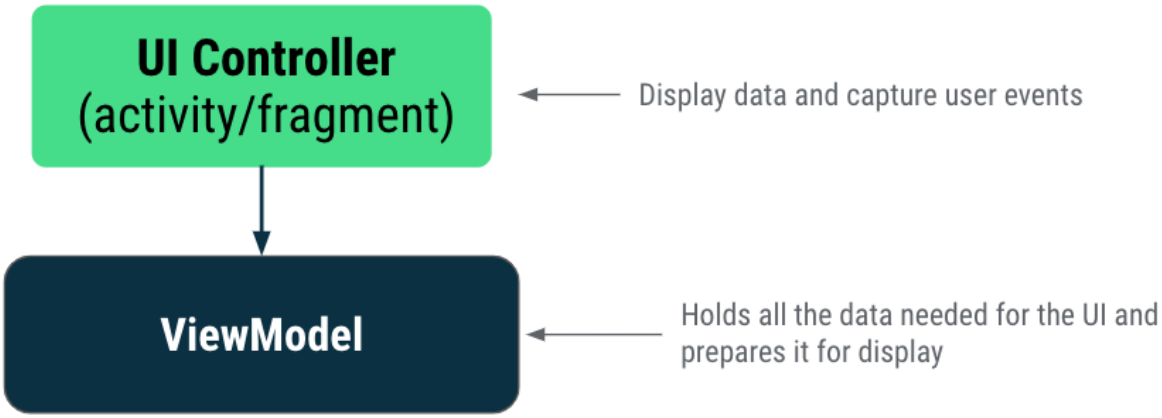
The separation of concerns design principle states that the app should be divided into classes, each with separate responsibilities.

Drive UI from a model

Another important principle is that you should drive your UI from a model, preferably a persistent model. Models are components that are responsible for handling the data for an app. They're independent from the `Views` and app components in your app, so they're unaffected by the app's lifecycle and the associated concerns.

The main classes or components in Android Architecture are UI Controller (activity/fragment), `ViewModel`, `LiveData` and `Room`. These components take care of some of the complexity of the lifecycle and help you avoid lifecycle related issues. You learn about `LiveData` and `Room` in later codelabs.

This diagram shows a basic portion of the architecture:



UI controller (Activity / Fragment)

Activities and fragments are UI controllers. UI controllers control the UI by drawing views on the screen, capturing user events, and anything else related to the UI that the user interacts with. Data in the app or any decision-making logic about that data should not be in the UI controller classes.

The Android system can destroy UI controllers at any time based on certain user interactions or because of system conditions like low memory. Because these events aren't under your control, you shouldn't store any app data or state in UI controllers. Instead, the decision-making logic about the data should be added in your `ViewModel`.

For example, in your **Unscramble** app, the scrambled word, score, and word count are displayed in a fragment (UI controller). The decision-making code such as figuring out the next scrambled word, and calculations of score and word count should be in your `ViewModel`.

ViewModel

The `ViewModel` is a model of the app data that is displayed in the views. Models are components that are responsible for handling the data for an app. They allow your app to follow the architecture principle, driving the UI from the model.

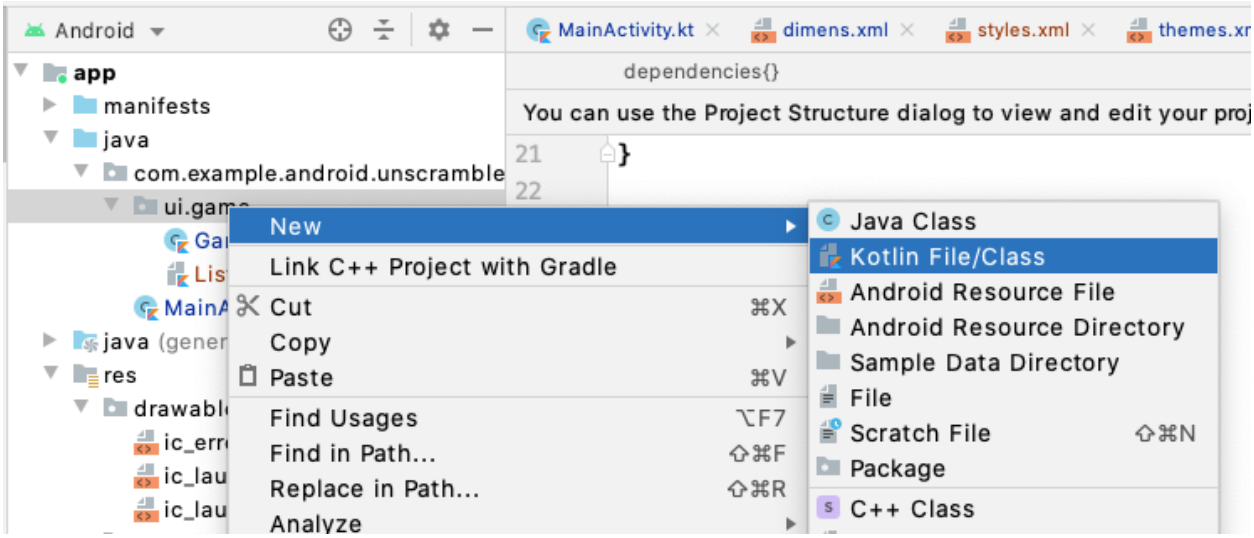
The `ViewModel` stores the app related data that isn't destroyed when activity or fragment is destroyed and recreated by the Android framework. `ViewModel` objects are automatically retained (they are not destroyed like the activity or a fragment instance) during configuration changes so that data they hold is immediately available to the next activity or fragment instance.

To implement `ViewModel` in your app, extend the `ViewModel` class, which is from the architecture components library, and store app data within that class.

```
// ViewModel
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
```

It is recommended to always use the latest version of the library in spite of the version mentioned in the codelab.

3. Create a new Kotlin class file called `GameViewModel`. In the **Android** window, right click on the **ui.game** folder. Select **New** > **Kotlin File/Class**.



4. Give it the name `GameViewModel`, and select **Class** from the list.
5. Change `GameViewModel` to be subclassed from `ViewModel`. `ViewModel` is an abstract class, so you need to extend it to use it in your app. See the `GameViewModel` class definition below.

```
class GameViewModel : ViewModel() {
}
```

Attach the ViewModel to the Fragment

To associate a `ViewModel` to a UI controller (activity / fragment), create a reference (object) to the `ViewModel` inside the UI controller.

In this step, you create an object instance of the `GameViewModel` inside the corresponding UI controller, which is `GameFragment`.

1. At the top of the `GameFragment` class, add a property of type `GameViewModel`.
2. Initialize the `GameViewModel` using the `by viewModels()` Kotlin property delegate. You will learn more about it in the next section.

```
private val viewModel: GameViewModel by viewModels()
```

3. If prompted by Android Studio, import `androidx.fragment.app.viewModels`.

Kotlin property delegate

In Kotlin, each mutable (`var`) property has default getter and setter functions automatically generated for it. The setter and getter functions are called when you assign a value or read the value of the property.

For a read-only property (`val`), it differs slightly from a mutable property. Only the getter function is generated by default. This getter function is called when you read the value of a read-only property.

Property delegation in Kotlin helps you to handoff the getter-setter responsibility to a different class.

This class (called *delegate class*) provides getter and setter functions of the property and handles its changes.

A delegate property is defined using the `by` clause and a delegate class instance:

```
// Syntax for property delegation
var <property-name> : <property-type> by <delegate-class>()
```

In your app, if you initialize the view model using default `GameViewModel` constructor, like below:

```
private val viewModel = GameViewModel()
```

Then the app will lose the state of the `viewModel` reference when the device goes through a configuration change. For example, if you rotate the device, then the activity is destroyed and created again, and you'll have a new view model instance with the initial state again.

Instead, use the property delegate approach and delegate the responsibility of the `viewModel` object to a separate class called `viewModels`. That means when you access the `viewModel` object, it is handled internally by the delegate class, `viewModels`. The delegate class creates the `viewModel` object for you on the first access, and retains its value through configuration changes and returns the value when requested.

5. Move data to the ViewModel

Separating your app's UI data from the UI controller (your `Activity` / `Fragment` classes) lets you better follow the single responsibility principle we discussed above. Your activities and fragments are responsible for drawing views and data to the screen, while your `ViewModel` is responsible for holding and processing all the data needed for the UI.

In this task, you move the data variables from `GameFragment` to `GameViewModel` class.

1. Move the data variables `score`, `currentWordCount`, `currentScrambledWord` to `GameViewModel` class.

```
class GameViewModel : ViewModel() {

    private var score = 0
    private var currentWordCount = 0
    private var currentScrambledWord = "test"
...
```

2. Notice the errors about unresolved references. This is because properties are private to the `ViewModel` and are not accessible by your UI controller. You'll fix these errors next.

To resolve this issue, you can't make the visibility modifiers of the properties `public`—the data should not be editable by other classes. This is risky because an outside class could change the data in unexpected ways that don't follow the game rules specified in the view model. For example, an outside class could change the `score` to a negative value.

Inside the `ViewModel`, the data should be editable, so they should be `private` and `var`. From outside the `ViewModel`, data should be readable, but not editable, so the data should be exposed as `public` and `val`. To achieve this behavior, Kotlin has a feature called a backing property.

Backing property

A backing property allows you to return something from a getter other than the exact object.

You have already learned that for every property, the Kotlin framework generates getters and setters.

For getter and setter methods, you could override one or both of these methods and provide your own custom behavior. To implement a backing property, you will override the getter method to return a read-only version of your data. Example of backing property:

```
// Declare private mutable variable that can only be modified
// within the class it is declared.
private var _count = 0

// Declare another public immutable field and override its getter method.
// Return the private property's value in the getter method.
// When count is accessed, the get() function is called and
// the value of _count is returned.
val count: Int
   get() = _count
```

Consider an example, in your app you want the app data to be private to the `ViewModel`:

Inside the `ViewModel` class:

- The property `_count` is `private` and mutable. Hence, it is only accessible and editable within the `ViewModel` class. The convention is to prefix the `private` property with an underscore.

Outside the `ViewModel` class:

- The default visibility modifier in Kotlin is `public`, so `count` is public and accessible from other classes like UI controllers. Since only the `get()` method is being overridden, this property is immutable and read-only. When an outside class accesses this property, it returns the value of `_count` and its value can't be modified. This protects the app data inside the `ViewModel` from unwanted and unsafe changes by external classes, but it allows external callers to safely access its value.

Add backing property to currentScrambledWord

1. In `GameViewModel` change the `currentScrambledWord` declaration to add a backing property. Now `_currentScrambledWord` is accessible and editable only within the `GameViewModel`. The UI controller, `GameFragment` can read its value using the read-only property, `currentScrambledWord`.

```
private var _currentScrambledWord = "test"
val currentScrambledWord: String
    get() = _currentScrambledWord
```

2. In `GameFragment`, update the method `updateNextWordOnScreen()` to use the read-only `viewModel` property, `currentScrambledWord`.

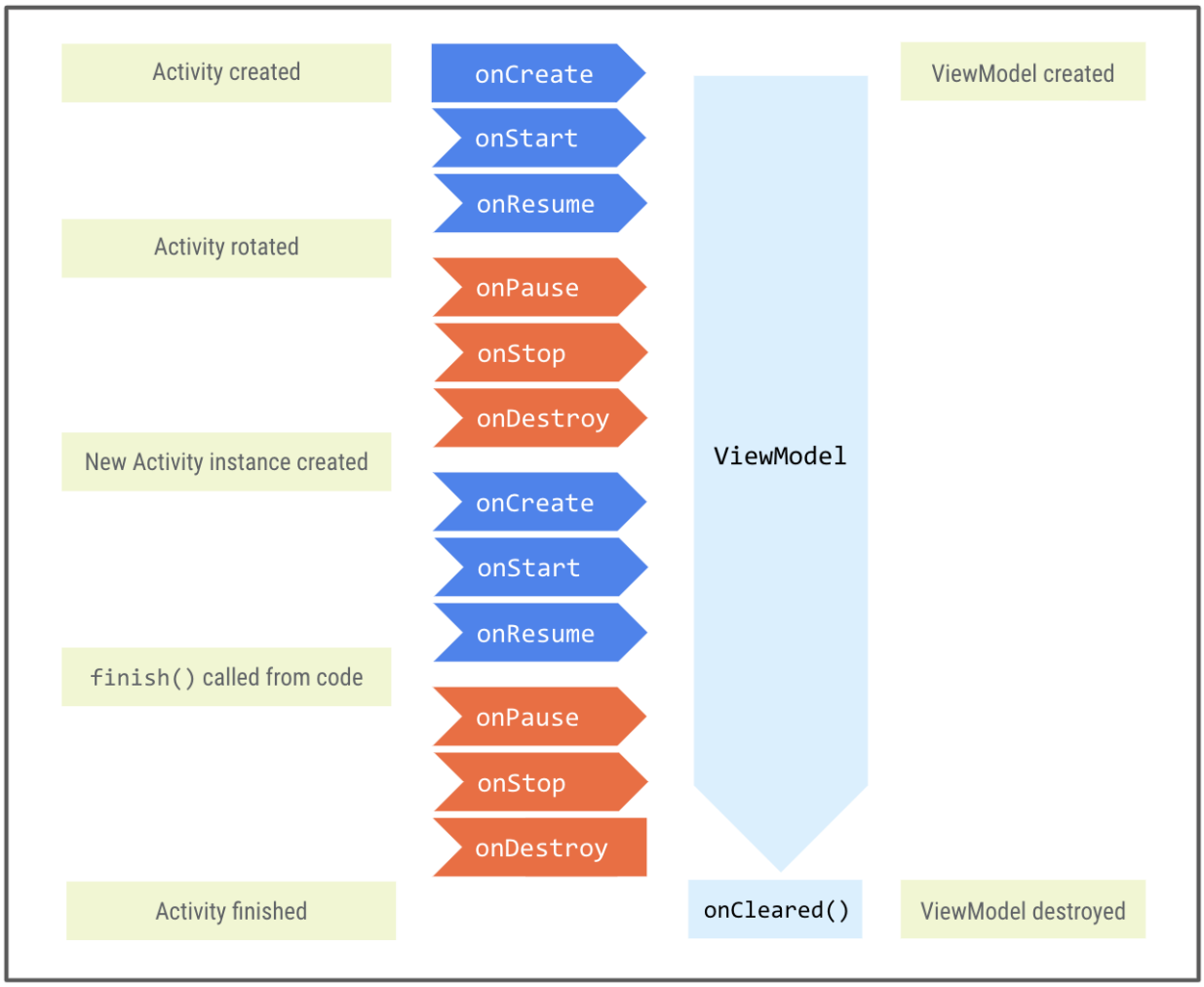
```
private fun updateNextWordOnScreen() {
    binding.textViewUnscrambledWord.text = viewModel.currentScrambledWord
}
```

3. In `GameFragment`, delete the code inside the methods `onSubmitWord()` and `onSkipWord()`. You will implement these methods later. You should be able to compile the code now without errors.

Warning: Never expose mutable data fields from your `ViewModel`—make sure this data can't be modified from another class. Mutable data inside the `ViewModel` should always be `private`.

6. The lifecycle of a ViewModel

The framework keeps the `ViewModel` alive as long as the scope of the activity or fragment is alive. A `ViewModel` is not destroyed if its owner is destroyed for a configuration change, such as screen rotation. The new instance of the owner reconnects to the existing `ViewModel` instance, as illustrated by the following diagram:



Understand ViewModel lifecycle

Add logging in the `GameViewModel` and `GameFragment` to help you better understand the lifecycle of the `ViewModel`.

1. In `GameViewModel.kt` add an `init` block with a log statement.

```
class GameViewModel : ViewModel() {
    init {
        Log.d("GameFragment", "GameViewModel created!")
    }

    ...
}
```

Kotlin provides the initializer block (also known as the `init` block) as a place for initial setup code needed during the initialization of an object instance. Initializer blocks are prefixed with the

`init` keyword followed by the curly braces `{}`. This block of code is run when the object instance is first created and initialized.

2. In the `GameViewModel` class, override the `onCleared()` method. The `ViewModel` is destroyed when the associated fragment is detached, or when the activity is finished. Right before the `ViewModel` is destroyed, the `onCleared()` callback is called.
3. Add a log statement inside `onCleared()` to track the `GameViewModel` lifecycle.

```
override fun onCleared() {
    super.onCleared()
    Log.d("GameFragment", "GameViewModel destroyed!")
}
```

4. In `GameFragment` inside `onCreateView()`, after you get a reference to the binding object, add a log statement to log the creation of the fragment. The `onCreateView()` callback will be triggered when the fragment is created for the first time and also every time it is re-created for any events like configuration changes.

```
override fun onCreateView(
   inflater: LayoutInflater, container: ViewGroup?,
   savedInstanceState: Bundle?
): View {
   binding = GameFragmentBinding.inflate(inflater, container, false)
   Log.d("GameFragment", "GameFragment created/re-created!")
   return binding.root
}
```

5. In `GameFragment`, override the `onDetach()` callback method, which will be called when the corresponding activity and fragment are destroyed.

```
override fun onDetach() {
    super.onDetach()
    Log.d("GameFragment", "GameFragment destroyed!")
}
```

6. In Android Studio, run the app, open the **Logcat** window and filter on `GameFragment`. Notice that `GameFragment` and the `GameViewModel` are created.

```
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: GameViewModel created!
```

7. Enable the auto-rotate setting on your device or emulator and change the screen orientation a few times. The `GameFragment` is destroyed and recreated each time, but the `GameViewModel` is created only once, and it is not re-created or destroyed for each call.

```
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: GameViewModel created!
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
```

```
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
```

8. Exit the game or navigate out of the app using the back arrow. The `GameViewModel` is destroyed, and the callback `onCleared()` is called. The `GameFragment` is destroyed.

```
com.example.android.unscramble D/GameFragment: GameViewModel destroyed!
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
```

7. Populate ViewModel

In this task, you further populate the `GameViewModel` with helper methods for getting the next word, validating the player's word to increase the score, and checking the word count to end the game.

Late initialization

Typically when you declare a variable, you provide it with an initial value upfront. However, if you're not ready to assign a value yet, you could initialize it later. To late initialize a property in Kotlin you use the keyword `lateinit`, which means late initialization. If you guarantee that you will initialize the property before using it, you can declare the property with `lateinit`. Memory is not allocated to the variable until it is initialized. If you try to access the variable before initializing it, the app will crash.

Get next word

Create the `getNextWord()` method in the `GameViewModel` class, with the following functionality:

- Get a random word from the `allWordsList` and assign it to `currentWord`.
- Create a scrambled word by scrambling the letters in the `currentWord` and assign it to the `currentScrambledWord`
- Handle the case where the scrambled word is the same as the unscrambled word.
- Make sure you don't show the same word twice during the game.

Implement the following steps in `GameViewModel` class:

1. In `GameViewModel`, add a new class variable of type `MutableList<String>` called `wordsList`, to hold a list of words you use in the game, to avoid repetitions.
2. Add another class variable called `currentWord` to hold the word the player is trying to unscramble. Use the `lateinit` keyword since you will initialize this property later.

```
private var wordsList: MutableList<String> = mutableListOf()
private lateinit var currentWord: String
```

3. Add a new `private` method called `getNextWord()`, above the `init` block, with no parameters that returns nothing.
4. Get a random word from the `allWordsList` and assign it to `currentWord`.

```
private fun getNextWord() {
   currentWord = allWordsList.random()
}
```

5. In `getNextWord()`, convert the `currentWord` string to an array of characters and assign it to a new `val` called `tempWord`. To scramble the word, shuffle characters in this array using the Kotlin method, `shuffle()`.

```
val tempWord = currentWord.toCharArray()
tempWord.shuffle()
```

An `Array` is similar to a `MutableList`, but it has a fixed size when it's initialized. An `Array` cannot expand or shrink its size (you need to copy an array to resize it) whereas a `MutableList` has `add()` and `remove()` functions, so that it can increase and decrease in size.

6. Sometimes the shuffled order of characters is the same as the original word. Add the following `while` loop around the call to shuffle, to continue the loop until the scrambled word is not the same as the original word.

```
while (String(tempWord).equals(currentWord, false)) {
    tempWord.shuffle()
}
```

7. Add an `if-else` block to check if a word has been used already. If the `wordsList` contains `currentWord`, call `getNextWord()`. If not, update the value of `_currentScrambledWord` with the newly scrambled word, increase the word count, and add the new word to the `wordsList`.

```
if (wordsList.contains(currentWord)) {
    getNextWord()
} else {
    _currentScrambledWord = String(tempWord)
    ++currentWordCount
    wordsList.add(currentWord)
}
```

8. Here is the completed `getNextWord()` method for your reference.

```
/*
* Updates currentWord and currentScrambledWord with the next word.
*/
private fun getNextWord() {
   currentWord = allWordsList.random()
   val tempWord = currentWord.toCharArray()
   tempWord.shuffle()

   while (String(tempWord).equals(currentWord, false)) {
       tempWord.shuffle()
   }
   if (wordsList.contains(currentWord)) {
       getNextWord()
   } else {
       _currentScrambledWord = String(tempWord)
       ++currentWordCount
       wordsList.add(currentWord)
   }
}
```

Late-initialize currentScrambledWord

Now you have created the `getNextWord()` method, to get the next scrambled word. You will make a call to it when the `GameViewModel` is initialized for the first time. Use the `init` block to initialize `lateinit` properties in the class such as the current word. The result will be that the first word displayed on the screen will be a scrambled word instead of **test**.

1. Run the app. Notice the first word is always "**test**".
2. To display a scrambled word at the start of the app, you need to call the `getNextWord()` method, which in turn updates `currentScrambledWord`. Make a call to the method `getNextWord()` inside the `init` block of the `GameViewModel`.

```
init {
    Log.d("GameFragment", "GameViewModel created!")
    getNextWord()
}
```

3. Add the `lateinit` modifier onto the `_currentScrambledWord` property. Add an explicit mention of the data type `String`, since no initial value is provided.

```
private lateinit var _currentScrambledWord: String
```

4. Run the app. Notice a new scrambled word is displayed at the app launch. Awesome!

Add a helper method

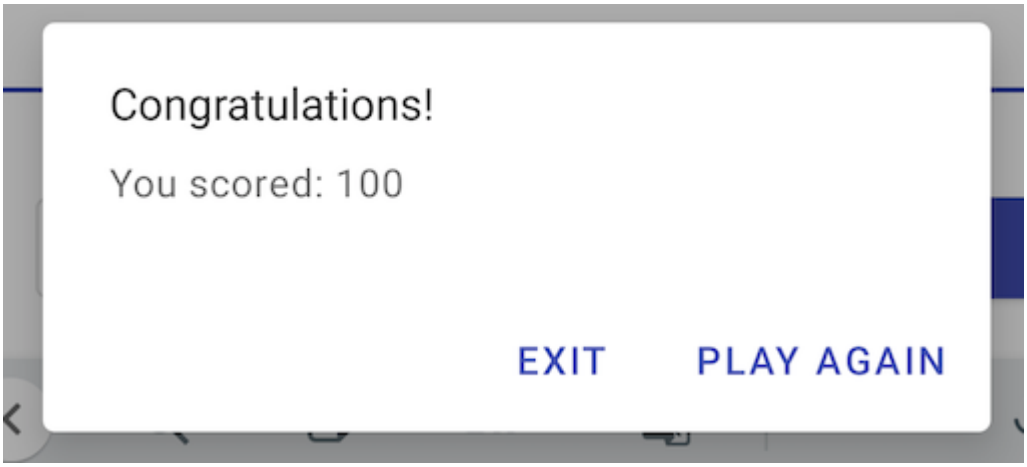
Next add a helper method to process and modify the data inside the `ViewModel`. You will use this method in later tasks.

1. In the `GameViewModel` class, add another method called `nextWord()`. Get the next word from the list and return `true` if the word count is less than the *`MAX_NO_OF_WORDS`*.

```
/*
* Returns true if the current word count is less than MAX_NO_OF_WORDS.
* Updates the next word.
*/
fun nextWord(): Boolean {
    return if (currentWordCount < MAX_NO_OF_WORDS) {
        getNextWord()
        true
    } else false
}
```

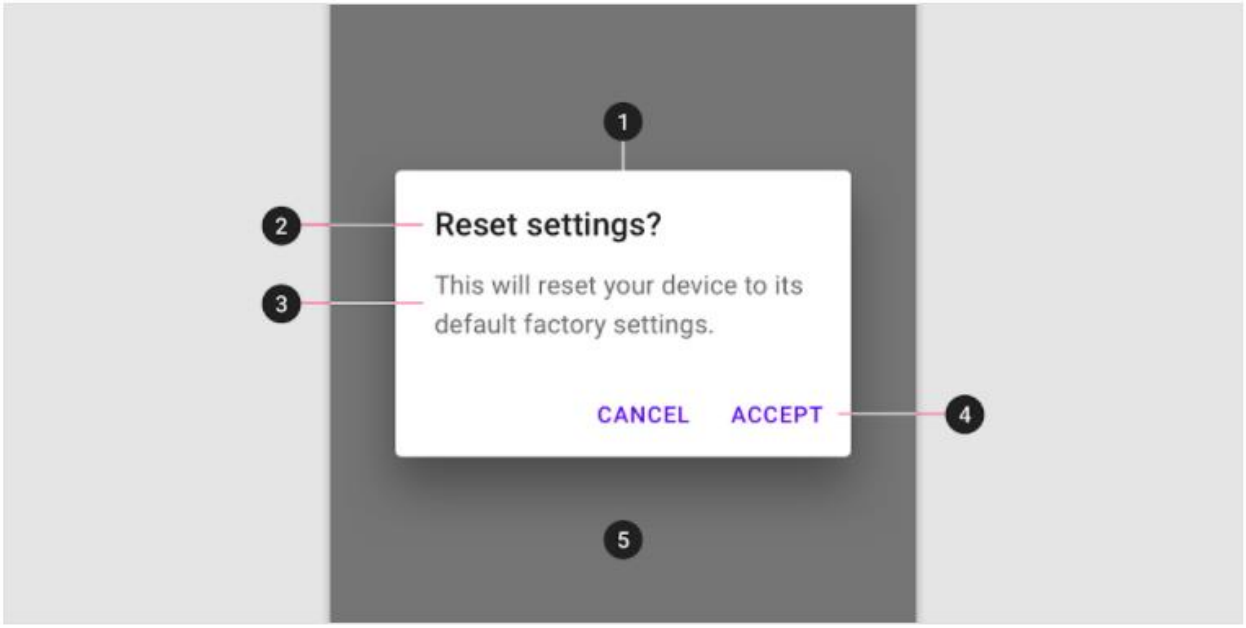
8. Dialogs

In the starter code, the game never ended, even after 10 words were played through. Modify your app so that after the user goes through 10 words, the game is over and you show a dialog with the final score. You will also give the user an option to play again or exit the game.



This is the first time you'll be adding a dialog to an app. A dialog is a small window (screen) that prompts the user to make a decision or enter additional information. Normally a dialog does not fill the entire screen, and it requires users to take an action before they can proceed. Android provides different types of Dialogs. In this codelab, you learn about Alert Dialogs.

Anatomy of alert dialog



1. Alert Dialog
2. Title (optional)
3. Message
4. Text buttons

Implement final score dialog

Use the `MaterialAlertDialog` from the Material Design Components library to add a dialog to your app that follows Material guidelines. Since a dialog is UI related, the `GameFragment` will be responsible for creating and showing the final score dialog.

1. First add a backing property to the `score` variable. In `GameViewModel`, change the `score` variable declaration to the following.

```
private var _score = 0
val score: Int
   get() = _score
```

2. In `GameFragment`, add a private function called `showFinalScoreDialog()`. To create a `MaterialAlertDialog`, use the `MaterialAlertDialogBuilder` class to build up parts of the dialog step-by-step. Call the `MaterialAlertDialogBuilder` constructor passing in the content using the fragment's `requireContext()` method. The `requireContext()` method returns a non-null `Context`.

```
/*
* Creates and shows an AlertDialog with the final score.
*/
private fun showFinalScoreDialog() {
```

```
    MaterialAlertDialogBuilder(requireContext())
}
```

As the name suggests, `Context` refers to the context or the current state of an application, activity, or fragment. It contains the information regarding the activity, fragment or application. Usually it is used to get access to resources, databases, and other system services. In this step, you pass the fragment context to create the alert dialog.

If prompted by Android Studio, `import com.google.android.material.dialog.MaterialAlertDialogBuilder`.

3. Add the code to set the title on the alert dialog, use a string resource from `strings.xml`.

```
MaterialAlertDialogBuilder(requireContext())
    .setTitle(getString(R.string.congratulations))
```

4. Set the message to show the final score, use the read-only version of the score variable (`viewModel.score`), you added earlier.

```
    .setMessage(getString(R.string.you_scored, viewModel.score))
```

5. Make your alert dialog not cancelable when the back key is pressed, using `setCancelable()` method and passing `false`.

```
    .setCancelable(false)
```

6. Add two text buttons **EXIT** and **PLAY AGAIN** using the methods `setNegativeButton()` and `setPositiveButton()`. Call `exitGame()` and `restartGame()` respectively from the lambdas.

```
    .setNegativeButton(getString(R.string.exit)) { _, _ ->
        exitGame()
    }
    .setPositiveButton(getString(R.string.play_again)) { _, _ ->
        restartGame()
    }
```

This syntax may be new to you, but this is shorthand for `setNegativeButton(getString(R.string.exit), { _, _ -> exitGame()})` where the `setNegativeButton()` method takes in two parameters: a `String` and a function, `DialogInterface.OnClickListener()` which can be expressed as a lambda. When the last argument being passed in is a function, you could place the lambda expression *outside* the parentheses. This is known as *trailing lambda syntax*. Both ways of writing the code (with the lambda inside or outside the parentheses) is acceptable. The same applies for the `setPositiveButton` function.

7. At the end, add `show()`, which creates and then displays the alert dialog.

```
        .show()
```

8. Here is the complete `showFinalScoreDialog()` method for reference.

```
/*
* Creates and shows an AlertDialog with the final score.
*/
private fun showFinalScoreDialog() {
   MaterialAlertDialogBuilder(requireContext())
        .setTitle(getString(R.string.congratulations))
        .setMessage(getString(R.string.you_scored, viewModel.score))
        .setCancelable(false)
        .setNegativeButton(getString(R.string.exit)) { _, _ ->
            exitGame()
        }
        .setPositiveButton(getString(R.string.play_again)) { _, _ ->
            restartGame()
        }
        .show()
}
```

9. Implement OnClickListener for Submit button

In this task, you use the `ViewModel` and the alert dialog you added to implement the game logic for the **Submit** button click listener.

Display the scrambled words

1. If you haven't already done so, in `GameFragment`, delete the code inside `onSubmitWord()` which gets called when the **Submit** button is tapped.
2. Add a check on the return value of `viewModel.nextWord()` method. If `true`, another word is available, so update the scrambled word on screen using `updateNextWordOnScreen()`. Otherwise the game is over, so display the alert dialog with the final score.

```
private fun onSubmitWord() {
    if (viewModel.nextWord()) {
        updateNextWordOnScreen()
    } else {
        showFinalScoreDialog()
    }
}
```

3. Run the app! Play through some words. Remember, you have not yet implemented the **Skip** button, so you can't skip the word.
4. Notice the text field is not updated, so the player has to manually delete the previous word. The final score in the alert dialog is always zero. You will fix these bugs in the coming steps.

Add a helper method to validate player word

1. In `GameViewModel`, add a new private method called `increaseScore()` with no parameters and no return value. Increase the `score` variable by *`SCORE_INCREASE`*.

```
private fun increaseScore() {
    _score += SCORE_INCREASE
}
```

2. In `GameViewModel`, add a helper method called `isUserWordCorrect()` which returns a `Boolean` and takes a `String`, the player's word, as a parameter.
3. In `isUserWordCorrect()` validate the player's word and increase the score if the guess is correct. This will update the final score in your alert dialog.

```
fun isUserWordCorrect(playerWord: String): Boolean {
    if (playerWord.equals(currentWord, true)) {
        increaseScore()
        return true
    }
    return false
}
```

Update the text field

Show errors in text field

For Material text fields, `TextInputLayout` comes with a built-in functionality to display error messages. For example in the following text field, the color of the label is changed, an error icon is displayed, an error message is displayed, and so on.

To show an error in the text field, you can set the error message either dynamically in code or statically in the layout file. Example to set and reset the error in code is shown below:

```
// Set error text
passwordLayout.error = getString(R.string.error)

// Clear error text
passwordLayout.error = null
```

In the starter code, you will find the helper method `setErrorTextField(error: Boolean)` is already defined to help you set and reset the error in the text field. Call this method with `true` or `false` as the input parameter based on whether you want an error to show up in the text field or not.

Code snippet in the starter code

```
private fun setErrorTextField(error: Boolean) {
   if (error) {
       binding.textField.isErrorEnabled = true
       binding.textField.error = getString(R.string.try_again)
   } else {
       binding.textField.isErrorEnabled = false
       binding.textInputEditText.text = null
   }
}
```

In this task, you implement the method `onSubmitWord()`. When a word is submitted, validate the user's guess by checking against the original word. If the word is correct, then go to the next word (or show the dialog if the game has ended). If the word is incorrect, show an error on the text field and stay on the current word.

1. In `GameFragment`, at the beginning of `onSubmitWord()`, create a `val` called `playerWord`. Store the player's word in it, by extracting it from the text field in the `binding` variable.

```
private fun onSubmitWord() {
    val playerWord = binding.textInputEditText.text.toString()
    ...
}
```

2. In `onSubmitWord()`, below the declaration of `playerWord`, validate the player's word. Add an `if` statement to check the player's word using the `isUserWordCorrect()` method, passing in the `playerWord`.
3. Inside the `if` block, reset the text field, call `setErrorTextField` passing in `false`.
4. Move the existing code inside the `if` block.

```
private fun onSubmitWord() {
    val playerWord = binding.textInputEditText.text.toString()

    if (viewModel.isUserWordCorrect(playerWord)) {
        setErrorTextField(false)
        if (viewModel.nextWord()) {
            updateNextWordOnScreen()
        } else {
            showFinalScoreDialog()
        }
    }
}
```

5. If the user word is incorrect, show an error message in the text field. Add an `else` block to the above `if` block, and call `setErrorTextField()` passing in `true`. Your completed `onSubmitWord()` method should look like this:

```
private fun onSubmitWord() {
    val playerWord = binding.textInputEditText.text.toString()

    if (viewModel.isUserWordCorrect(playerWord)) {
        setErrorTextField(false)
        if (viewModel.nextWord()) {
            updateNextWordOnScreen()
        } else {
            showFinalScoreDialog()
        }
    } else {
        setErrorTextField(true)
    }
}
```

6. Run your app. Play through some words. if the player's word is correct, the word is cleared on clicking the **Submit** button, otherwise a message saying "Try again!" is displayed. Notice that the **Skip** button is still not functional. You will add this implementation in the next task.

10. Implement the Skip button

In this task, you add the implementation for `onSkipWord()` which handles when the **Skip** button is clicked.

1. Similar to `onSubmitWord()`, add a condition in the `onSkipWord()` method. If `true`, display the word on screen and reset the text field. If `false` and there's no more words left in this round, show the alert dialog with the final score.

```
/*
* Skips the current word without changing the score.
*/
private fun onSkipWord() {
    if (viewModel.nextWord()) {
        setErrorTextField(false)
        updateNextWordOnScreen()
    } else {
        showFinalScoreDialog()
    }
}
```

2. Run your app. Play the game. Notice the **Skip** and **Submit** buttons are working as intended. Excellent!

11. Verify the ViewModel preserves data

For this task, add logging in `GameFragment` to observe that your app data is preserved in the `ViewModel`, during configuration changes. To access `currentWordCount` in `GameFragment`, you need to expose a read-only version using a backing property.

1. In `GameViewModel`, right click on the variable `currentWordCount`, select **Refactor** > **Rename...** . Prefix the new name with an underscore, `_currentWordCount`.
2. Add a backing field.

```
private var _currentWordCount = 0
val currentWordCount: Int
   get() = _currentWordCount
```

3. In `GameFragment` inside `onCreateView()`, above the return statement add another log to print the app data, word, score, and word count.

```
Log.d("GameFragment", "Word: ${viewModel.currentScrambledWord} " +
       "Score: ${viewModel.score} WordCount: ${viewModel.currentWordCount}")
```

4. In Android Studio open **Logcat**, filter on `GameFragment`. Run your app and play through some words. Change the orientation of your device. The fragment (UI controller) is destroyed and recreated. Observe the logs. Now you can see the score and word count increasing!

```
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: GameViewModel created!
com.example.android.unscramble D/GameFragment: Word: oimfnru Score: 0
WordCount: 1
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: Word: ofx Score: 80 WordCount:
5
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: Word: ofx Score: 80 WordCount:
5
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: Word: nvoiil Score: 160
WordCount: 9
com.example.android.unscramble D/GameFragment: GameFragment destroyed!
com.example.android.unscramble D/GameFragment: GameFragment created/re-
created!
com.example.android.unscramble D/GameFragment: Word: nvoiil Score: 160
WordCount: 9
```

Notice that the app data is preserved in the `ViewModel` during orientation changes. You will update score value and word count in the UI using `LiveData` and Data Binding in later codelabs.

12. Update game restart logic

1. Run the app again, play the game through all the words. In the **Congratulations!** alert dialog, click **PLAY AGAIN**. The app won't let you play again because the word count has now reached the value *`MAX_NO_OF_WORDS`*. You need to reset the word count to 0 to play the game again from the beginning.
2. To reset the app data, in `GameViewModel` add a method called `reinitializeData()`. Set the score and word count to `0`. Clear the word list and call `getNextWord()` method.

```
/*
* Re-initializes the game data to restart the game.
*/
fun reinitializeData() {
    _score = 0
    _currentWordCount = 0
    wordsList.clear()
    getNextWord()
}
```

3. In `GameFragment` at the top the method `restartGame()`, make a call to the newly created method, `reinitializeData()`.

```
private fun restartGame() {
    viewModel.reinitializeData()
    setErrorTextField(false)
    updateNextWordOnScreen()
}
```

4. Run your app again. Play the game. When you reach the congratulations dialog, click on **Play Again**. Now you should be able to successfully play the game again!

This is what your final app should look like. The game shows ten random scrambled words for the player to unscramble. You can either **Skip** the word or guess a word and tap **Submit**. If you guess correctly, the score increases. An incorrect guess shows an error state in the text field. With each new word, the word count also increases.

Note that the score and word count displayed on screen do not update yet. But the information is still being stored in the view model and preserved during configuration changes like device rotation. You will update the score and word count on screen in later codelabs.

Congratulations! You have created your first `ViewModel` and you saved the data!

13. Solution code

NOTE: Make sure to always include the package name of your app in all the Kotlin source files.

GameFragment.kt

```
import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import com.example.android.unscramble.R
import com.example.android.unscramble.databinding.GameFragmentBinding
import com.google.android.material.dialog.MaterialAlertDialogBuilder

/**
 * Fragment where the game is played, contains the game logic.
 */
class GameFragment : Fragment() {

    private val viewModel: GameViewModel by viewModels()

    // Binding object instance with access to the views in the
game_fragment.xml layout
    private lateinit var binding: GameFragmentBinding

    // Create a ViewModel the first time the fragment is created.
    // If the fragment is re-created, it receives the same GameViewModel
instance created by the
    // first fragment

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        // Inflate the layout XML file and return a binding object instance
        binding = GameFragmentBinding.inflate(inflater, container, false)
        Log.d("GameFragment", "GameFragment created/re-created!")
        Log.d("GameFragment", "Word: ${viewModel.currentScrambledWord} " +
                "Score: ${viewModel.score} WordCount:
${viewModel.currentWordCount}")
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // Setup a click listener for the Submit and Skip buttons.
        binding.submit.setOnClickListener { onSubmitWord() }
        binding.skip.setOnClickListener { onSkipWord() }
```

```
        // Update the UI
        updateNextWordOnScreen()
        binding.score.text = getString(R.string.score, 0)
        binding.wordCount.text = getString(
            R.string.word_count, 0, MAX_NO_OF_WORDS)
    }

    /*
    * Checks the user's word, and updates the score accordingly.
    * Displays the next scrambled word.
    * After the last word, the user is shown a Dialog with the final score.
    */
    private fun onSubmitWord() {
        val playerWord = binding.textInputEditText.text.toString()

        if (viewModel.isUserWordCorrect(playerWord)) {
            setErrorTextField(false)
            if (viewModel.nextWord()) {
                updateNextWordOnScreen()
            } else {
                showFinalScoreDialog()
            }
        } else {
            setErrorTextField(true)
        }
    }

    /*
    * Skips the current word without changing the score.
    */
    private fun onSkipWord() {
        if (viewModel.nextWord()) {
            setErrorTextField(false)
            updateNextWordOnScreen()
        } else {
            showFinalScoreDialog()
        }
    }

    /*
     * Gets a random word for the list of words and shuffles the letters in
it.
     */
    private fun getNextScrambledWord(): String {
        val tempWord = allWordsList.random().toCharArray()
        tempWord.shuffle()
        return String(tempWord)
    }

    /*
    * Creates and shows an AlertDialog with the final score.
    */
    private fun showFinalScoreDialog() {
        MaterialAlertDialogBuilder(requireContext())
            .setTitle(getString(R.string.congratulations))
            .setMessage(getString(R.string.you_scored, viewModel.score))
            .setCancelable(false)
```

```
            .setNegativeButton(getString(R.string.exit)) { _, _ ->
                exitGame()
            }
            .setPositiveButton(getString(R.string.play_again)) { _, _ ->
                restartGame()
            }
            .show()
    }

    /*
     * Re-initializes the data in the ViewModel and updates the views with
the new data, to
     * restart the game.
     */
    private fun restartGame() {
        viewModel.reinitializeData()
        setErrorTextField(false)
        updateNextWordOnScreen()
    }

    /*
     * Exits the game.
     */
    private fun exitGame() {
        activity?.finish()
    }

    override fun onDetach() {
        super.onDetach()
        Log.d("GameFragment", "GameFragment destroyed!")
    }

    /*
    * Sets and resets the text field error status.
    */
    private fun setErrorTextField(error: Boolean) {
        if (error) {
            binding.textField.isErrorEnabled = true
            binding.textField.error = getString(R.string.try_again)
        } else {
            binding.textField.isErrorEnabled = false
            binding.textInputEditText.text = null
        }
    }

    /*
     * Displays the next scrambled word on screen.
     */
    private fun updateNextWordOnScreen() {
        binding.textViewUnscrambledWord.text = viewModel.currentScrambledWord
    }
}
```

GameViewModel.kt

```
import android.util.Log
import androidx.lifecycle.ViewModel

/**
 * ViewModel containing the app data and methods to process the data
 */
class GameViewModel : ViewModel(){
    private var _score = 0
    val score: Int
        get() = _score

    private var _currentWordCount = 0
    val currentWordCount: Int
        get() = _currentWordCount

    private lateinit var _currentScrambledWord: String
    val currentScrambledWord: String
        get() = _currentScrambledWord

    // List of words used in the game
    private var wordsList: MutableList<String> = mutableListOf()
    private lateinit var currentWord: String

    init {
        Log.d("GameFragment", "GameViewModel created!")
        getNextWord()
    }

    override fun onCleared() {
        super.onCleared()
        Log.d("GameFragment", "GameViewModel destroyed!")
    }

    /*
    * Updates currentWord and currentScrambledWord with the next word.
    */
    private fun getNextWord() {
        currentWord = allWordsList.random()
        val tempWord = currentWord.toCharArray()
        tempWord.shuffle()

        while (String(tempWord).equals(currentWord, false)) {
            tempWord.shuffle()
        }
        if (wordsList.contains(currentWord)) {
            getNextWord()
        } else {
            _currentScrambledWord = String(tempWord)
            ++_currentWordCount
            wordsList.add(currentWord)
        }
    }

    /*
    * Re-initializes the game data to restart the game.
    */
    fun reinitializeData() {
```

```
        _score = 0
        _currentWordCount = 0
        wordsList.clear()
        getNextWord()
    }


    /*
    * Increases the game score if the player's word is correct.
    */
    private fun increaseScore() {
        _score += SCORE_INCREASE
    }

    /*
    * Returns true if the player word is correct.
    * Increases the score accordingly.
    */
    fun isUserWordCorrect(playerWord: String): Boolean {
        if (playerWord.equals(currentWord, true)) {
            increaseScore()
            return true
        }
        return false
    }

    /*
    * Returns true if the current word count is less than MAX_NO_OF_WORDS
    */
    fun nextWord(): Boolean {
        return if (_currentWordCount < MAX_NO_OF_WORDS) {
            getNextWord()
            true
        } else false
    }
}
```

14. Summary

- The Android app architecture guidelines recommend separating classes that have different responsibilities and driving the UI from a model.
- A UI controller is a UI-based class like `Activity` or `Fragment`. UI controllers should only contain logic that handles UI and operating system interactions; they shouldn't be the source of data to be displayed in the UI. Put that data and any related logic in a `ViewModel`.
- The `ViewModel` class stores and manages UI-related data. The `ViewModel` class allows data to survive configuration changes such as screen rotations.
- `ViewModel` is one of the recommended Android Architecture Components.

15. Learn more

- ViewModel Overview

Use LiveData with ViewModel

1. Before you begin

You have learned in the previous codelabs, how to use a `ViewModel` to store the app data. `ViewModel` allows the app's data to survive configuration changes. In this codelab, you'll learn how to integrate `LiveData` with the data in the `ViewModel`.

The `LiveData` class is also part of the Android Architecture Components and is a data holder class that can be observed.

Prerequisites

- How to download source code from GitHub and open it in Android studio.
- How to create and run a basic Android app in Kotlin, using activities and fragments.
- How the activity and fragment life cycles work.
- How to retain UI data through device-configuration changes using a `ViewModel`.
- How to write lambda expressions.

What you'll learn

- How to use `LiveData` and `MutableLiveData` in your app.
- How to encapsulate the data stored in a `ViewModel` with `LiveData`.

- How to add observer methods to observe changes in the `LiveData`.
- How to write binding expressions in a layout file.

What you'll build

- Use `LiveData` for the app's data (word, word count and the score) in the Unscramble app.
- Add observer methods that get notified when the data changes, update the scrambled word text view automatically.
- Write binding expressions in the layout file, which are triggered when the underlying `LiveData` is changed. The score, word count and the scrambled word text views are updated automatically.

What you need

- A computer with Android Studio installed.
- Solution code from the previous codelab (Unscramble app with `ViewModel`).

Download the starter code for this codelab

This codelab uses the Unscramble app that you built in the previous codelab (Store data in ViewModel) as the starter code.

Starter Code URL: https://github.com/google-developer-training/android-basics-kotlin-unscramble-app/tree/starter

Add the solution code from the previous codelab (Store data in ViewModel) to the above starter branch, and use it as starter code for this codelab.

2. Starter app overview

This codelab uses the Unscramble solution code that you are familiar with from the previous codelab. The app displays a scrambled word for the player to unscramble it. The player can try any number of times to guess the correct word. The app data such as the current word, player's score and word count are saved in the `ViewModel`. However, the app's UI does not reflect the new score and word count values. In this codelab, you will implement the missing features using `LiveData`.

3. What is Livedata

`LiveData` is an observable data holder class that is lifecycle-aware.

Some characteristics of `LiveData`:

- `LiveData` holds data; `LiveData` is a wrapper that can be used with any type of data.
- `LiveData` is observable, which means that an observer is notified when the data held by the `LiveData` object changes.
- `LiveData` is lifecycle-aware. When you attach an observer to the `LiveData`, the observer is associated with a `LifecycleOwner` (usually an activity or fragment). The `LiveData` only updates observers that are in an active lifecycle state such as `STARTED` or `RESUMED`. You can read more about `LiveData` and observation here.

UI updation in the starter code

In the starter code the `updateNextWordOnScreen()` method is called explicitly, every time you want to display a new scrambled word in the UI. You call this method during game initialization, and when players press the **Submit** or **Skip** button. This method is called from the methods `onViewCreated()`, `restartGame()`, `onSkipWord()`, and `onSubmitWord()`. With `Livedata`, you will not have to call this method from multiple places to update the UI. You will do it only once in the observer.

4. Add LiveData to the current scrambled word

In this task, you will learn how to wrap any data with `LiveData`, by converting the current word in the `GameViewModel` to `LiveData`. In a later task, you will add an observer to these `LiveData` objects and learn how to observe the `LiveData`.

MutableLiveData

`MutableLiveData` is the mutable version of the `LiveData`, that is, the value of the data stored within it can be changed.

1. In `GameViewModel`, change the type of the variable `_currentScrambledWord` to `MutableLiveData<String>`. `LiveData` and `MutableLiveData` are generic classes, so you need to specify the type of data that they hold.
2. Change the variable type of `_currentScrambledWord` to `val` because the value of the `LiveData`/`MutableLiveData` object will remain the same, and only the data stored within the object will change.

```
private val _currentScrambledWord = MutableLiveData<String>()
```

3. Change the backing field, `currentScrambledWord` type to `LiveData<String>`, because it is immutable. Android Studio will show some errors which you will fix in the next steps.

```
val currentScrambledWord: LiveData<String>
    get() = _currentScrambledWord
```

4. To access the data within a `LiveData` object, use the `value` property. In `GameViewModel` inside the `getNextWord()` method, within the `else` block, change the reference of **`_currentScrambledWord`** to **`_currentScrambledWord.value`**.

```
private fun getNextWord() {
 ...
    } else {
        _currentScrambledWord.value = String(tempWord)
        ...
    }
}
```

5. Attach observer to the LiveData object

In this task you set up an observer in the app component, `GameFragment`. The observer you will add observes the changes to the app's data `currentScrambledWord`. `LiveData` is lifecycle-aware, meaning it only updates observers that are in an active lifecycle state. So the observer in the `GameFragment` will only be notified when the `GameFragment` is in `STARTED` or `RESUMED` states.

1. In `GameFragment`, delete the method `updateNextWordOnScreen()` and all the calls to it. You do not require this method, as you will be attaching an observer to the `LiveData`.
2. In `onSubmitWord()`, modify the empty `if-else` block as follows. The complete method should look like this.

```
private fun onSubmitWord() {
    val playerWord = binding.textInputEditText.text.toString()

    if (viewModel.isUserWordCorrect(playerWord)) {
        setErrorTextField(false)
        if (!viewModel.nextWord()) {
            showFinalScoreDialog()
        }
    } else {
        setErrorTextField(true)
    }
}
```

3. Attach an observer for `currentScrambledWord LiveData`. In `GameFragment` at the end of the callback `onViewCreated()`, call the `observe()` method on `currentScrambledWord`.

```
// Observe the currentScrambledWord LiveData.
viewModel.currentScrambledWord.observe()
```

Android Studio will display an error about missing parameters. You will fix the error in the next step.

4. Pass *`viewLifecycleOwner`* as the first parameter to the `observe()` method. The *`viewLifecycleOwner`* represents the Fragment's View lifecycle. This parameter helps the `LiveData` to be aware of the `GameFragment` lifecycle and notify the observer only when the `GameFragment` is in active states (`STARTED` or `RESUMED`).
5. Add a lambda as a second parameter with `newWord` as a function parameter. The `newWord` will contain the new scrambled word value.

```
// Observe the scrambledCharArray LiveData, passing in the LifecycleOwner and
the observer.
viewModel.currentScrambledWord.observe(viewLifecycleOwner,
    { newWord ->
    })
```

A lambda expression is an anonymous function that isn't declared, but is passed immediately as an expression. A lambda expression is always surrounded by curly braces { }.

6. In the function body of the lambda expression, assign `newWord` to the scrambled word text view.

```
// Observe the scrambledCharArray LiveData, passing in the LifecycleOwner and
the observer.
viewModel.currentScrambledWord.observe(viewLifecycleOwner,
    { newWord ->
        binding.textViewUnscrambledWord.text = newWord
    })
```

7. Compile and run app. Your game app should work exactly as before, but now the scrambled word text view is automatically updated in the `LiveData` observer, not in the `updateNextWordOnScreen()` method.

6. Attach observer to score and word count

As in the previous task, in this task you will add `LiveData` to the other data in the app, score and word count, so that the UI is updated with correct values of the score and word count during the game.

Step 1: Wrap score and wordcount with LiveData

1. In `GameViewModel`, change the type of the `_score` and `_currentWordCount` class variables to `val`.

2. Change the data type of the variables `_score` and `_currentWordCount` to `MutableLiveData` and initialize them to `0`.
3. Change backing fields type to `LiveData<Int>`.

```
private val _score = MutableLiveData(0)
val score: LiveData<Int>
   get() = _score

private val _currentWordCount = MutableLiveData(0)
val currentWordCount: LiveData<Int>
   get() = _currentWordCount
```

4. In `GameViewModel` at the beginning of the `reinitializeData()` method, change the reference of `_score` and `_currentWordCount` to `_score.value` and `_currentWordCount.value` respectively.

```
fun reinitializeData() {
   _score.value = 0
   _currentWordCount.value = 0
   wordsList.clear()
   getNextWord()
}
```

5. In the `GameViewModel`, inside the `nextWord()` method, change the reference of `_currentWordCount` to `_currentWordCount.value!!`.

```
fun nextWord(): Boolean {
    return if (_currentWordCount.value!! < MAX_NO_OF_WORDS) {
           getNextWord()
           true
       } else false
   }
```

6. In `GameViewModel`, inside the `increaseScore()` and `getNextWord()` methods, change the reference of `_score` and `_currentWordCount` to `_score.value` and `_currentWordCount.value` respectively. Android Studio will show you an error because `_score` is no longer an integer, it's `LiveData`, you will fix it in the next steps.
7. Use the `plus()` Kotlin function to increase the `_score` value, which performs the addition with null-safety.

```
private fun increaseScore() {
    _score.value = (_score.value)?.plus(SCORE_INCREASE)
}
```

8. Similarly use `inc()` Kotlin function to increment the value by one with null-safety.

```
private fun getNextWord() {
   ...
    } else {
        _currentScrambledWord.value = String(tempWord)
        _currentWordCount.value = (_currentWordCount.value)?.inc()
```

```
            wordsList.add(currentWord)
        }
    }
```

9. In `GameFragment`, access the value of `score` using the `value` property. Inside the `showFinalScoreDialog()` method, change `viewModel.score` to `viewModel.score.value`.

```
private fun showFinalScoreDialog() {
    MaterialAlertDialogBuilder(requireContext())
        .setTitle(getString(R.string.congratulations))
        .setMessage(getString(R.string.you_scored, viewModel.score.value))
        ...
        .show()
}
```

Step 2: Attach observers to score and word count

In the app, the score and the word count are not updated. You will update them in this task using `LiveData` observers.

1. In `GameFragment` inside the `onViewCreated()` method, delete the code that updates the score and word count text views.

Remove:

```
binding.score.text = getString(R.string.score, 0)
binding.wordCount.text = getString(R.string.word_count, 0, MAX_NO_OF_WORDS)
```

2. In the `GameFragment` at the end of `onViewCreated()` method, attach observer for `score`. Pass in the `viewLifecycleOwner` as the first parameter to the observer and a lambda expression for the second parameter. Inside the lambda expression, pass the new score as a parameter and inside the function body, set the new score to the text view.

```
viewModel.score.observe(viewLifecycleOwner,
    { newScore ->
        binding.score.text = getString(R.string.score, newScore)
    })
```

3. At the end of the `onViewCreated()` method, attach an observer for the `currentWordCount LiveData`. Pass in the `viewLifecycleOwner` as the first parameter to the observer and a lambda expression for the second parameter. Inside the lambda expression, pass the new word count as a parameter and in the function body, set the new word count along with the `MAX_NO_OF_WORDS` to the text view.

```
viewModel.currentWordCount.observe(viewLifecycleOwner,
    { newWordCount ->
        binding.wordCount.text =
```

```
            getString(R.string.word_count, newWordCount, MAX_NO_OF_WORDS)
    })
```

The new observers will be triggered when the value of score and word count change inside the `ViewModel`, during the lifetime of the lifecycle owner, that is, the `GameFragment`.

4. Run your app to see the magic. Play the game through some words. Score and word count are also updated correctly on the screen. Observe that you are not updating these text views based on some conditions in the code. The `score` and `currentWordCount` are `LiveData` and the corresponding observers are automatically called when the underlying value changes.

7. Use LiveData with data binding

In the previous tasks, your app listens to the data changes in the code. Similarly, apps can listen to the data changes from the layout. With Data Binding, when an observable `LiveData` value changes, the UI elements in the layout it's bound to are also notified, and the UI can be updated from within the layout.

Concept: Data binding

In the previous codelabs you have seen View Binding, which is a one-way binding. You can bind views to code but not vice versa.

Refresher for View binding:

View binding is a feature that allows you to more easily access views in code. It generates a binding class for each XML layout file. An instance of a binding class contains direct references to all views that have an ID in the corresponding layout. For example, the Unscramble app currently uses view binding, so the views can be referenced in the code using the generated binding class.

Example:

```
binding.textViewUnscrambledWord.text = newWord
binding.score.text = getString(R.string.score, newScore)
binding.wordCount.text =
                  getString(R.string.word_count, newWordCount,
MAX_NO_OF_WORDS)
```

Using view binding you can't reference the app data in the views (layout files). This can be accomplished using Data binding.

Data Binding

Data Binding Library is also a part of the Android Jetpack library. Data binding binds the UI components in your layouts to data sources in your app using a declarative format, which you will learn later in the codelab.

In simpler terms Data binding is binding data (from code) to views + view binding (binding views to code):

Example using view binding in UI controller

```
binding.textViewUnscrambledWord.text = viewModel.currentScrambledWord
```

Example using data binding in layout file

```
android:text="@{gameViewModel.currentScrambledWord}"
```

The above example shows how to use the Data Binding Library to assign app data to the views/widget directly in the layout file. Note the use of `@{}` syntax in the assignment expression.

The main advantage of using data binding is, it lets you remove many UI framework calls in your activities, making them simpler and easier to maintain. This can also improve your app's performance and help prevent memory leaks and null pointer exceptions.

Step 1: Change view binding to data binding

1. In the `build.gradle(Module)` file, enable the `dataBinding` property under the `buildFeatures` section.

Replace

```
buildFeatures {
   viewBinding = true
}
```

with

```
buildFeatures {
   dataBinding = true
}
```

Do a gradle sync when prompted by Android Studio.

2. To use data binding in any Kotlin project, you should apply the `kotlin-kapt` plugin. This step is already done for you in the `build.gradle(Module)` file.

```
plugins {
   id 'com.android.application'
   id 'kotlin-android'
   id 'kotlin-kapt'
}
```

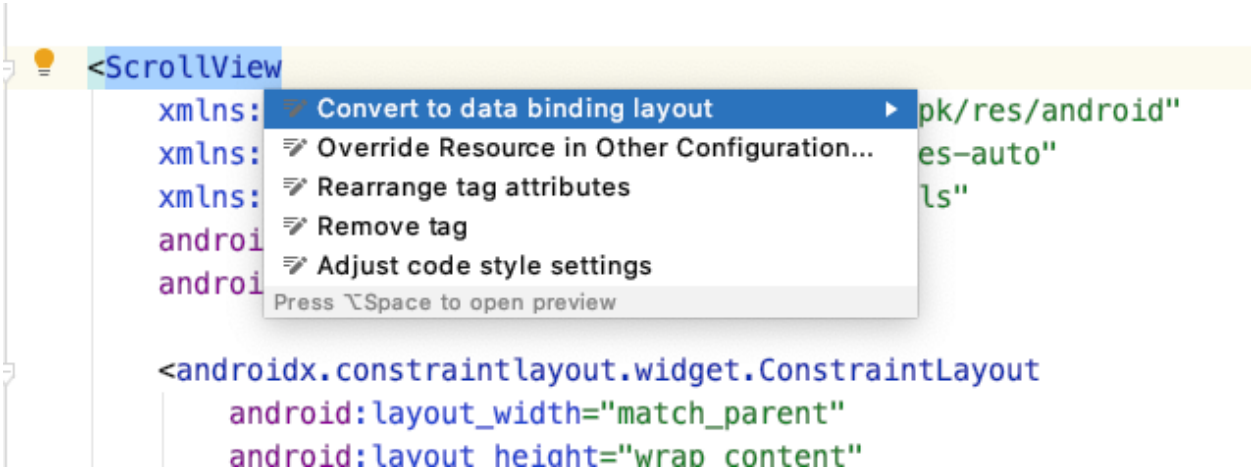
Above steps auto generates a binding class for every layout XML file in the app. If the layout file name is `activity_main.xml` then your autogen class will be called `ActivityMainBinding`.

Step 2: Convert layout file to data binding layout

Data binding layout files are slightly different and start with a root tag of `<layout>` followed by an optional `<data>` element and a `view` root element. This view element is what your root would be in a non-binding layout file.

1. Open `game_fragment.xml`, select **code** tab.

2. To convert the layout to a Data Binding layout, wrap the root element in a `<layout>` tag. You'll also have to move the namespace definitions (the attributes that start with `xmlns:`) to the new root element. Add `<data></data>` tags inside `<layout>` tag above the root element. Android Studio offers a handy way to do this automatically: Right-click the root element (`ScrollView`), select **Show Context Actions** > **Convert to data binding layout**.



3. Your layout should look something like this:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>

    </data>

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

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

4. In `GameFragment`, at the beginning of the `onCreateView()` method, change the instantiation of the `binding` variable to use data binding.

Replace

```
binding = GameFragmentBinding.inflate(inflater, container, false)
```

with

```
binding = DataBindingUtil.inflate(inflater, R.layout.game_fragment,
container, false)
```

5. Compile the code; you should be able to compile without any issues. Your app now uses data binding and the views in the layout can access the app data.

8. Add data binding variables

In this task you will add properties in the layout file to access the app data from the `viewModel`. You will initialize the layout variables in the code.

1. In `game_fragment.xml`, inside the `<data>` tag add a child tag called `<variable>`, declare a property called `gameViewModel` and of the type `GameViewModel`. You will use this to bind the data in `ViewModel` to the layout.

```
<data>
   <variable
       name="gameViewModel"
       type="com.example.android.unscramble.ui.game.GameViewModel" />
</data>
```

Notice the type of `gameViewModel` contains the package name. Make sure this package name matches with the package name in your app.

2. Below the `gameViewModel` declaration, add another variable inside the `<data>` tag of type `Integer`, and name it `maxNoOfWords`. You will use this to bind to the variable in ViewModel to store the number of words per game.

```
<data>
   ...
   <variable
       name="maxNoOfWords"
       type="int" />
</data>
```

3. In `GameFragment` at the beginning of the `onViewCreated()` method, initialize the layout variables `gameViewModel` and `maxNoOfWords`.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
   super.onViewCreated(view, savedInstanceState)

   binding.gameViewModel = viewModel

   binding.maxNoOfWords = MAX_NO_OF_WORDS
...
}
```

4. The `LiveData` is lifecycle-aware observable, so you have to pass the lifecycle owner to the layout. In the `GameFragment`, inside the `onViewCreated()` method, below the initialization of the binding variables, add the following code.

```
// Specify the fragment view as the lifecycle owner of the binding.
// This is used so that the binding can observe LiveData updates
binding.lifecycleOwner = viewLifecycleOwner
```

Recall that you implemented a similar functionality when implementing `LiveData` observers. You passed `viewLifecycleOwner` as one of the parameters to the `LiveData` observers.

9. Use binding expressions

Binding expressions are written within the layout in the attribute properties (such as `android:text`) referencing the layout properties. Layout properties are declared at the top of the data binding layout file, via the `<variable>` tag. When any of the dependent variables change, the 'DB Library' will run your binding expressions (and thus updates the views). This change-detection is a great optimization which you get for free, when you use a Data Binding Library.

Syntax for binding expressions

Binding expressions start with an `@` symbol and are wrapped inside curly braces `{}`. In the following example, the `TextView` text is set to the `firstName` property of the `user` variable:

Example:

```
<TextView android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@{user.firstName}" />
```

Step 1: Add binding expression to the current word

In this step, you bind the current word text view to the `LiveData` object in the `ViewModel`.

1. In `game_fragment.xml`, add a `text` attribute to the `textView_unscrambled_word` text view. Use the new layout variable, `gameViewModel` and assign `@{gameViewModel.currentScrambledWord}` to the `text` attribute.

```
<TextView
   android:id="@+id/textView_unscrambled_word"
   ...
   android:text="@{gameViewModel.currentScrambledWord}"
   .../>
```

2. In `GameFragment`, remove the `LiveData` observer code for `currentScrambledWord`: You don't need the observer code in fragment any more. The layout receives the updates of the changes to the `LiveData` directly.

Remove:

```
viewModel.currentScrambledWord.observe(viewLifecycleOwner,
   { newWord ->
       binding.textViewUnscrambledWord.text = newWord
   })
```

3. Run your app, your app should work as before. But now the scrambled word text view uses the binding expressions to update the UI, not the `LiveData` observers.

Step 2: Add binding expression to the score and the word count

Resources in data binding expressions

A data binding expression can reference app resources with the following syntax.

Example:

```
android:padding="@{@dimen/largePadding}"
```

In the above example, the `padding` attribute is assigned a value of `largePadding` from the `dimen.xml` resource file.

You can also pass layout properties as resource parameters.

Example:

```
android:text="@{@string/example_resource(user.lastName)}"

strings.xml

<string name="example_resource">Last Name: %s</string>
```

In the above example, `example_resource` is a string resource with `%s` placeholder. You are passing `user.lastName` as a resource parameter in the binding expression, where `user` is a layout variable.

In this step you will add binding expressions to the score and word count text views, passing in the resource parameters. This step is similar to what you did for `textView_unscrambled_word` above.

1. In `game_fragment.xml`, update the `text` attribute for `word_count` text view with the following binding expression. Use `word_count` string resource and pass in `gameViewModel.currentWordCount`, and `maxNoOfWords` as resource parameters.

```
<TextView
   android:id="@+id/word_count"
   ...
   android:text="@{@string/word_count(gameViewModel.currentWordCount,
maxNoOfWords)}"
   .../>
```

2. Update the `text` attribute for `score` text view with the following binding expression. Use `score` string resource and pass in `gameViewModel.score` as a resource parameter.

```
<TextView
   android:id="@+id/score"
   ...
   android:text="@{@string/score(gameViewModel.score)}"
   ... />
```

3. Remove `LiveData` observers from the `GameFragment`. You don't need them any longer, binding expressions update the UI when the corresponding `LiveData` changes.

Remove:

```
viewModel.score.observe(viewLifecycleOwner,
   { newScore ->
       binding.score.text = getString(R.string.score, newScore)
   })

viewModel.currentWordCount.observe(viewLifecycleOwner,
   { newWordCount ->
       binding.wordCount.text =
           getString(R.string.word_count, newWordCount, MAX_NO_OF_WORDS)
   })
```

4. Run your app and play through some words. Now your code uses `LiveData` and binding expressions to update the UI.

Congratulations! You have learned how to use `LiveData` with `LiveData` observers and `LiveData` with binding expressions.

10. Test Unscramble app with Talkback enabled

As you've been learning throughout this course, you want to build apps that are accessible to as many users as possible. Some users may use Talkback to access and navigate your app. TalkBack is the Google screen reader included on Android devices. TalkBack gives you spoken feedback so that you can use your device without looking at the screen.

With Talkback enabled, ensure that a player can play the game.

1. Enable Talkback on your device by following these instructions.
2. Return to the Unscramble app.
3. Explore your app with Talkback using these instructions. Swipe right to navigate through screen elements in sequence, and swipe left to go in the opposite direction. Double-tap anywhere to select. Verify that you can reach all elements of your app with swipe gestures.
4. Ensure that a Talkback user is able to navigate to each item on the screen.
5. Observe that Talkback tries to read the scrambled word as a word. This may be confusing to the player since this is not a real word.
6. A better user experience would be to have Talkback read aloud the individual characters of the scrambled word. Within the `GameViewModel`, convert the scrambled word `String` to a `Spannable` string. A spannable string is a string with some extra information attached to it. In this case, we want to associate the string with a `TtsSpan` of `TYPE_VERBATIM`, so that the text-to-speech engine reads aloud the scrambled word verbatim, character by character.
7. In `GameViewModel`, use the following code to modify how the `currentScrambledWord` variable is declared:

```
val currentScrambledWord: LiveData<Spannable> =
Transformations.map(_currentScrambledWord) {
    if (it == null) {
        SpannableString("")
    } else {
        val scrambledWord = it.toString()
        val spannable: Spannable = SpannableString(scrambledWord)
        spannable.setSpan(
            TtsSpan.VerbatimBuilder(scrambledWord).build(),
            0,
            scrambledWord.length,
            Spannable.SPAN_INCLUSIVE_INCLUSIVE
        )
        spannable
    }
}
```

This variable is now a `LiveData<Spannable>` instead of `LiveData<String>`. You don't have to worry about understanding all the details of how this works, but the implementation uses a

`LiveData` transformation to convert the current scrambled word `String` into a Spannable string that can be handled appropriately by the accessibility service. In the next codelab, you will learn more about `LiveData` transformations, which allow you to return a different `LiveData` instance based on the value of corresponding `LiveData`.

8. Run the Unscramble app, explore your app with Talkback. TalkBack should read out the individual characters of the scrambled word now.

For more information on how to make your app more accessible, check out these principles.

Note: The accessibility service included on your device may vary on your device depending on the device manufacturer, you may experience different behavior. If the individual characters are not being read aloud for the scrambled word, try running your app on the emulator in Android Studio. You will need to install the Android Accessibility Suite app on the emulator in order to enable Talkback.

11. Delete unused code

It is a good practice to delete the dead, unused, unwanted code for the solution code. This makes the code easy to maintain, which also makes it easier for new teammates to understand the code better.

1. In `GameFragment`, delete `getNextScrambledWord()` and `onDetach()` methods.
2. In `GameViewModel` delete `onCleared()` method.
3. Delete any unused imports, at the top of the source files. They will be greyed out.

You don't need the log statements any more, you can delete them from the code if you prefer.

1. [Optional] Delete the `Log` statements in the source files(`GameFragment.kt` and `GameViewModel.kt`) you added in the previous codelab, to understand the `ViewModel` lifecycle.

12. Solution code

The solution code for this codelab is in the project shown below.

Solution Code URL: https://github.com/google-developer-training/android-basics-kotlin-unscramble-app/tree/main

1. Navigate to the provided GitHub repository page for the project.
2. Verify that the branch name matches the branch name specified in the codelab. For example, in the following screenshot the branch name is **main**.

Shared ViewModel Across Fragments

1. Before you begin

You have learned how to use activities, fragments, intents, data binding, navigation components, and the basics of architecture components. In this codelab, you will put everything together and work on an advanced sample, a cupcake ordering app.

You will learn how to use a shared `ViewModel` to share data between the fragments of the same activity and new concepts like `LiveData` transformations.

Prerequisites

- Comfortable with reading and understanding Android layouts in XML
- Familiar with the basics of the Jetpack Navigation Component
- Able to create a navigation graph with fragment destinations in an app
- Have previously used fragments within an activity
- Can create a `ViewModel` to store app data
- Can use data binding with `LiveData` to keep the UI up-to-date with the app data in the `ViewModel`

What you'll learn

- How to implement recommended app architecture practices within a more advanced use case
- How to use a shared `ViewModel` across fragments in an activity

3. In the **Import Project** dialog, navigate to where the unzipped project folder is located (likely in your **Downloads** folder).
4. Double-click on that project folder.
5. Wait for Android Studio to open the project.
6. Click the **Run** button  to build and run the app. Make sure it builds as expected.
7. Browse the project files in the **Project** tool window to see how the app is set-up.

Starter code walk through

1. Open the downloaded project in Android Studio. The folder name of the project is `android-basics-kotlin-cupcake-app-starter`. Then run the app.
2. Browse the files to understand the starter code. For layout files, you can use the **Split** option in the top right corner to see a preview of the layout and the XML at the same time.
3. When you compile and run the app, you'll notice the app is incomplete. The buttons don't do much (except for displaying a `Toast` message) and you can't navigate to the other fragments.

Here's a walkthrough of important files in the project.

MainActivity:

The `MainActivity` has similar code to the default generated code, which sets the activity's content view as `activity_main.xml`. This code uses a parameterized constructor `AppCompatActivity(@LayoutRes int contentLayoutId)` which takes in a layout that will be inflated as part of `super.onCreate(savedInstanceState)`.

Code in the `MainActivity` class

```
class MainActivity : AppCompatActivity(R.layout.activity_main)
```

is same as the following code using the default `AppCompatActivity` constructor:

```
class MainActivity : AppCompatActivity() {

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

Layouts (res/layout folder):

The `layout` resource folder contains activity and fragment layout files. These are simple layout files, and the XML is familiar from the previous codelabs.

- `fragment_start.xml` is the first screen shown in the app. It has a cupcake image and three buttons to choose the number of cupcakes to order: one cupcake, six cupcakes, and twelve cupcakes.
- `fragment_flavor.xml` shows a list of cupcake flavors as radio button options with a **Next** button.
- `fragment_pickup.xml` provides an option to select pickup day and a **Next** button to go to the summary screen.
- `fragment_summary.xml` displays a summary of the order details such as quantity, flavor and a button to send the order to another app.

Fragment classes:

- `StartFragment.kt` is the first screen shown in the app. This class contains the view binding code and a click handler for the three buttons.
- `FlavorFragment.kt` , `PickupFragment.kt`, and `SummaryFragment.kt` classes contain mostly boilerplate code and a click handler for the **Next** or **Send Order to Another App** button, which show a toast message.

Resources (res folder):

- `drawable` folder contains the cupcake asset for the first screen, as well as the launcher icon files.
- `navigation/nav_graph.xml` contains four fragment destinations (`startFragment`, `flavorFragment`, `pickupFragment`, and `summaryFragment`) without *Actions*, which you will define later in the codelab.
- `values` folder contains the colors, dimensions, strings, styles, and themes used for customizing the app theme. You should be familiar with these resource types from previous codelabs.

3. Complete the Navigation Graph

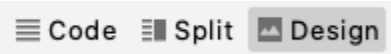
In this task, you'll connect the screens of the **Cupcake** app together and finish implementing proper navigation within the app.

Do you remember what we need to use the Navigation component? Follow this guide for a refresher on how to set up your project and app to:

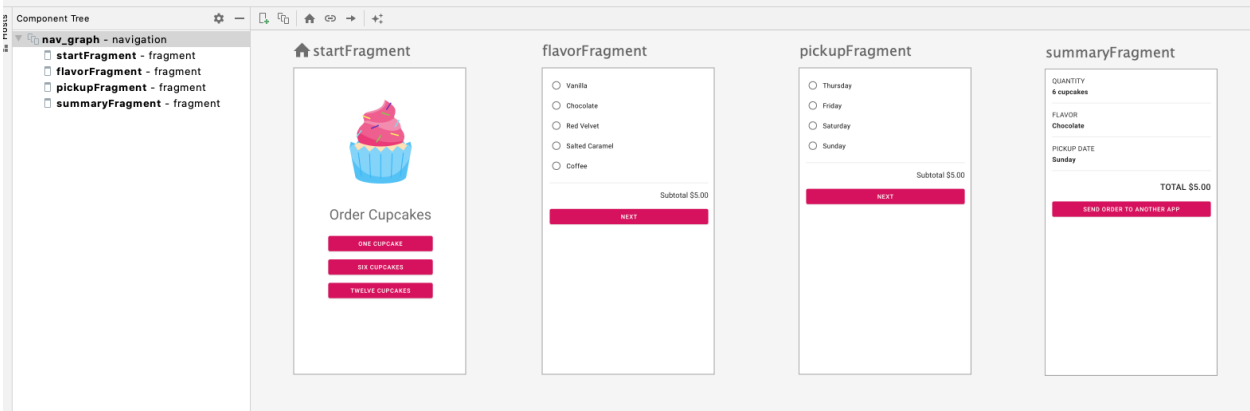
- Include the Jetpack Navigation library
- Add a `NavHost` to the activity
- Create a navigation graph
- Add fragment destinations to the navigation graph

Connect destinations in navigation graph

1. In Android Studio, in the **Project** window, open **res > navigation > nav_graph.xml** file. Switch to the **Design** tab, if it's not already selected.

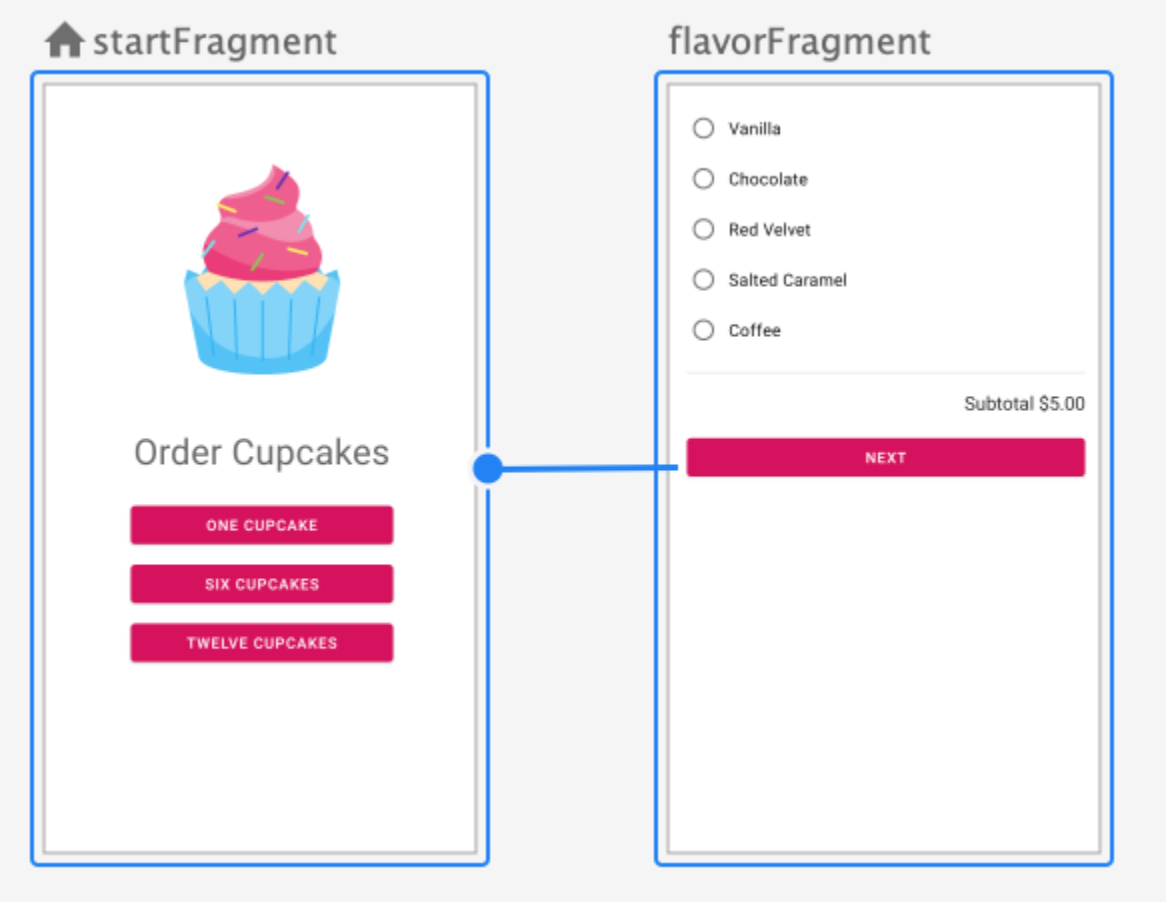


2. This opens the **Navigation Editor** to visualize the navigation graph in your app. You should see the four fragments that already exist in the app.

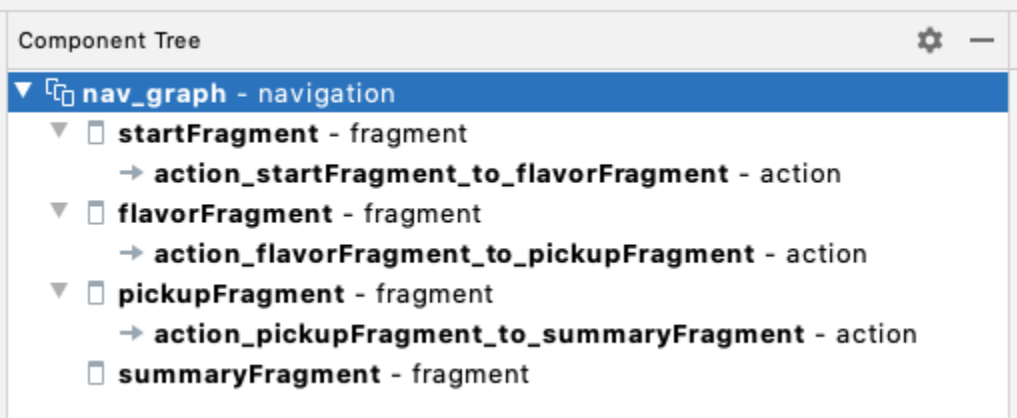


Note: If the destination fragments are laid out differently in your Android Studio, click and drag the destinations to rearrange similarly to the above screenshot. This makes it easier to configure navigation actions later in the codelab.

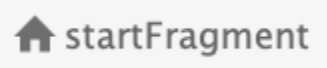
3. Connect the fragment destinations in the nav graph. Create an action from the `startFragment` to the `flavorFragment`, a connection from the `flavorFragment` to the `pickupFragment`, and a connection from the `pickupFragment` to the `summaryFragment`. Follow the next few steps if you need more detailed instructions.
4. Hover over the **startFragment** until you see the gray border around the fragment and the gray circle appear over the center of the right edge of the fragment. Click on the circle and drag to the **flavorFragment**, and then release the mouse.



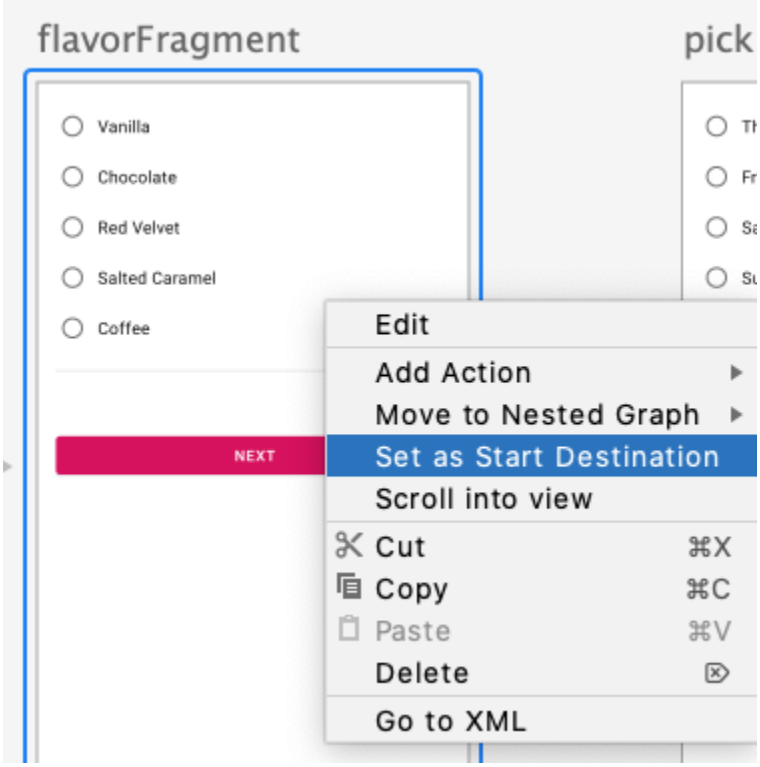
5. An arrow between the two fragments indicates a successful connection, meaning you will be able to navigate from the **startFragment** to the **flavorFragment**. This is called a Navigation action, which you have learned in a previous codelab.



8. When you define a navigation graph, you also want to specify the start destination. Currently you can see that startFragment has a little house icon next to it.



That indicates that **startFragment** will be the first fragment to be shown in the `NavHost`. Leave this as the desired behavior for our app. For future reference, you can always change the start destination by right clicking on a fragment and selecting the menu option **Set as Start Destination**.



Navigate from start fragment to flavor fragment

Next, you will add code to navigate from **startFragment** to **flavorFragment** by tapping the buttons in the first fragment, instead of displaying a `Toast` message. Below is the reference of the start fragment layout. You will pass the quantity of cupcakes to the flavor fragment in a later task.

1. In the **Project** window, open the **app > java > com.example.cupcake > StartFragment** Kotlin file.
2. In the `onViewCreated()` method, notice the click listeners are set on the three buttons. When each button is tapped, the `orderCupcake()` method is called with the quantity of cupcakes (either 1, 6, or 12 cupcakes) as its parameter.

Reference code:

```
orderOneCupcake.setOnClickListener { orderCupcake(1) }
orderSixCupcakes.setOnClickListener { orderCupcake(6) }
orderTwelveCupcakes.setOnClickListener { orderCupcake(12) }
```

3. In the `orderCupcake()` method, replace the code displaying the toast message with the code to navigate to the flavor fragment. Get the `NavController` using `findNavController()` method and call `navigate()` on it, passing in the action ID, `R.id.action_startFragment_to_flavorFragment`. Make sure this action ID matches the action declared in your `nav_graph.xml`.

Replace

```
fun orderCupcake(quantity: Int) {
    Toast.makeText(activity, "Ordered $quantity cupcake(s)",
Toast.LENGTH_SHORT).show()
}
```

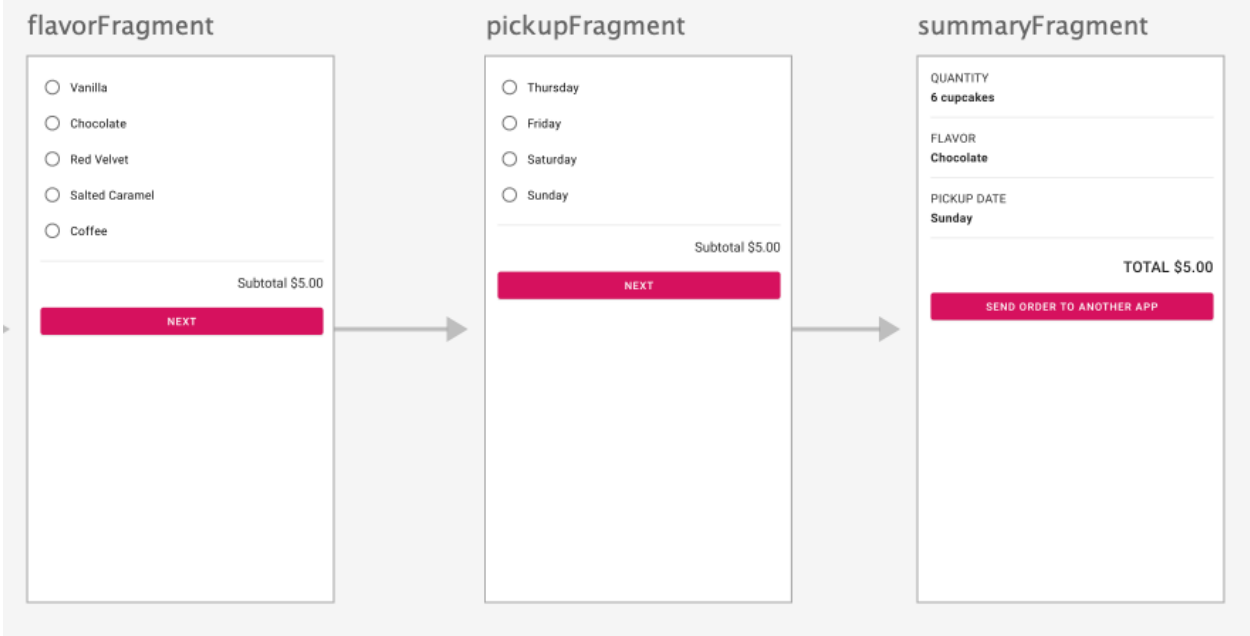
with

```
fun orderCupcake(quantity: Int) {
   findNavController().navigate(R.id.action_startFragment_to_flavorFragment)
}
```

4. Add the Import `import androidx.navigation.fragment.findNavController` or you can select from the options provided by Android Studio.

Add Navigation to the flavor and pickup fragments

Similar to the previous task, in this task you will add the navigation to the other fragments: flavor and the pickup fragments.



1. Open **app > java > com.example.cupcake > FlavorFragment.kt**. Notice the method called within the **Next** button click listener is `goToNextScreen()` method.
2. In `FlavorFragment.kt`, inside the `goToNextScreen()` method, replace the code displaying the toast to navigate to the pickup fragment. Use the action ID, `R.id.action_flavorFragment_to_pickupFragment` and make sure this ID matches the action declared in your `nav_graph.xml`.

```
fun goToNextScreen() {
    findNavController().navigate(R.id.action_flavorFragment_to_pickupFragment
)
}
```

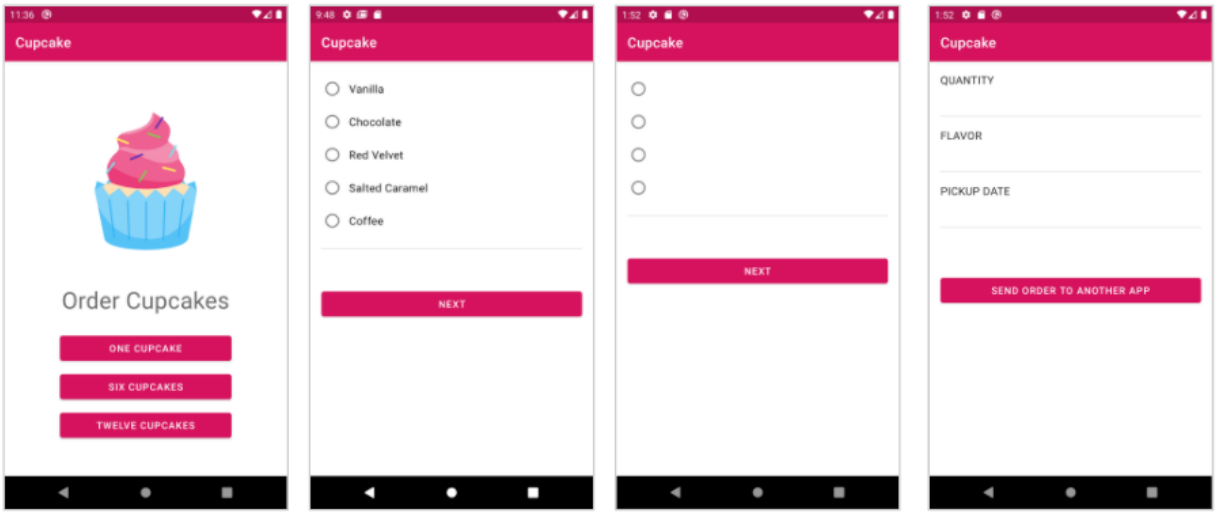
Remember to `import androidx.navigation.fragment.findNavController`.

3. Similarly in `PickupFragment.kt`, inside the `goToNextScreen()` method, replace the existing code to navigate to the summary fragment.

```
fun goToNextScreen() {
    findNavController().navigate(R.id.action_pickupFragment_to_summaryFragmen
t)
}
```

Import `androidx.navigation.fragment.findNavController`.

4. Run the app. Make sure the buttons work to navigate from screen to screen. The information displayed on each fragment may be incomplete, but don't worry, you'll be populating those fragments with the correct data in upcoming steps.

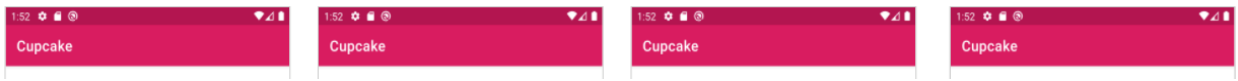


Update title in app bar

As you navigate through the app, notice the title in the app bar. It is always displayed as **Cupcake.**

It would be a better user experience to provide a more relevant title based on the functionality of the current fragment.

Change the title in the app bar (also known as action bar) for each fragment using the `NavController` and display an **Up** (←) button.



1. In `MainActivity.kt`, override the `onCreate()` method to set up the navigation controller. Get an instance of `NavController` from the `NavHostFragment`.
2. Make a call to `setupActionBarWithNavController(navController)` passing in the instance of `NavController`. This will do the following: Show a title in the app bar based off of the destination's label, and display the **Up** button whenever you're not on a top-level destination.

```
class MainActivity : AppCompatActivity(R.layout.activity_main) {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val navHostFragment = supportFragmentManager
                .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController

        setupActionBarWithNavController(navController)
```

```
    }
}
```

3. Add necessary imports when prompted by Android Studio.

```
import android.os.Bundle
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.setupActionBarWithNavController
```

4. Set the app bar titles for each fragment. Open `navigation/nav_graph.xml` and switch to **Code** tab.
5. In `nav_graph.xml`, modify the `android:label` attribute for each fragment destination. Use the following string resources that have already been declared in the starter app.

For start fragment, use `@string/app_name` with value `Cupcake`.

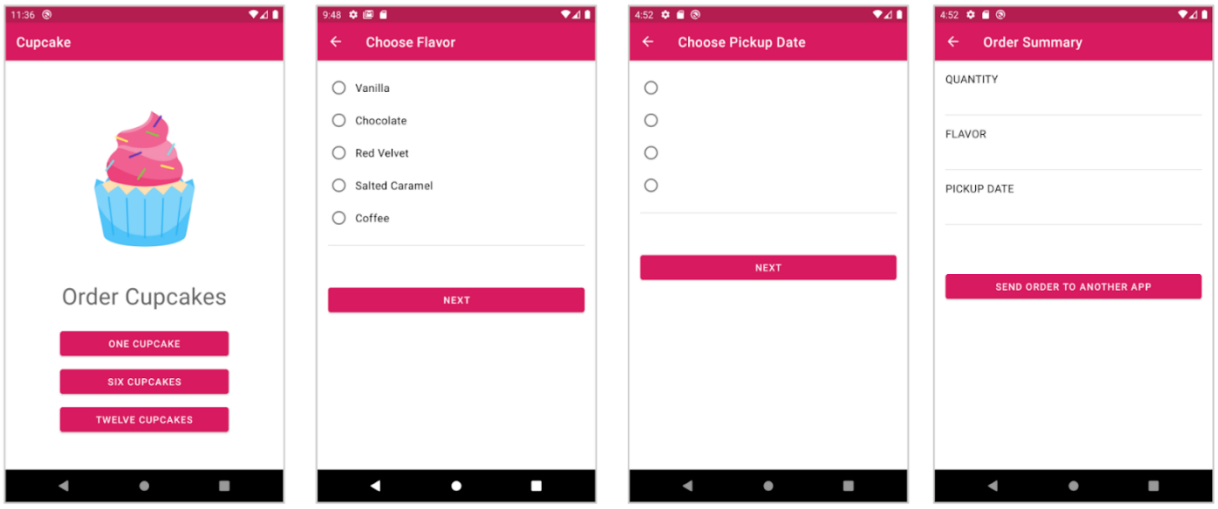
For flavor fragment, use `@string/choose_flavor` with value `Choose Flavor`.

For pickup fragment, use `@string/choose_pickup_date` with value `Choose Pickup Date`.

For summary fragment, use `@string/order_summary` with value `Order Summary`.

```
<navigation ...>
    <fragment
        android:id="@+id/startFragment"
        ...
        android:label="@string/app_name" ... >
        <action ... />
    </fragment>
    <fragment
        android:id="@+id/flavorFragment"
        ...
        android:label="@string/choose_flavor" ... >
        <action ... />
    </fragment>
    <fragment
        android:id="@+id/pickupFragment"
        ...
        android:label="@string/choose_pickup_date" ... >
        <action ... />
    </fragment>
    <fragment
        android:id="@+id/summaryFragment"
        ...
        android:label="@string/order_summary" ... />
</navigation>
```

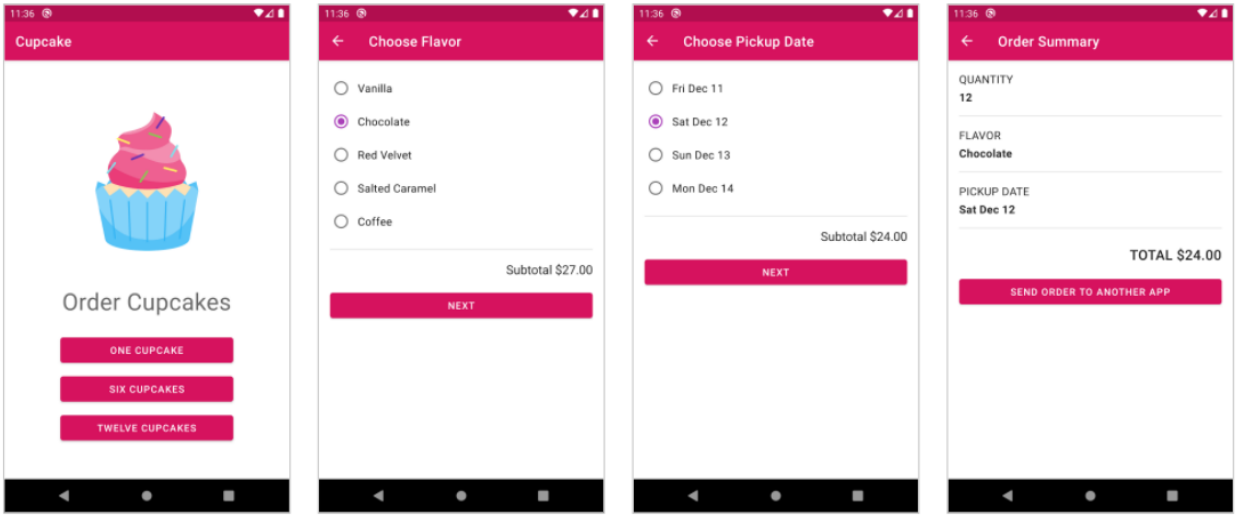
6. Run the app. Notice the title in the app bar changes as you navigate to each fragment destination. Also notice that the **Up** button (arrow ←) is now showing in the app bar. If you tap on it, it doesn't do anything. You will implement the **Up** button behavior in the next codelab.



4. Create a shared ViewModel

Let's move onto populating the correct data in each of the fragments. You'll be using a shared `ViewModel` to save the app's data in a single `ViewModel`. Multiple fragments in the app will access the shared `ViewModel` using their activity scope.

It is a common use case to share data between fragments in most production apps. For example in the final version(of this codelab) of the **Cupcake** app (notice the screenshots below), the user selects the quantity of cupcakes in the first screen, and in the second screen the price is calculated and displayed based on the quantity of the cupcakes. Similarly other app data such as flavor and pickup date are also used in summary screen.



From looking at the app features, you can reason that it would be useful to store this order information in a single `ViewModel`, which can be shared across the fragments in this activity.

Recollect that `ViewModel` is a part of the Android Architecture Components. The app data saved within the `ViewModel` is retained during configuration changes. To add a `ViewModel` to your app, you create a new class that extends from the `ViewModel` class.

Create OrderViewModel

In this task, you will create a shared `ViewModel` for the **Cupcake** app called `OrderViewModel`. You will also add the app data as properties inside the `ViewModel` and methods to update and modify the data. Here are the properties of the class:

- Order quantity (`Integer`)
- Cupcake flavor (`String`)
- Pickup date (`String`)
- Price (`Double`)

Follow `ViewModel` best practices

In a `ViewModel`, it is a recommended practice to *not* expose view model data as `public` variables. Otherwise the app data can be modified in unexpected ways by the external classes and create edge cases your app didn't expect to handle. Instead, make these mutable properties `private`, implement a backing property, and expose a `public` immutable version of each property, if needed. The convention is to prefix the name of the `private` mutable properties with an underscore (_).

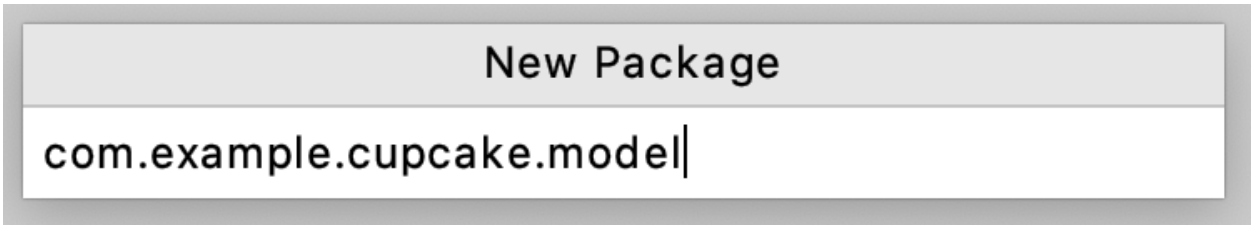
Here are the methods to update the properties above, depending on the user's choice:

- `setQuantity(numberCupcakes: Int)`
- `setFlavor(desiredFlavor: String)`
- `setDate(pickupDate: String)`

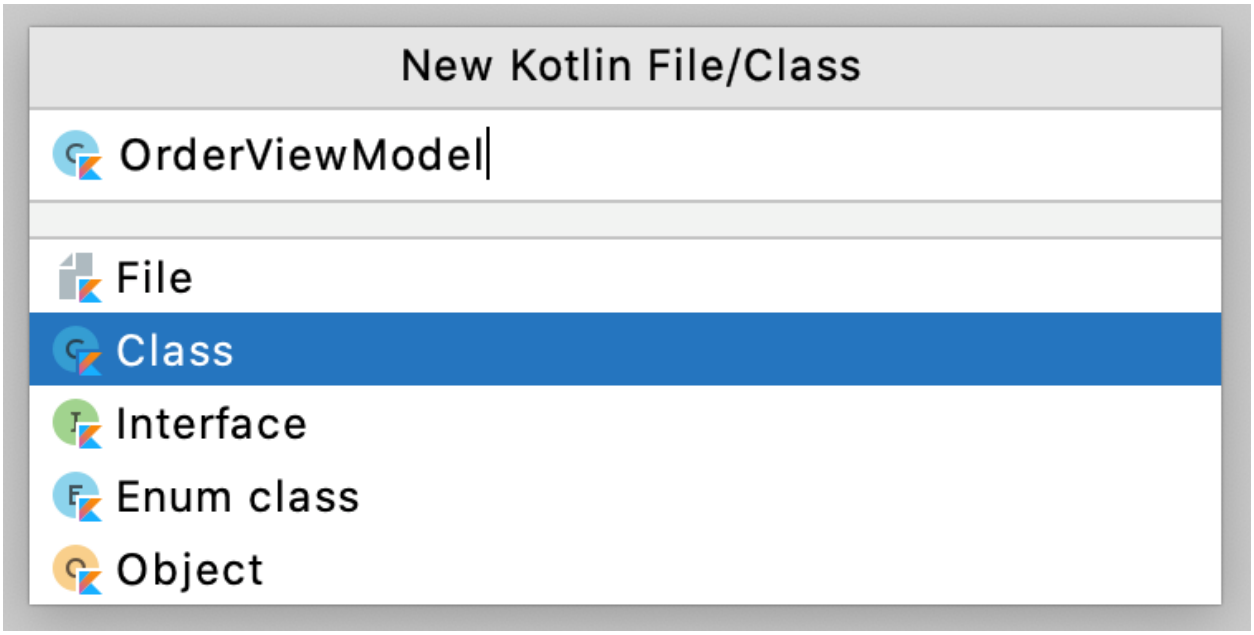
You don't need a setter method for the price because you will calculate it within the `OrderViewModel` using other properties. The steps below walk you through how to implement the shared `ViewModel`.

You will create a new package in your project called `model` and add the `OrderViewModel` class. This will separate out the view model code from the rest of your UI code (fragments and activities). It is a coding best practice to separate code into packages depending on the functionality.

1. In the **Project** window of Android Studio, right click on **com.example.cupcake** > **New** > **Package**.
2. A **New Package** dialog will be opened, give the package name as `com.example.cupcake.model`.



3. Create the `OrderViewModel` Kotlin class under the `model` package. In the **Project** window, right-click on the `model` package and select **New > Kotlin File/Class**. In the new dialog, give the filename `OrderViewModel`.



4. In `OrderViewModel.kt`, change the class signature to extend from `ViewModel`.

```
import androidx.lifecycle.ViewModel

class OrderViewModel : ViewModel() {

}
```

5. Inside the `OrderViewModel` class, add the properties that were discussed above as `private val`.
6. Change the property types to `LiveData` and add backing fields to the properties, so that these properties can be observable and UI can be updated when the source data in the view model changes.

```
private val _quantity = MutableLiveData<Int>(0)
val quantity: LiveData<Int> = _quantity

private val _flavor = MutableLiveData<String>("")
val flavor: LiveData<String> = _flavor
```

```
private val _date = MutableLiveData<String>("")
val date: LiveData<String> = _date

private val _price = MutableLiveData<Double>(0.0)
val price: LiveData<Double> = _price
```

You will need to import these classes:

```
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
```

7. In `OrderViewModel` class, add the methods that were discussed above. Inside the methods, assign the argument passed in to the mutable properties.
8. Since these setter methods need to be called from outside the view model, leave them as `public` methods (meaning no `private` or other visibility modifier needed before the `fun` keyword). The default visibility modifier in Kotlin is `public`.

```
fun setQuantity(numberCupcakes: Int) {
    _quantity.value = numberCupcakes
}

fun setFlavor(desiredFlavor: String) {
    _flavor.value = desiredFlavor
}

fun setDate(pickupDate: String) {
    _date.value = pickupDate
}
```

9. Build and run your app to make sure there are no compile errors. There should be no visible change in your UI yet.

Nice work! Now you have the start to your view model. You'll incrementally add more to this class as you build out more features in your app and realize you need more properties and methods in your class.

If you see the class names, property names, or method names in gray font in Android Studio, that's expected. That means the class, properties, or methods or not being used at the moment, but they will be! That's coming up next.

5. Use the ViewModel to update the UI

In this task, you will use the shared view model you created to update the app's UI. The main difference in the implementation of a shared view model is the way we access it from the UI controllers. You will use the activity instance instead of the fragment instance, and you will see how to do this in the coming sections.

That means the view model can be shared across fragments. Each fragment could access the view model to check on some detail of the order or update some data in the view model.

Update StartFragment to use view model

To use the shared view model in `StartFragment` you will initialize the `OrderViewModel` using `activityViewModels()` instead of `viewModels()` delegate class.

- `viewModels()` gives you the `ViewModel` instance scoped to the current fragment. This will be different for different fragments.
- `activityViewModels()` gives you the `ViewModel` instance scoped to the current activity. Therefore the instance will remain the same across multiple fragments in the same activity.

Use Kotlin property delegate

In Kotlin, each mutable (`var`) property has default getter and setter functions automatically generated for it. The setter and getter functions are called when you assign a value or read the value of the property. (For a read-only property (`val`), only the getter function is generated by default. This getter function is called when you read the value of a read-only property.)

Property delegation in Kotlin helps you to handoff the getter-setter responsibility to a different class.

This class (called *delegate class*) provides getter and setter functions of the property and handles its changes.

A delegate property is defined using the `by` clause and a delegate class instance:

```
// Syntax for property delegation
var <property-name> : <property-type> by <delegate-class>()
```

1. In `StartFragment` class, get a reference to the shared view model as a class variable. Use the `by activityViewModels()` Kotlin property delegate from the `fragment-ktx` library.

```
private val sharedViewModel: OrderViewModel by activityViewModels()
```

You may need these new imports:

```
import androidx.fragment.app.activityViewModels
import com.example.cupcake.model.OrderViewModel
```

2. Repeat the above step for `FlavorFragment`, `PickupFragment`, `SummaryFragment` classes, you will use this `sharedViewModel` instance in later sections of the codelab.

3. Going back to the `StartFragment` class, you can now use the view model. At the beginning of the `orderCupcake()` method, call the `setQuantity()` method in the shared view model to update quantity, before navigating to the flavor fragment.

```
fun orderCupcake(quantity: Int) {
    sharedViewModel.setQuantity(quantity)
    findNavController().navigate(R.id.action_startFragment_to_flavorFragment)
}
```

4. Within the `OrderViewModel` class, add the following method to check if the flavor for the order has been set or not. You will use this method in the `StartFragment` class in a later step.

```
fun hasNoFlavorSet(): Boolean {
    return _flavor.value.isNullOrEmpty()
}
```

5. In `StartFragment` class, inside `orderCupcake()` method, after setting the quantity, set the default flavor as Vanilla if no flavor is set, before navigating to the flavor fragment. Your complete method will look like this:

```
fun orderCupcake(quantity: Int) {
    sharedViewModel.setQuantity(quantity)
    if (sharedViewModel.hasNoFlavorSet()) {
        sharedViewModel.setFlavor(getString(R.string.vanilla))
    }
    findNavController().navigate(R.id.action_startFragment_to_flavorFragment)
}
```

6. Build the app to make sure there are no compile errors. There should be no visible change in your UI though.

6. Use ViewModel with data binding

Next you will use data binding to bind the view model data to the UI. You will also update the shared view model based on the selections the user makes in the UI.

Refresher on Data binding

Recall that the Data Binding Library is a part of Android Jetpack. Data binding binds the UI components in your layouts to data sources in your app using a declarative format. In simpler terms, data binding is binding data (from code) to views + view binding (binding views to code). By setting up these bindings and having updates be automatic, this helps you reduce the chance for errors if you forget to manually update the UI from your code.

Update flavor with user choice

1. In `layout/fragment_flavor.xml`, add a `<data>` tag inside the root `<layout>` tag. Add a layout variable called `viewModel` of the type `com.example.cupcake.model.OrderViewModel`. Make sure the package name in the type attribute matches with the package name of the shared view model class, `OrderViewModel` in your app.

```
<layout ...>

    <data>
        <variable
            name="viewModel"
            type="com.example.cupcake.model.OrderViewModel" />
    </data>

    <ScrollView ...>

    ...
```

2. Similarly, repeat the above step for `fragment_pickup.xml`, and `fragment_summary.xml` to add the `viewModel` layout variable. You will use this variable in later sections. You don't need to add this code in `fragment_start.xml`, because this layout doesn't use the shared view model.
3. In the `FlavorFragment` class, inside `onViewCreated()`, bind the view model instance with the shared view model instance in the layout. Add the following code inside the `binding?.apply` block.

```
binding?.apply {
    viewModel = sharedViewModel
    ...
}
```

Apply scope function

This may be the first time you're seeing the `apply` function in Kotlin. *`apply`* is a scope function in the Kotlin standard library. It executes a block of code within the context of an object. It forms a temporary scope, and in that scope, you can access the object without its name. The common use case for `apply` is to configure an object. Such calls can be read as *"apply the following assignments to the object."*

Example:

```
clark.apply {
    firstName = "Clark"
    lastName = "James"
    age = 18
}

// The equivalent code without apply scope function would look like the
following.
```

```
clark.firstName = "Clark"
clark.lastName = "James"
clark.age = 18
```

4. Repeat the same step for the `onViewCreated()` method inside the `PickupFragment` and `SummaryFragment` classes.

```
binding?.apply {
    viewModel = sharedViewModel
    ...
}
```

5. In `fragment_flavor.xml`, use the new layout variable, `viewModel` to set the `checked` attribute of the radio buttons based on the `flavor` value in the view model. If the flavor represented by a radio button is the same as the flavor that's saved in the view model, then display the radio button as selected (`checked` = true). The binding expression for the checked state of the **Vanilla** `RadioButton` would look like the following:

```
@{viewModel.flavor.equals(@string/vanilla)}
```

Essentially, you are comparing the `viewModel.flavor` property with the corresponding string resource using the `equals` function, to determine if the checked state should be true or false.

Note: Remember that binding expressions start with an `@` symbol and are wrapped inside curly braces `{}`.

```
<RadioGroup
   ...>

   <RadioButton
       android:id="@+id/vanilla"
       ...
       android:checked="@{viewModel.flavor.equals(@string/vanilla)}"
       .../>

   <RadioButton
       android:id="@+id/chocolate"
       ...
       android:checked="@{viewModel.flavor.equals(@string/chocolate)}"
       .../>

   <RadioButton
       android:id="@+id/red_velvet"
       ...
       android:checked="@{viewModel.flavor.equals(@string/red_velvet)}"
       .../>

   <RadioButton
       android:id="@+id/salted_caramel"
       ...
       android:checked="@{viewModel.flavor.equals(@string/salted_caramel)}"
       .../>
```

```
    <RadioButton
        android:id="@+id/coffee"
        ...
        android:checked="@{viewModel.flavor.equals(@string/coffee)}"
        .../>
</RadioGroup>
```

Listener bindings

Listener bindings are lambda expressions that run when an event happens, such as an `onClick` event. They are similar to method references such as `textview.setOnClickListener(clickListener)` but listener bindings let you run arbitrary data binding expressions.

1. In `fragment_flavor.xml`, add event listeners to the radio buttons using listener bindings. Use a lambda expression with no parameters and make a call to the `viewModel.setFlavor()` method by passing in the corresponding flavor string resource.

```
<RadioGroup
    ...>

    <RadioButton
        android:id="@+id/vanilla"
        ...
        android:onClick="@{() -> viewModel.setFlavor(@string/vanilla)}"
        .../>

    <RadioButton
        android:id="@+id/chocolate"
        ...
        android:onClick="@{() -> viewModel.setFlavor(@string/chocolate)}"
        .../>

    <RadioButton
        android:id="@+id/red_velvet"
        ...
        android:onClick="@{() -> viewModel.setFlavor(@string/red_velvet)}"
        .../>

    <RadioButton
        android:id="@+id/salted_caramel"
        ...
        android:onClick="@{() -> viewModel.setFlavor(@string/salted_caramel)}"
        .../>

    <RadioButton
        android:id="@+id/coffee"
        ...
        android:onClick="@{() -> viewModel.setFlavor(@string/coffee)}"
        .../>
</RadioGroup>
```

7. Run the app and notice how the **Vanilla** option is selected by default in the flavor fragment.

Great! Now you can move onto the next fragments.

7. Update pickup and summary fragment to use view model

Navigate through the app and notice that in the pickup fragment, the radio button option labels are blank. In this task, you will calculate the 4 pickup dates available and display them in the pickup fragment. There are different ways to display a formatted date, and here are some helpful utilities provided by Android to do this.

Create pickup options list

Date formatter

The Android framework provides a class called `SimpleDateFormat`, which is a class for formatting and parsing dates in a locale-sensitive manner. It allows for formatting (date → text) and parsing (text → date) of dates.

You can create an instance of `SimpleDateFormat` by passing in a pattern string and a locale:

```
SimpleDateFormat("E MMM d", Locale.getDefault())
```

A pattern string like `"E MMM d"` is a representation of Date and Time formats. Letters from `'A'` to `'Z'` and from `'a'` to `'z'` are interpreted as pattern letters representing the components of a date or time string. For example, `d` represents day in a month, `y` for year and `M` for month. If the date is January 4 in 2018, the pattern string `"EEE, MMM d"` parses to `"Wed, Jul 4"`. For a complete list of pattern letters, please see the documentation.

A `Locale` object represents a specific geographical, political, or cultural region. It represents a language/country/variant combination. Locales are used to alter the presentation of information such as numbers or dates to suit the conventions in the region. Date and time are locale-sensitive, because they are written differently in different parts of the world. You will use the method `Locale.getDefault()` to retrieve the locale information set on the user's device and pass it into the `SimpleDateFormat` constructor.

Locale in Android is a combination of language and country code. The language codes are two-letter lowercase ISO language codes, such as "en" for english. The country codes are two-letter uppercase ISO country codes, such as "US" for the United States.

Now use `SimpleDateFormat` and `Locale` to determine the available pickup dates for the **Cupcake** app.

1. In `OrderViewModel` class, add the following function called `getPickupOptions()` to create and return the list of pickup dates. Within the method, create a `val` variable called `options` and initialize it to *`mutableListOf`*`<String>()`.

```
private fun getPickupOptions(): List<String> {
   val options = mutableListOf<String>()
}
```

2. Create a formatter string using `SimpleDateFormat` passing pattern string `"E MMM d"`, and the locale. In the pattern string, `E` stands for day name in week and it parses to "**Tue Dec 10**".

```
val formatter = SimpleDateFormat("E MMM d", Locale.getDefault())
```

Import `java.text.SimpleDateFormat` and `java.util.Locale`, when prompted by Android Studio.

3. Get a `Calendar` instance and assign it to a new variable. Make it a `val`. This variable will contain the current date and time. Also, import `java.util.Calendar`.

```
val calendar = Calendar.getInstance()
```

4. Build up a list of dates starting with the current date and the following three dates. Because you'll need 4 date options, repeat this block of code 4 times. This `repeat` block will format a date, add it to the list of date options, and then increment the calendar by 1 day.

```
repeat(4) {
    options.add(formatter.format(calendar.time))
    calendar.add(Calendar.DATE, 1)
}
```

5. Return the updated `options` at the end of the method. Here is your completed method:

```
private fun getPickupOptions(): List<String> {
   val options = mutableListOf<String>()
   val formatter = SimpleDateFormat("E MMM d", Locale.getDefault())
   val calendar = Calendar.getInstance()
   // Create a list of dates starting with the current date and the following
3 dates
   repeat(4) {
       options.add(formatter.format(calendar.time))
       calendar.add(Calendar.DATE, 1)
   }
   return options
}
```

6. In `OrderViewModel` class, add a class property called `dateOptions` that's a `val`. Initialize it using the `getPickupOptions()` method you just created.

```
val dateOptions = getPickupOptions()
```

Update the layout to display pickup options

Now that you have the four available pickup dates in the view model, update the `fragment_pickup.xml` layout to display these dates. You will also use data binding to display the checked status of each radio button and to update the date in the view model when a different radio button is selected. This implementation is similar to the data binding in the flavor fragment.

In `fragment_pickup.xml`:

Radio button `option0` represents `dateOptions[0]` in `viewModel` (today)

Radio button `option1` represents `dateOptions[1]` in `viewModel` (tomorrow)

Radio button `option2` represents `dateOptions[2]` in `viewModel` (the day after tomorrow)

Radio button `option3` represents `dateOptions[3]` in `viewModel` (two days after tomorrow)

1. In `fragment_pickup.xml`, for the `option0` radio button, use the new layout variable, `viewModel` to set the `checked` attribute based on the `date` value in the view model. Compare the `viewModel.date` property with the first string in the `dateOptions` list, which is the current date. Use the `equals` function to compare and the final binding expression looks like the following:

```
@{viewModel.date.equals(viewModel.dateOptions[0])}
```

2. For the same radio button, add an event listener using listener binding to the `onClick` attribute. When this radio button option is clicked, make a call to `setDate()` on `viewModel`, passing in `dateOptions[0]`.
3. For the same radio button, set the `text` attribute value to the first string in the `dateOptions` list.

```
<RadioButton
   android:id="@+id/option0"
   ...
   android:checked="@{viewModel.date.equals(viewModel.dateOptions[0])}"
   android:onClick="@{() -> viewModel.setDate(viewModel.dateOptions[0])}"
   android:text="@{viewModel.dateOptions[0]}"
   ...
   />
```

4. Repeat the above steps for the other radio buttons, change the index of the `dateOptions` accordingly.

```
<RadioButton
   android:id="@+id/option1"
   ...
   android:checked="@{viewModel.date.equals(viewModel.dateOptions[1])}"
   android:onClick="@{() -> viewModel.setDate(viewModel.dateOptions[1])}"
   android:text="@{viewModel.dateOptions[1]}"
   ... />
```

```
<RadioButton
   android:id="@+id/option2"
   ...
   android:checked="@{viewModel.date.equals(viewModel.dateOptions[2])}"
   android:onClick="@{() -> viewModel.setDate(viewModel.dateOptions[2])}"
   android:text="@{viewModel.dateOptions[2]}"
   ... />

<RadioButton
   android:id="@+id/option3"
   ...
   android:checked="@{viewModel.date.equals(viewModel.dateOptions[3])}"
   android:onClick="@{() -> viewModel.setDate(viewModel.dateOptions[3])}"
   android:text="@{viewModel.dateOptions[3]}"
   ... />
```

5. Run the app and you should see the next few days as pickup options available. Your screenshot will differ depending on what the current day is for you. Notice that there is no option selected by default. You will implement this in the next step.

6. Within the `OrderViewModel` class, create a function called `resetOrder()`, to reset the `MutableLiveData` properties in the view model. Assign the current date value from the `dateOptions` list to `_date.value`.

```
fun resetOrder() {
    _quantity.value = 0
    _flavor.value = ""
    _date.value = dateOptions[0]
    _price.value = 0.0
}
```

7. Add an `init` block to the class, and call the new method `resetOrder()` from it.

```
init {
    resetOrder()
}
```

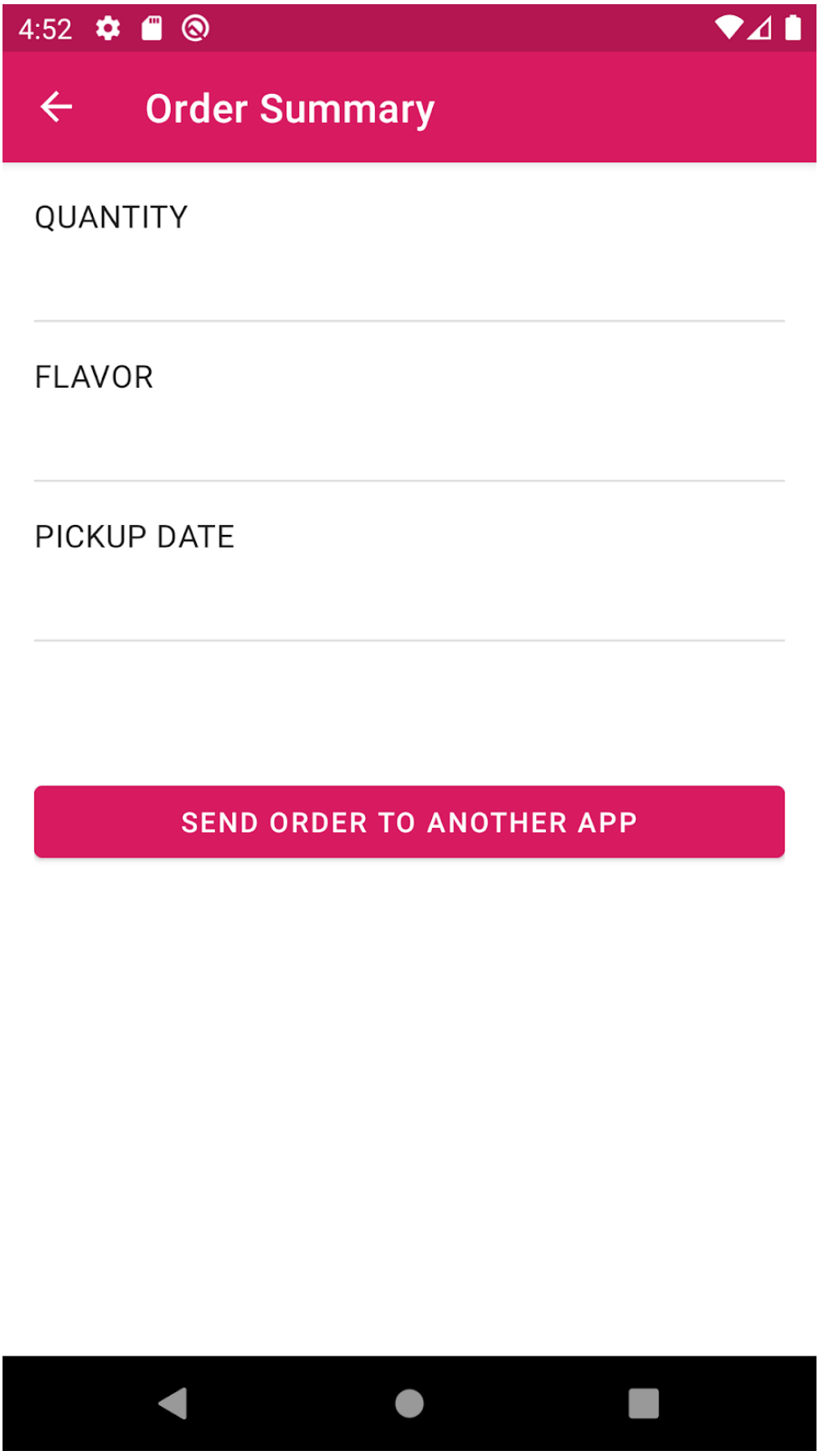
8. Remove the initial values from the declaration of the properties in the class. Now you are using the `init` block to initialize the properties when an instance of `OrderViewModel` is created.

```
private val _quantity = MutableLiveData<Int>()
val quantity: LiveData<Int> = _quantity

private val _flavor = MutableLiveData<String>()
val flavor: LiveData<String> = _flavor

private val _date = MutableLiveData<String>()
val date: LiveData<String> = _date

private val _price = MutableLiveData<Double>()
val price: LiveData<Double> = _price
```

9. Run your app again, notice today's date is selected by default.

Update Summary fragment to use view model

Now let's move onto the last fragment. The order summary fragment is intended to show a summary of the order details. In this task, you take advantage of all the order information from the shared view model and update the onscreen order details using data binding.


4:52
Order Summary
QUANTITY
FLAVOR
PICKUP DATE
SEND ORDER TO ANOTHER APP

1. In `fragment_summary.xml`, make sure you have the view model data variable, `viewModel` declared.

```
<layout ...>

    <data>
        <variable
            name="viewModel"
            type="com.example.cupcake.model.OrderViewModel" />
    </data>

    <ScrollView ...>

    ...
```

2. In `SummaryFragment`, in `onViewCreated()`, make sure `binding.viewModel` is initialized.
3. In `fragment_summary.xml`, read from the view model to update the screen with the order summary details. Update the quantity, flavor, and date `TextViews` by adding the following text attributes. Quantity is of the type `Int`, so you need to convert it to a string.

```
<TextView
   android:id="@+id/quantity"
   ...
   android:text="@{viewModel.quantity.toString()}"
   ... />
<TextView
   android:id="@+id/flavor"
   ...
   android:text="@{viewModel.flavor}"
   ... />
<TextView
   android:id="@+id/date"
   ...
   android:text="@{viewModel.date}"
   ... />
```

4. Run and test the app to verify that the order options you selected show up in the order summary.

Order Summary

QUANTITY

1

FLAVOR

Salted Caramel

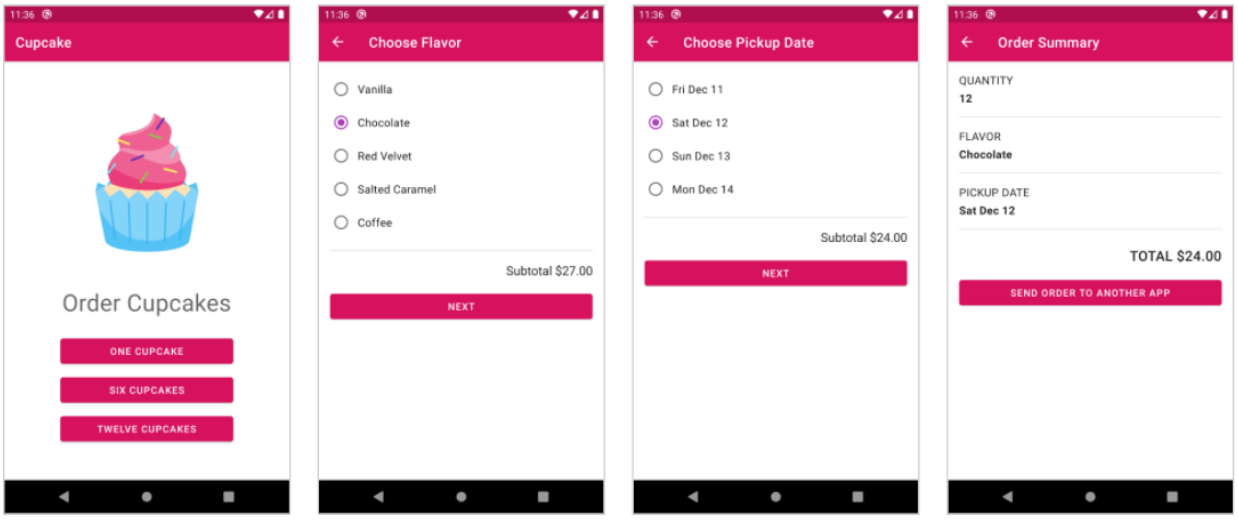
PICKUP DATE

Sat Dec 12

SEND ORDER TO ANOTHER APP

8. Calculate price from order details

Looking at the final app screenshots of this codelab, you'll notice that the price is actually displayed on each fragment (except the `StartFragment`) so the user knows the price as they create the order.



Here are the rules from our cupcake shop on how to calculate price.

- Each cupcake is $2.00 each
- Same day pickup adds an extra $3.00 to the order

Hence, for an order of 6 cupcakes, the price would be 6 cupcakes x $2 each = $12. If the user wants same day pickup, the extra $3 cost would lead to a total order price of $15.

Update price in view model

To add support for this functionality in your app, first tackle the price per cupcake and ignore the same day pickup cost for now.

1. Open `OrderViewModel.kt`, and store the price per cupcake in a variable. Declare it as a top-level private constant at the top of the file, outside the class definition (but after the import statements). Use the `const` modifier and to make it read-only use `val`.

```
package ...

import ...

private const val PRICE_PER_CUPCAKE = 2.00

class OrderViewModel : ViewModel() {
    ...
```

Recollect that constant values (marked with the `const` keyword in Kotlin) do not change and the value is known at compile time. To learn more about constants, check out the documentation.

2. Now that you have defined a price per cupcake, create a helper method to calculate the price. This method can be `private` because it's only used within this class. You will change the price logic to include same day pickup charges in the next task.

```
private fun updatePrice() {
    _price.value = (quantity.value ?: 0) * PRICE_PER_CUPCAKE
}
```

This line of code multiplies the price per cupcake by the quantity of cupcakes ordered. For the code in parentheses, since the value of `quantity.value` could be null, use an elvis operator (`?:`) . The elvis operator (`?:`) means that if the expression on the left is not null, then use it. Otherwise if the expression on the left is null, then use the expression to the right of the elvis operator (which is `0` in this case).

Fun fact: Elvis operator (`?:`) is named after the rock star, Elvis Presley, because when you view it sideways, it resembles an emoticon of Elvis Presley with his quiff.

3. In the same `OrderViewModel` class, update the price variable when the quantity is set. Make a call to the new function in the `setQuantity()` function.

```
fun setQuantity(numberCupcakes: Int) {
    _quantity.value = numberCupcakes
    updatePrice()
}
```

Bind the price property to the UI

1. In the layouts for `fragment_flavor.xml`, `fragment_pickup.xml`, and `fragment_summary.xml`, make sure the data variable `viewModel` of type `com.example.cupcake.model.OrderViewModel` is defined.

```
<layout ...>

    <data>
        <variable
            name="viewModel"
            type="com.example.cupcake.model.OrderViewModel" />
    </data>

    <ScrollView ...>

    ...
```

2. In the `onViewCreated()` method of each fragment class, make sure you bind the view model object instance in the fragment to the view model data variable in the layout.

```
binding?.apply {
    viewModel = sharedViewModel
    ...
}
```

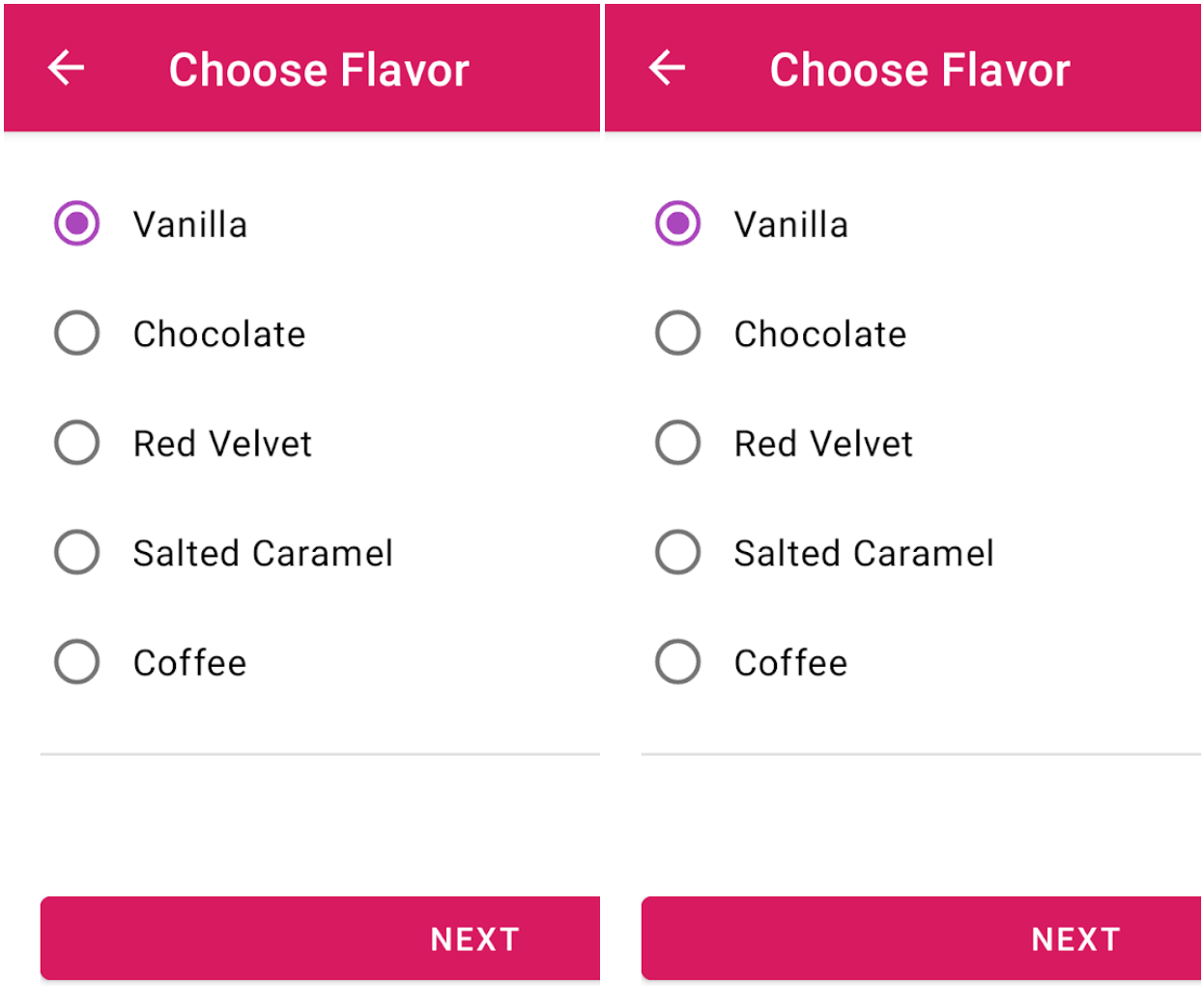
3. Within each fragment layout, use the `viewModel` variable to set the price if it's shown in the layout. Start with modifying the `fragment_flavor.xml` file. For the `subtotal` text view, set the value of the `android:text` attribute to be `"@{@string/subtotal_price(viewModel.price)}"`. This data binding layout expression uses the string resource `@string/subtotal_price` and passes in a parameter, which is the price from the view model, so the output will show **Subtotal 12.0** for example.

```
...

<TextView
    android:id="@+id/subtotal"
    android:text="@{@string/subtotal_price(viewModel.price)}"
    ... />

...
```

You're using this string resource that was already declared in the `strings.xml` file:

```
<string name="subtotal_price">Subtotal %s</string>
```

4. Run the app. If you select **One cupcake** in the start fragment, the flavor fragment will show **Subtotal 2.0**. If you select **Six cupcakes**, the flavor fragment will show **Subtotal 12.0**, and etc... You will format the price into the proper currency format later, so this behavior is expected for now.


Choose Flavor
Vanilla
Chocolate
Red Velvet
Salted Caramel
Coffee
NEXT
Choose Flavor
Vanilla
Chocolate
Red Velvet
Salted Caramel
Coffee
NEXT

5. Now make a similar change for the pickup and summary fragments. In `fragment_pickup.xml` and `fragment_summary.xml` layouts, modify the text views to use the `viewModel` `price` property as well.

`fragment_pickup.xml`

```
...

<TextView
    android:id="@+id/subtotal"
    ...
    android:text="@{@string/subtotal_price(viewModel.price)}"
    ... />

...
```

`fragment_summary.xml`

```
...

<TextView
   android:id="@+id/total"
   ...
   android:text="@{@string/total_price(viewModel.price)}"
   ... />

...
```

4. Run the app. Make sure the price shown in the order summary is calculated correctly for an order quantity of 1, 6, and 12 cupcakes. As mentioned, it's expected that the price formatting isn't correct at the moment (it'll show up as 2.0 for $2 or 12.0 for $12).

Charge extra for same day pickup

In this task, you will implement the second rule which is that same day pickup adds an extra $3.00 to the order.

1. In `OrderViewModel` class, define a new top-level private constant for the same day pickup cost.

```
private const val PRICE_FOR_SAME_DAY_PICKUP = 3.00
```

2. In `updatePrice()`, check if the user selected the same day pickup. Check if the date in the view model (`_date.value`) is the same as the first item in the `dateOptions` list which is always the current day.

```
private fun updatePrice() {
    _price.value = (quantity.value ?: 0) * PRICE_PER_CUPCAKE
    if (dateOptions[0] == _date.value) {

    }
}
```

3. To make these calculations simpler, introduce a temporary variable, `calculatedPrice`. Calculate the updated price and assign it back to `_price.value`.

```
private fun updatePrice() {
    var calculatedPrice = (quantity.value ?: 0) * PRICE_PER_CUPCAKE
    // If the user selected the first option (today) for pickup, add the
surcharge
    if (dateOptions[0] == _date.value) {
        calculatedPrice += PRICE_FOR_SAME_DAY_PICKUP
    }
    _price.value = calculatedPrice
}
```

4. Call `updatePrice()` helper method from `setDate()` method to add the same day pickup charges.

```
fun setDate(pickupDate: String) {
    _date.value = pickupDate
    updatePrice()
}
```

5. Run your app, navigate through the app. You will notice that changing the pickup date does not remove the same day pickup charges from the total price. This is because the price is changed in the view model but it is not notified to the binding layout.

Set Lifecycle owner to observe LiveData

`LifecycleOwner` is a class that has an Android lifecycle, such as an activity or a fragment. A `LiveData` observer observes the changes to the app's data only if the lifecycle owner is in active states (`STARTED` or `RESUMED`).

In your app, the `LiveData` object or the observable data is the `price` property in the view model. The lifecycle owners are the flavor, pickup and the summary fragments. The `LiveData` observers are the binding expressions in layout files with observable data like price. With Data Binding, when an observable value changes, the UI elements it's bound to are updated automatically.

Example of binding expression:
`android:text="@{@string/subtotal_price(viewModel.price)}"`

For the UI elements to automatically update, you have to associate `binding.`*`lifecycleOwner`*

with the lifecycle owners in the app. You will implement this next.

1. In the `FlavorFragment`, `PickupFragment`, `SummaryFragment` classes, inside the `onViewCreated()` method, add the following in the `binding?.apply` block. This will set the lifecycle owner on the binding object. By setting the lifecycle owner, the app will be able to observe `LiveData` objects.

```
binding?.apply {
    lifecycleOwner = viewLifecycleOwner
    ...
}
```

2. Run your app again. In the pickup screen, change the pickup date and notice the difference in how the price changes automatically. And the pick up charges are correctly reflected in the summary screen.
3. Notice that when you select today's date for pickup, the price of the order is increased by $3.00. The price for selecting any future date should still be the quantity of cupcakes x $2.00.

4. Test different cases with different cupcake quantities, flavors, and pickup dates. Now you should see the price updating from the view model on each fragment. The best part is that you didn't have to write extra Kotlin code to keep the UI updated with the price each time.

To finish implementing the price feature, you'll need to format the price to the local currency.

Format price with LiveData transformation

The `LiveData` transformation method(s) provides a way to perform data manipulations on the source `LiveData` and return a resulting `LiveData` object. In simple terms, it transforms the value of `LiveData` into another value. These transformations aren't calculated unless an observer is observing the `LiveData` object.

The `Transformations.map()` is one of the transformation functions, this method takes the source `LiveData` and a function as parameters. The function manipulates the source `LiveData` and returns an updated value which is also observable.

Some real-time examples where you may use a LiveData transformation:

- Format date, time strings for display
- Sorting a list of items
- Filtering or grouping the items
- Calculate the result from a list like sum of all the items, number of items, return the last item, and so on.

In this task, you will use `Transformations.map()` method to format the price to use the local currency. You'll transform the original price as a decimal value (`LiveData<Double>`) into a string value (`LiveData<String>`).

1. In `OrderViewModel` class, change the backing property type to `LiveData<String>` instead of `LiveData<Double>`. The formatted price will be a string with a currency symbol such as a '$'. You will fix the initialization error in the next step.

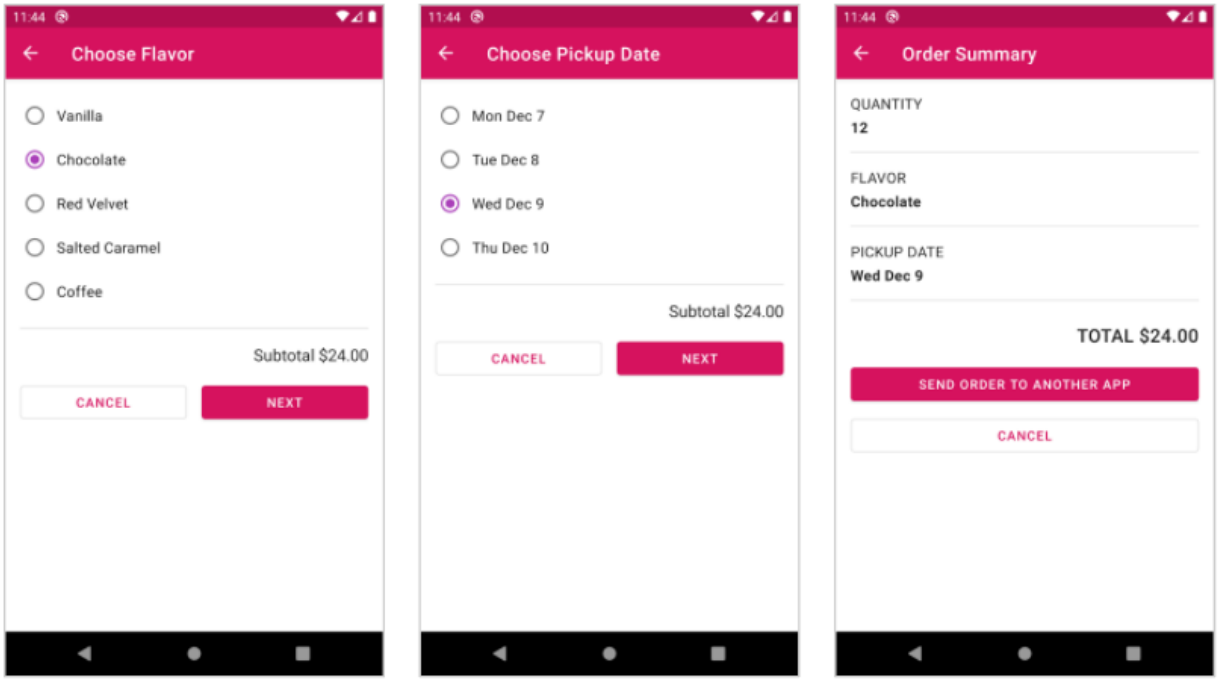
```
private val _price = MutableLiveData<Double>()
val price: LiveData<String>
```

2. Use `Transformations.map()` to initialize the new variable, pass in the `_price` and a lambda function. Use `getCurrencyInstance()` method in the `NumberFormat` class to convert the price to local currency format. The transformation code will look like this.

```
private val _price = MutableLiveData<Double>()
val price: LiveData<String> = Transformations.map(_price) {
   NumberFormat.getCurrencyInstance().format(it)
}
```

You'll need to import `androidx.lifecycle.Transformations` and `java.text.NumberFormat`.

3. Run the app. Now you should see the formatted price string for subtotal and total. This is much more user-friendly!



4. Test that it works as expected. Test cases like: Order one cupcake, order six cupcakes, order 12 cupcakes. Make sure the price is correctly updated on each screen. It should say **Subtotal $2.00** for the Flavor and Pickup fragments, and **Total $2.00** for the order summary. Also, make sure the order summary shows the correct order details.

9. Setup click listeners using listener binding

In this task, you will use listener binding to bind the button click listeners in the fragment classes to the layout.

1. In the layout file `fragment_start.xml`, add a data variable called `startFragment` of the type `com.example.cupcake.StartFragment`. Make sure the package name of the fragment matches with your app's package name.

```
<layout ...>

    <data>
        <variable
            name="startFragment"
            type="com.example.cupcake.StartFragment" />
    </data>
    ...
    <ScrollView ...>
```

2. In `StartFragment.kt`, in `onViewCreated()` method, bind the new data variable to the fragment instance. You can access the fragment instance inside the fragment using `this`

keyword. Remove the `binding?.apply` block and along with the code within. The completed method should look like this.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding?.startFragment = this
}
```

3. In `fragment_start.xml`, add event listeners using listener binding to the `onClick` attribute for the buttons, make a call to `orderCupcake()` on `startFragment`, passing in the number of cupcakes.

```
<Button
    android:id="@+id/order_one_cupcake"
    android:onClick="@{() -> startFragment.orderCupcake(1)}"
    ... />

<Button
    android:id="@+id/order_six_cupcakes"
    android:onClick="@{() -> startFragment.orderCupcake(6)}"
    ... />

<Button
    android:id="@+id/order_twelve_cupcakes"
    android:onClick="@{() -> startFragment.orderCupcake(12)}"
    ... />
```

4. Run the app. Notice the button click handlers in the start fragment are working as expected.
5. Similarly add the above data variable in other layouts as well to bind the fragment instance, `fragment_flavor.xml`, `fragment_pickup.xml`, and `fragment_summary.xml`.

In `fragment_flavor.xml`

```
<layout ...>

    <data>
        <variable
            ... />

        <variable
            name="flavorFragment"
            type="com.example.cupcake.FlavorFragment" />
    </data>

    <ScrollView ...>
```

In `fragment_pickup.xml`:

```
<layout ...>

    <data>
```

```
        <variable
            ... />

        <variable
            name="pickupFragment"
            type="com.example.cupcake.PickupFragment" />
    </data>

    <ScrollView ...>
```

In `fragment_summary.xml`:

```
<layout ...>

    <data>
        <variable
            ... />

        <variable
            name="summaryFragment"
            type="com.example.cupcake.SummaryFragment" />
    </data>

    <ScrollView ...>
```

6. In the rest of the fragment classes, in `onViewCreated()` methods, delete the code that manually sets the click listener on the buttons.
7. In the `onViewCreated()` methods bind the fragment data variable with the fragment instance. You will use `this` keyword differently here, because inside the `binding?.apply` block, the keyword `this` refers to the binding instance, not the fragment instance. Use `@` and explicitly specify the fragment class name, for example `this@FlavorFragment`. The completed `onViewCreated()` methods should look as follows:

The `onViewCreated()` method in `FlavorFragment` class should look like this:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding?.apply {
        lifecycleOwner = viewLifecycleOwner
        viewModel = sharedViewModel
        flavorFragment = this@FlavorFragment
    }
}
```

The `onViewCreated()` method in `PickupFragment` class should look like this:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
   super.onViewCreated(view, savedInstanceState)

   binding?.apply {
```

```
        lifecycleOwner = viewLifecycleOwner
        viewModel = sharedViewModel
        pickupFragment = this@PickupFragment
    }
}
```

The resulting `onViewCreated()` method in `SummaryFragment` class method should look like this:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding?.apply {
        lifecycleOwner = viewLifecycleOwner
        viewModel = sharedViewModel
        summaryFragment = this@SummaryFragment
    }
}
```

8. Similarly in the other layout files, add listener binding expressions to the `onClick` attribute for the buttons.

In `fragment_flavor.xml`:

```
<Button
    android:id="@+id/next_button"
    android:onClick="@{() -> flavorFragment.goToNextScreen()}"
    ... />
```

In `fragment_pickup.xml`:

```
<Button
    android:id="@+id/next_button"
    android:onClick="@{() -> pickupFragment.goToNextScreen()}"
    ... />
```

In `fragment_summary.xml`:

```
<Button
    android:id="@+id/send_button"
    android:onClick="@{() -> summaryFragment.sendOrder()}"
    ...>
```

9. Run the app to verify the buttons still work as expected. There should be no visible change in behavior, but now you've used listener bindings to set up the click listeners!

Congratulations on completing this codelab and building out the **Cupcake** app! However, the app is not quite done yet. In the next codelab, you will add a **Cancel** button and modify the backstack. You will also learn what is a backstack and other new topics. See you there!

Bước 3. Cập nhật các thư viện cần thiết cho Navigation UI và Safe Args

Cập nhật trong file src/build.gradle.kts

```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
    id("androidx.navigation.safeargs") version "2.8.3" apply false
}
```

Cập nhật trong file src/build.gradle.kts

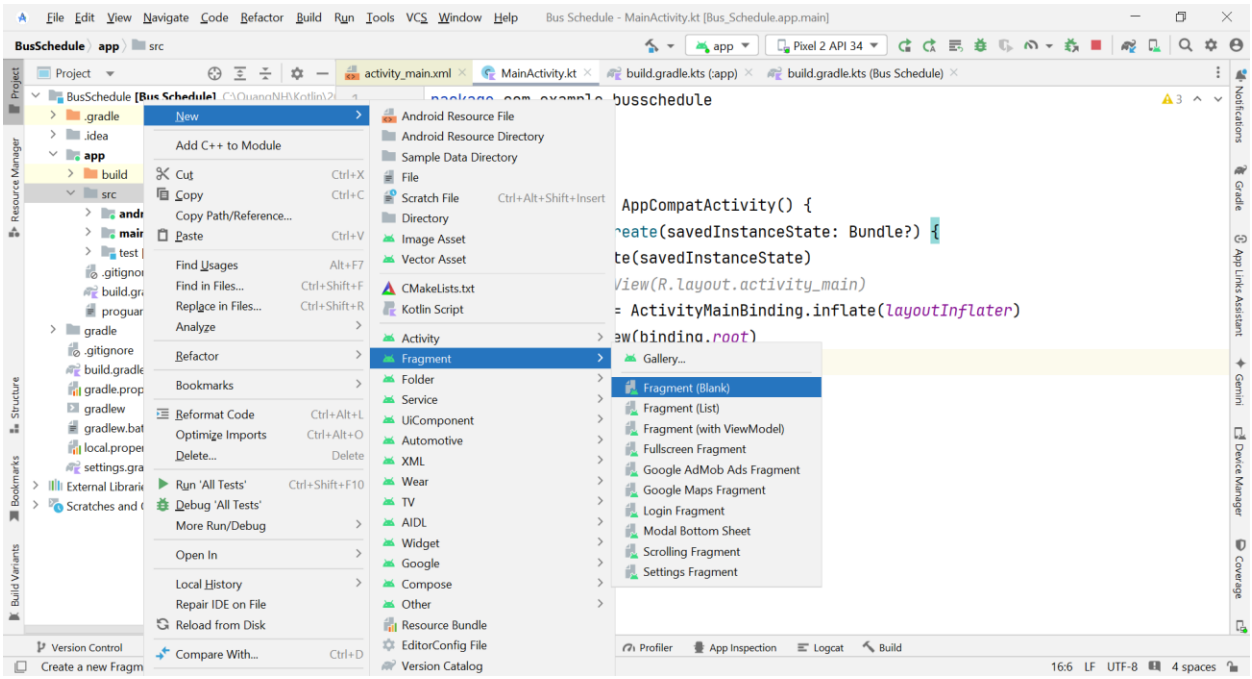
```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("androidx.navigation.safeargs.kotlin")
}

android {
    namespace = "com.example.busschedule"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.example.busschedule"
        minSdk = 26
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
```

```
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = "1.8"
    }

    buildFeatures{
        viewBinding = true
    }
}

dependencies {
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.activity)
    implementation(libs.androidx.constraintlayout)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)

    implementation("androidx.navigation:navigation-fragment-ktx:2.8.3")
    implementation("androidx.navigation:navigation-ui-ktx:2.8.3")
}
```

Bước 4. Cập nhật trong file MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //setContentView(R.layout.activity_main)
        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

Bước 5. Tạo FullScheduleFragment



Bước 6. Cập nhật file layout cho FullScheduleFragment

<?xml version="1.0" encoding="utf-8"?>

<!--

Copyright (C) 2021 The Android Open Source Project

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

```
     limitations under the License.
-->
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FullScheduleFragment"
    android:orientation="vertical">

    <TextView
        android:id="@+id/bus_stop_header"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="@string/bus_stop_header"
        android:textSize="16sp"
        android:gravity="center_horizontal"
        android:padding="8dp"
        app:layout_constraintWidth_percent="0.5"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="@id/arrival_time_header"
        app:layout_constraintEnd_toStartOf="parent"/>

    <TextView
        android:id="@+id/arrival_time_header"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

```
        android:text="@string/arrival_time_header"
        android:textSize="16sp"
        android:gravity="center_horizontal"
        android:padding="8dp"
        app:layout_constraintWidth_percent="0.5"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toEndOf="@id/bus_stop_header"
        app:layout_constraintEnd_toEndOf="parent"/>

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        app:layout_constraintTop_toBottomOf="@id/bus_stop_header"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>

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

Bổ sung các xâu ký tự vào /values/strings.xml

```
<resources>
    <string name="app_name">Bus Schedule</string>
    <!-- TODO: Remove or change this placeholder text -->
    <string name="hello_blank_fragment">Hello blank fragment</string>
    <!-- Shown above left column listing bus stops in full schedule fragment -->
    <string name="bus_stop_header">Stop Name</string>
    <!-- Shown above right column listing arrival times in full schedule fragment -->
    <string name="arrival_time_header">Arrival Time</string>
</resources>
```

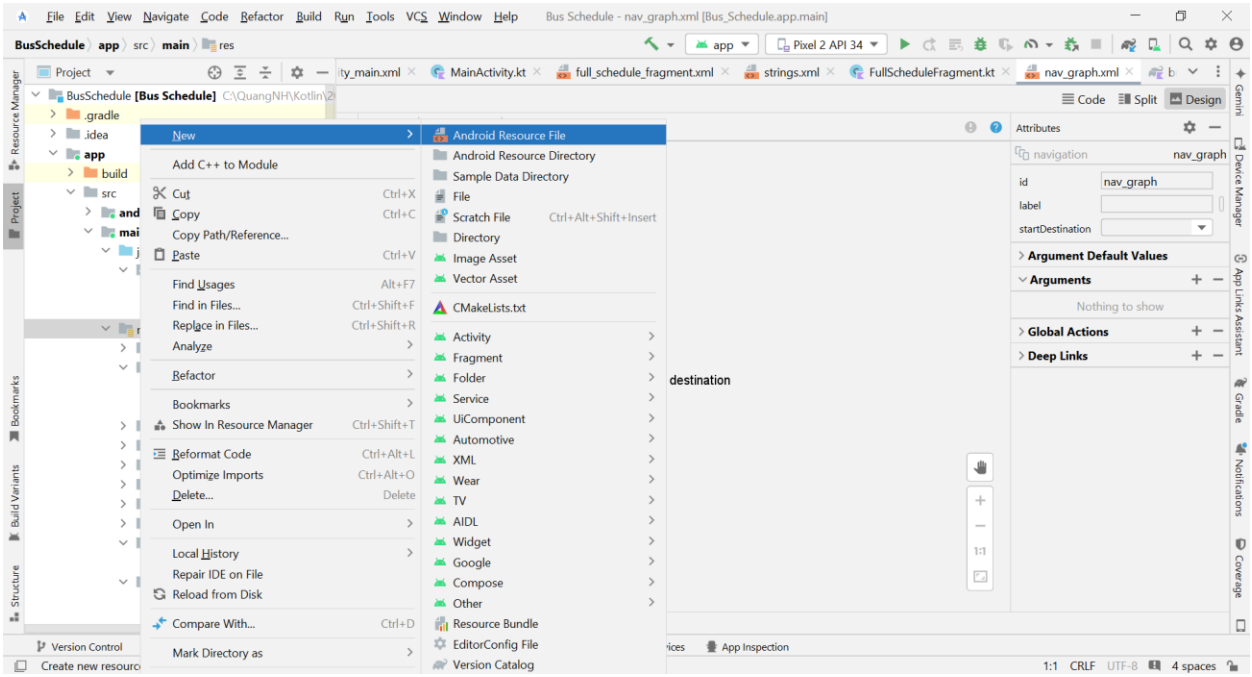
Bước 7. Cập nhật file FullScheduleFragment.kt

```
package com.example.busschedule

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.example.busschedule.databinding.FullScheduleFragmentBinding

class FullScheduleFragment : Fragment() {
    private var _binding: FullScheduleFragmentBinding? = null

    private val binding get() = _binding!!

    private lateinit var recyclerView: RecyclerView

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FullScheduleFragmentBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
```

```
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        recyclerView = binding.recyclerView
        recyclerView.layoutManager = LinearLayoutManager(requireContext())
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

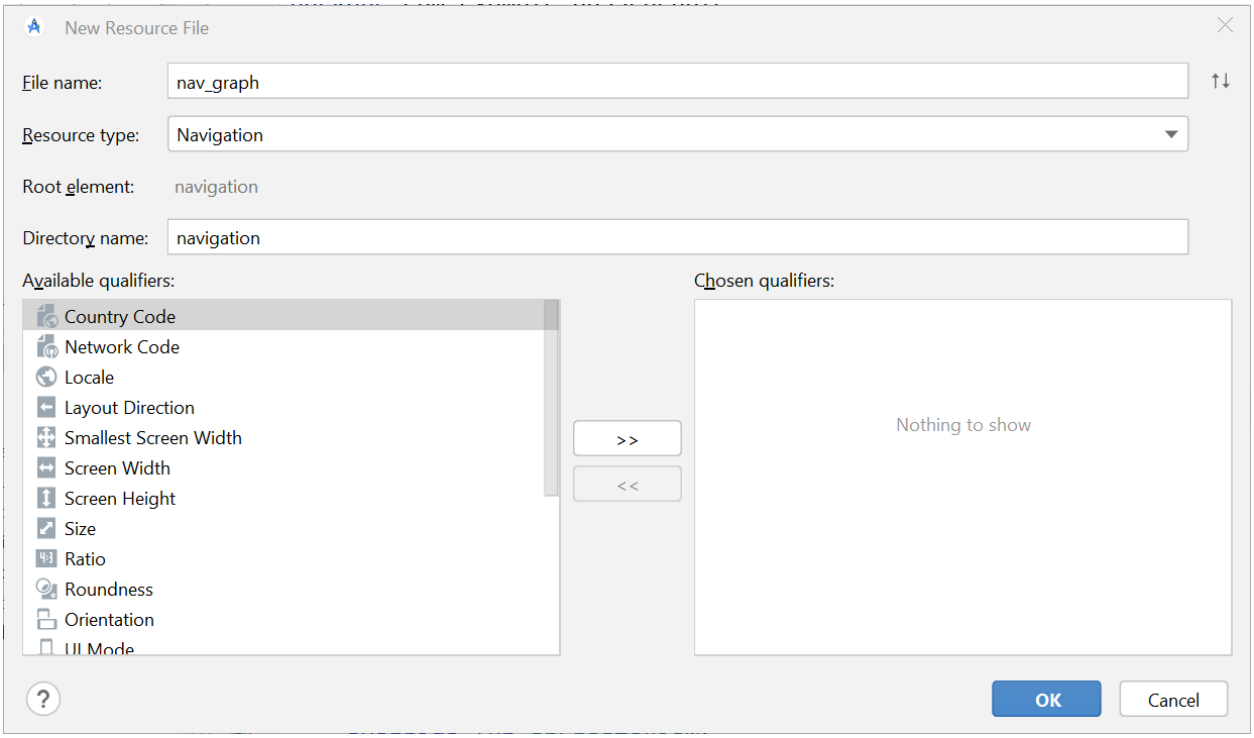
Bước 8. Bổ sung file navigation graph

Ấn chuột phải vào mục res => New => Android Resource File:

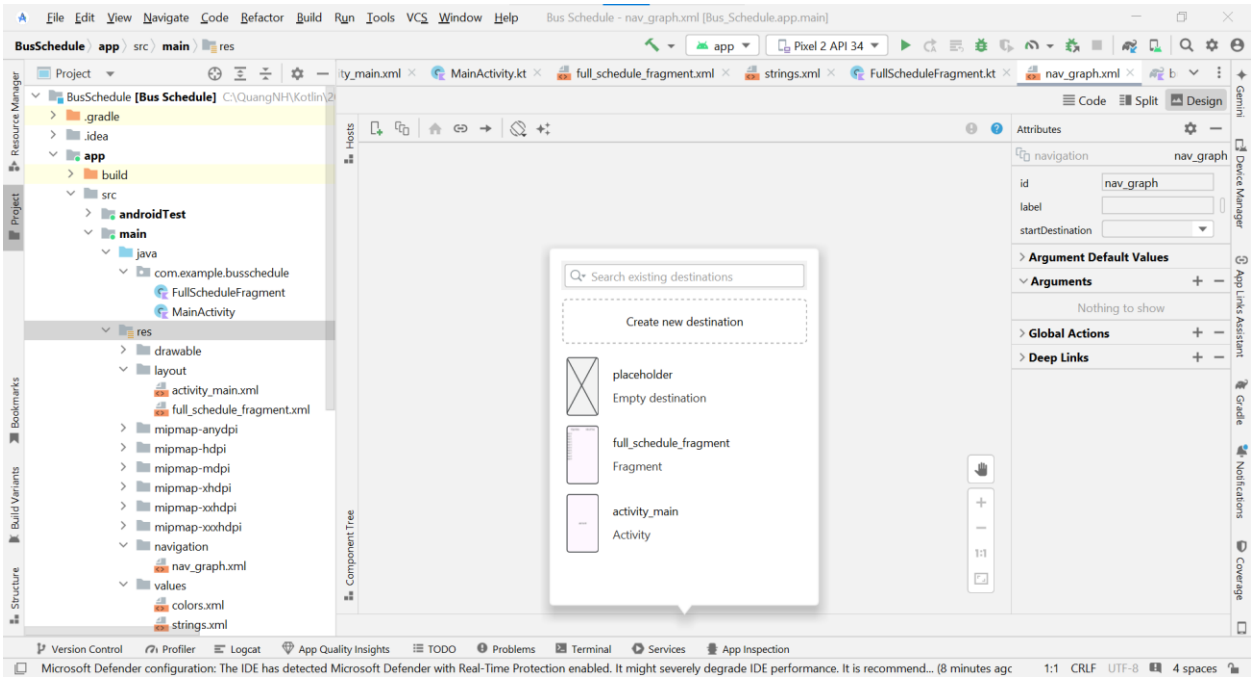


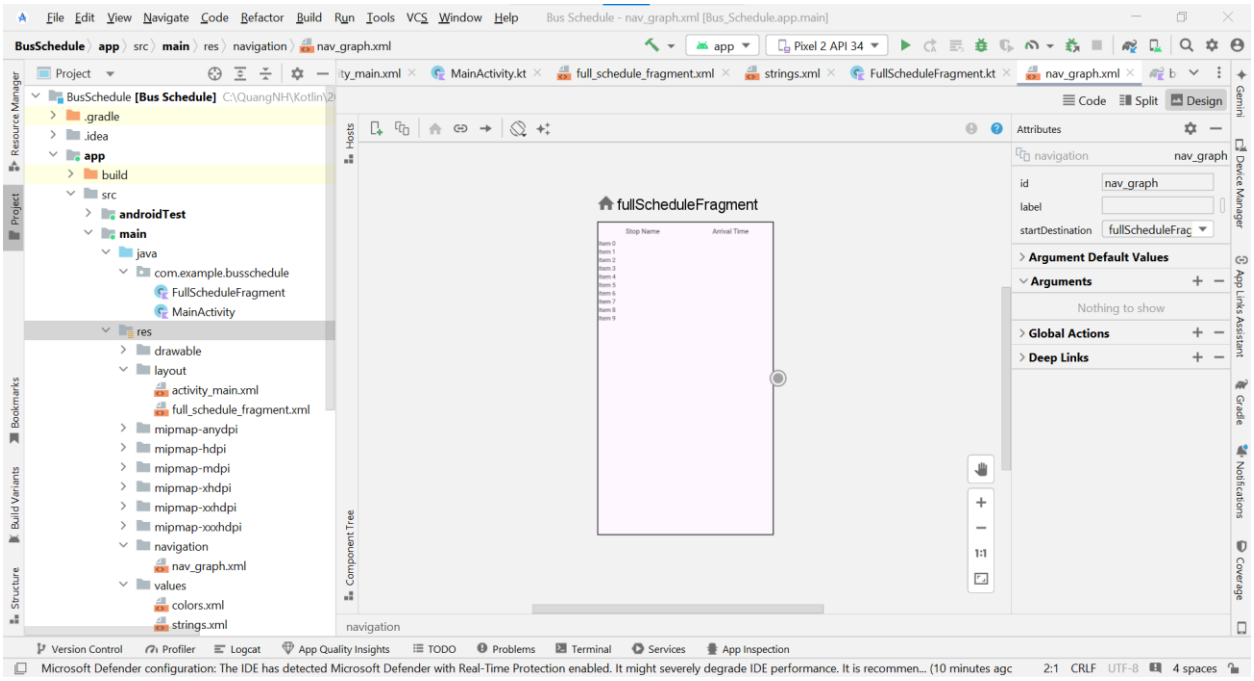
Trong cửa sổ “New Resource File”:

- Trong mục “Resource type”, chọn “Navigation”
- Trong mục “File name”, chọn: nav_graph



Bước 9. Thêm full_schedule_fragment vào navigation





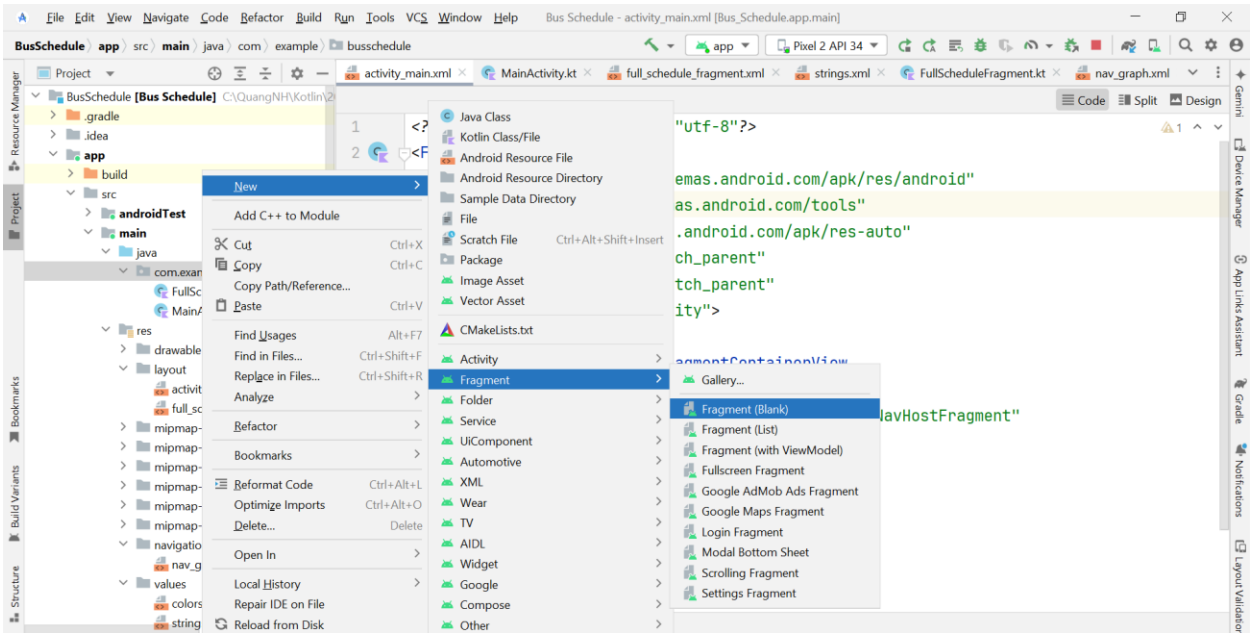
Bước 10. Cập nhật file layout của MainActivity

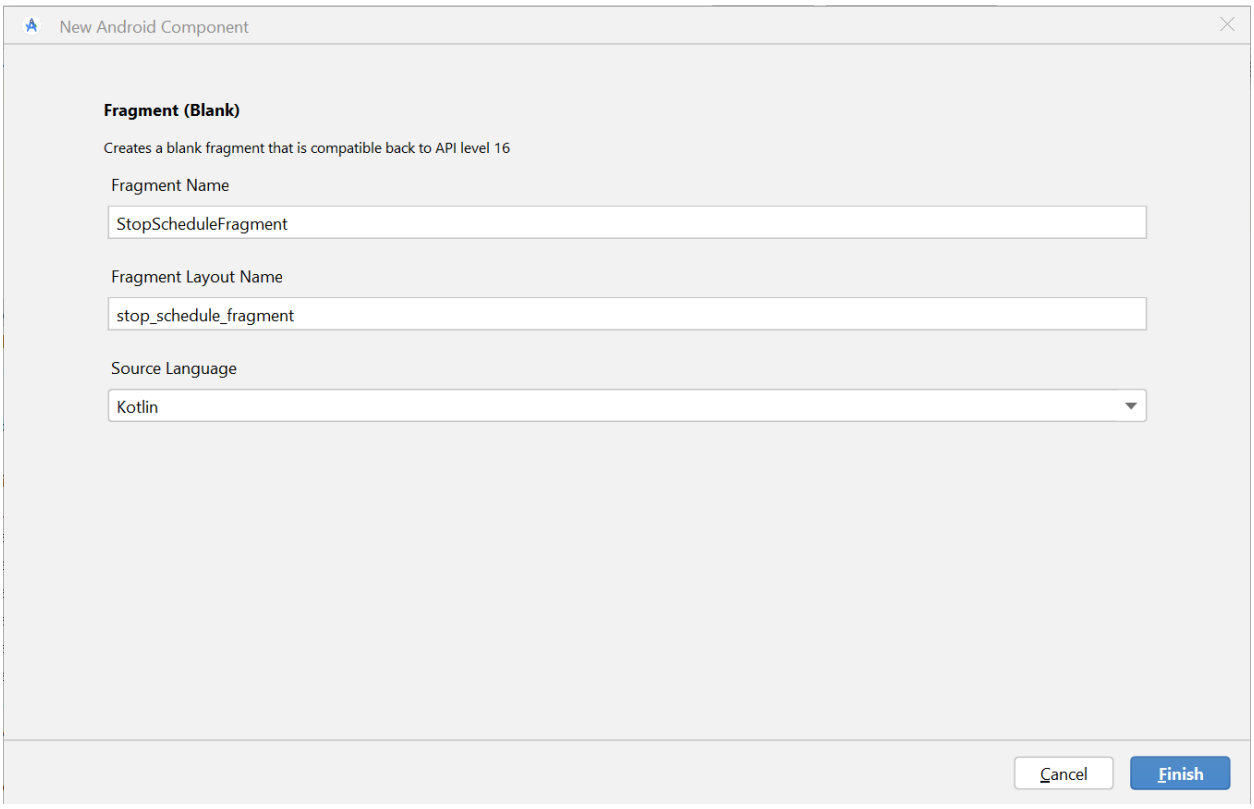
Cập nhật file activity_main.xml

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

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph"/>

</FrameLayout>
```

Chạy chương trình:

Bước 11. Thêm StopScheduleFragment





Bước 12. Cập nhật file StopScheduleFragment.kt

```
package com.example.busschedule

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.example.busschedule.databinding.StopScheduleFragmentBinding

class StopScheduleFragment : Fragment() {
    companion object {
        var STOP_NAME = "stopName"
    }

    private var _binding: StopScheduleFragmentBinding? = null

    private val binding get() = _binding!!

    private lateinit var recyclerView: RecyclerView

    private lateinit var stopName: String

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        arguments?.let {
            stopName = it.getString(STOP_NAME).toString()
        }
    }
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        _binding = StopScheduleFragmentBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        recyclerView = binding.recyclerView
        recyclerView.layoutManager = LinearLayoutManager(requireContext())
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

}
```

Bước 13. Cập nhật file stop_schedule_fragment.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".StopScheduleFragment">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</FrameLayout>
```

Bước 14. Thêm StopScheduleFragment vào navigation graph

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/fullScheduleFragment">

    <fragment
        android:id="@+id/fullScheduleFragment"
        android:name="com.example.busschedule.FullScheduleFragment"
        android:label="full_schedule_fragment"
```

```
        tools:layout="@layout/full_schedule_fragment" >
        <action
            android:id="@+id/action_fullScheduleFragment_to_stopScheduleFragment"
            app:destination="@id/stopScheduleFragment" />
    </fragment>
    <fragment
        android:id="@+id/stopScheduleFragment"
        android:name="com.example.busschedule.StopScheduleFragment"
        android:label="stop_schedule_fragment"
        tools:layout="@layout/stop_schedule_fragment" >
        <argument
            android:name="stopName"
            app:argType="string" />
    </fragment>
</navigation>
```

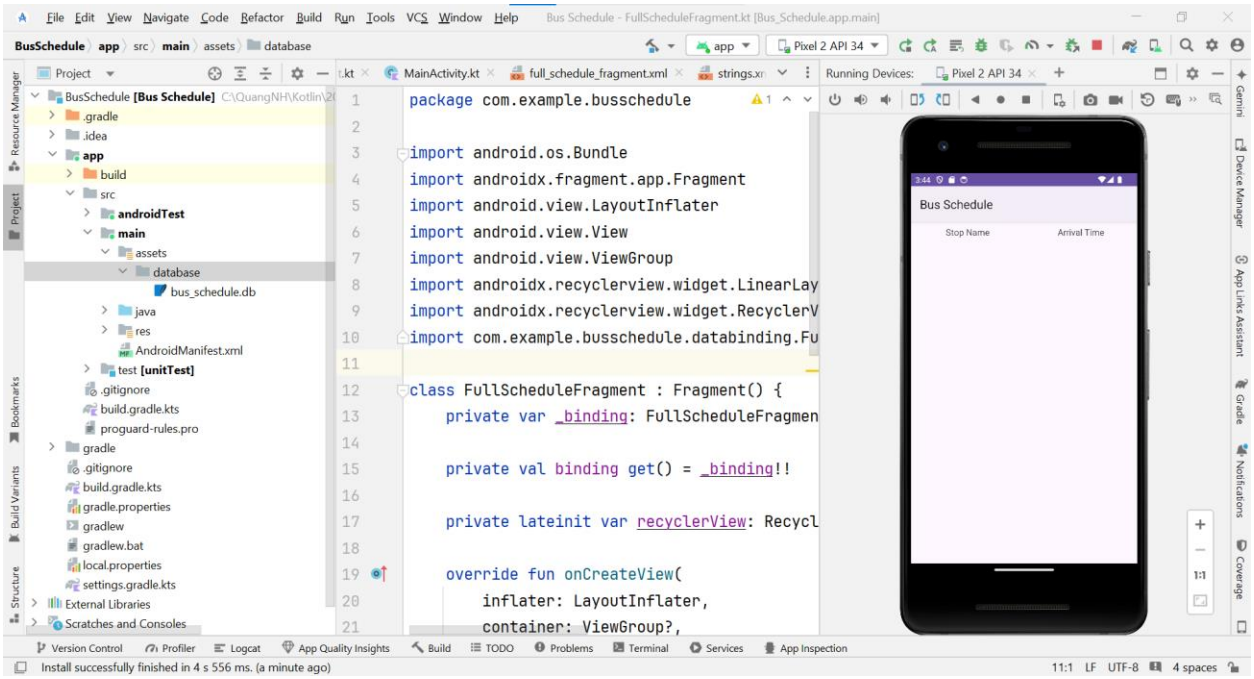
Bước 15. Bổ sung Database vào project

Tạo thư mục assets/database trong thư mục main

Download file bus_schedule.db và copy vào thư mục này

https://github.com/google-developer-training/android-basics-kotlin-bus-schedule-app/blob/starter/app/src/main/assets/database/bus_schedule.db

Bước 16. Kết quả



Introduction to Room and Flow

1. Before you begin

In the previous codelab, you learned about the fundamentals of relational databases, and how to read and write data using the SQL commands: SELECT, INSERT, UPDATE, and DELETE. Learning to work with relational databases is a fundamental skill you'll take with you throughout your programming journey. Knowing how relational databases work is also essential for implementing data persistence in an Android application, which you'll start doing in this lesson.

An easy way to use a database in an Android app is with a library called Room. Room is what's called an ORM (Object Relational Mapping) library, which as the name implies, maps the tables in a relational database to objects usable in Kotlin code. In this lesson, you're just going to focus on reading data. Using a pre-populated database, you'll load data from a table of bus arrival times and present them in a `RecyclerView`.

In the process, you'll learn about the fundamentals of using Room, including the database class, the DAO, entities, and view models. You'll also be introduced to the `ListAdapter` class, another way to present data in a `RecyclerView`, and flow, a Kotlin language feature similar to `LiveData` that will allow your UI to respond to changes in the database.

Prerequisites

- Familiarity with object-oriented programming and using classes, objects and inheritance in Kotlin.
- Basic knowledge of relational databases and SQL taught in the SQL basics codelab.
- Experience using Kotlin coroutines.

What you'll learn

At the end of this lesson, you should be able to

- Represent database tables as Kotlin objects (entities).
- Define the database class to use Room in the app, and pre-populate a database from a file.
- Define the DAO class and use SQL queries to access the database from Kotlin code.
- Define a view model to allow the UI to interact with the DAO.
- How to use ListAdapter with a recycler view.
- The basics of Kotlin flow and how to use it to make the UI respond to changes in the underlying data.

What you'll build

- Read data from a prepopulated database using Room and present it in a recycler view in a simple bus schedule app.

2. Get started

The app you'll be working with in this codelab is called Bus Schedule. The app presents a list of bus stops and arrival times from earliest to latest.

3. Add Room dependency

Like with any other library, you first need to add the necessary dependencies to be able to use Room in the Bus Schedule app. This will require just two small changes, one in each Gradle file.

1. In the project-level `build.gradle` file, define the `room_version` in the ext block.

```
ext {
    kotlin_version = "1.6.20"
    nav_version = "2.4.1"
    room_version = '2.4.2'
}
```

2. In the app-level `build.gradle` file, at the end of the dependencies list, add the following dependencies.

```
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"

// optional - Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:$room_version"
```

3. Sync the changes and build the project to verify the dependencies were added correctly.

Over the next few pages, you'll be introduced to the components needed to integrate Room into an app: models, the DAO, view models, and the database class.

4. Create an entity

When you learned about relational databases in the previous codelab, you saw how data was organized into tables consisting of multiple columns, each one representing a specific property of a specific data type. Much like classes in Kotlin provide a template for each object, a table in a database provides a template for each item, or row, in that table. It should come as no surprise then that a Kotlin class can be used to represent each table in the database.

When working with Room, each table is represented by a class. In an ORM (Object Relational Mapping) library, such as Room, these are often called *model classes*, or *entities*.

The database for the Bus Schedule app just consists of a single table, schedule, which includes some basic information about a bus arrival.

- `id`: An integer providing a unique identifier that serves as the primary key
- `stop_name`: A string

- `arrival_time`: An integer

Note that the SQL types used in the database are actually `INTEGER` for `Int` and `TEXT` for `String`. When working with Room, however, you should only be concerned with the Kotlin types when defining your model classes. Mapping the data types in your model class to the ones used in the database is handled automatically.

When a project has many files, you should consider organizing your files in different packages to provide better access control for each class and to make it easier to locate related classes. To create an entity for the "schedule" table, in the **com.example.busschedule** package, add a new package called **database**. Within that package, add a new package called **schedule**, for your entity. Then in the **database.schedule** package, create a new file called **Schedule.kt** and define a data class called `Schedule`.

```
data class Schedule(
)
```

As discussed in the SQL Basics lesson, data tables should have a primary key to uniquely identify each row. The first property you'll add to the `Schedule` class is an integer to represent a unique id. Add a new property and mark it with the `@PrimaryKey` annotation. This tells Room to treat this property as the primary key when new rows are inserted.

```
@PrimaryKey val id: Int
```

Add a column for the name of the bus stop. The column should be of type `String`. For new columns, you'll need to add a `@ColumnInfo` annotation to specify a name for the column. Typically, SQL column names will have words separated by an underscore, as opposed to the lowerCamelCase used by Kotlin properties. For this column, we also don't want the value to be null , so you should mark it with the `@NonNull` annotation.

```
@NonNull @ColumnInfo(name = "stop_name") val stopName: String,
```

Note: In SQL, columns can have null values by default and need to be explicitly marked as non null if you want otherwise. This is the opposite of how things work in Kotlin, where values can't be null by default.

Arrival times are represented in the database using integers. This is a Unix timestamp that can be converted into a usable date. While different versions of SQL offer ways to convert dates, for your purposes, you'll stick with Kotlin date formatting functions. Add the following `@NonNull` column to the model class.

```
@NonNull @ColumnInfo(name = "arrival_time") val arrivalTime: Int
```

Finally, for Room to recognize this class as something that can be used to define database tables, you need to add an annotation to the class itself. Add `@Entity` on a separate line before the class name.

By default, Room uses the class name as the database table name. Thus, the table name as defined by the class right now would be Schedule. Optionally, you could also specify @Entity(tableName="schedule"), but since Room queries are not case sensitive, you can omit explicitly defining a lowercase table name here.

The class for the schedule entity should now look like the following.

```
@Entity
data class Schedule(
   @PrimaryKey val id: Int,
   @NonNull @ColumnInfo(name = "stop_name") val stopName: String,
   @NonNull @ColumnInfo(name = "arrival_time") val arrivalTime: Int
)
```

5. Define the DAO

The next class you'll need to add to integrate Room is the DAO. DAO stands for Data Access Object and is a Kotlin class that provides access to the data. Specifically, the DAO is where you would include functions for reading and manipulating data. Calling a function on the DAO is the equivalent of performing a SQL command on the database. In-fact, DAO functions like the ones you'll define in this app, often specify a SQL command so you can specify exactly what you want the function to do. Your knowledge of SQL from the previous codelab will come in handy when defining the DAO.

1. Add a DAO class for the Schedule entity. In the **database.schedule** package, create a new file called `ScheduleDao.kt` and define an interface called `ScheduleDao`. Similar to the `Schedule` class, you need to add an annotation, this time `@Dao`, to make the interface usable with Room.

```
@Dao
interface ScheduleDao {
}
```

Note: While DAO is an acronym, naming conventions for Kotlin code only capitalize the first letter in acronyms, thus the name `ScheduleDao` and not `ScheduleDAO`.

2. There are two screens in the app and each will need a different query. The first screen shows all the bus stops in ascending order by arrival time. In this use case, the query just needs to get all columns and include an appropriate `ORDER BY` clause. The query is specified as a string passed into a `@Query` annotation. Define a function `getAll()` that returns a List of `Schedule` objects including the `@Query` annotation as shown.

```
@Query("SELECT * FROM schedule ORDER BY arrival_time ASC")
fun getAll(): List<Schedule>
```

3. For the second query, you also want to select all columns from the schedule table. However, you only want results that match the selected stop name, so you need to add a

`WHERE` clause. You can reference Kotlin values from the query by preceding it with a colon (`:`) (e.g. `:stopName` from the function parameter). Like before, the results are ordered in ascending order by arrival time. Define a `getByStopName()` function that takes a `String` parameter called `stopName` and returns a `List` of `Schedule` objects, with a `@Query` annotation as shown.

```
@Query("SELECT * FROM schedule WHERE stop_name = :stopName ORDER BY
arrival_time ASC")
fun getByStopName(stopName: String): List<Schedule>
```

6. Define the ViewModel

Now that you've set up the DAO, you technically have everything you need to start accessing the database from your fragments. However, while this works in theory, it's generally not considered best practice. The reason is that in more complex apps, you likely have multiple screens that access only a specific portion of the data. While `ScheduleDao` is relatively simple, it's easy to see how this can get out of hand when working with two or more different screens. For example, a DAO might look something like this:

```
@Dao
interface ScheduleDao {

    @Query(...)
    getForScreenOne() ...

    @Query(...)
    getForScreenTwo() ...

    @Query(...)
    getForScreenThree()

}
```

While the code for Screen 1 can access `getForScreenOne()`, there's no good reason for it to access the other methods. Instead, it's considered best practice to separate the part of the DAO you expose to the view into a separate class called a *view model*. This is a common architectural pattern in mobile apps. Using a view model helps enforce a clear separation between the code for your app's UI and its data model. It also helps with testing each part of your code independently, a topic you'll explore further as you continue your Android development journey.

to be recreated. This is not possible with accessing a DAO class directly, so it's best practice to use `ViewModel` subclass to separate the responsibility of loading data from your activity or fragment.

Note: Bus Schedule is a relatively simple app and only includes two screens of mostly identical content. For teaching purposes, we'll be creating a single view model class that can be used by both screens, but in a larger app, you may want to use a separate view model for each fragment.

1. To create a view model class, create a new file called **BusScheduleViewModel.kt** in a new package called **viewmodels**. Define a class for the view model. It should take a single parameter of type `ScheduleDao`.

```
class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel()
{
```

2. Since this view model will be used with both screens, you'll need to add a method to get the full schedule as well as a filtered schedule by stop name. You can do this by calling the corresponding methods from `ScheduleDao`.

```
fun fullSchedule(): List<Schedule> = scheduleDao.getAll()

fun scheduleForStopName(name: String): List<Schedule> =
scheduleDao.getByStopName(name)
```

Although you've finished defining the view model, you can't just instantiate a `BusScheduleViewModel` directly and expect everything to work. As the ViewModel class `BusScheduleViewModel` is meant to be lifecycle aware, it should be instantiated by an object that can respond to lifecycle events. If you instantiate it directly in one of your fragments, then your fragment object will have to handle everything, including all the memory management, which is beyond the scope of what your app's code should do. Instead, you can create a class, called a factory, that will instantiate view model objects for you.

1. To create a factory, below the view model class, create a new class `BusScheduleViewModelFactory`, that inherits from `ViewModelProvider.Factory`.

```
class BusScheduleViewModelFactory(
   private val scheduleDao: ScheduleDao
) : ViewModelProvider.Factory {
}
```

2. You'll just need a bit of boilerplate code to correctly instantiate a view model. Instead of initializing the class directly, you'll override a method called `create()` that returns a `BusScheduleViewModelFactory` with some error checking. Implement the `create()` inside the `BusScheduleViewModelFactory` class as follows.

```
override fun <T : ViewModel> create(modelClass: Class<T>): T {
       if (modelClass.isAssignableFrom(BusScheduleViewModel::class.java)) {
           @Suppress("UNCHECKED_CAST")
           return BusScheduleViewModel(scheduleDao) as T
```

```
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
```

You can now instantiate a `BusScheduleViewModelFactory` object with `BusScheduleViewModelFactory.create()`, so that your view model can be lifecycle aware without your fragment having to handle this directly.

7. Create database class and pre-populate database

Now that you've defined the models, DAO, and a view model for fragments to access the DAO, you still need to tell Room what to do with all of these classes. That's where the `AppDatabase` class comes in. An Android app using Room, such as yours, subclasses the `RoomDatabase` class and has a few key responsibilities. In your app, the `AppDatabase` needs to

1. Specify which entities are defined in the database.
2. Provide access to a single instance of each DAO class.
3. Perform any additional setup, such as pre-populating the database.

While you may be wondering why Room can't just find all the entities and DAO objects for you, it's quite possible that your app could have multiple databases, or any number of scenarios where the library can't assume the intent of you, the developer. The `AppDatabase` class gives you complete control over your models, DAO classes, and any database setup you wish to perform.

1. To add an `AppDatabase` class, in the **database** package, create a new file called `AppDatabase.kt`, and define a new abstract class `AppDatabase` that inherits from `RoomDatabase`.

```
abstract class AppDatabase: RoomDatabase() {
}
```

2. The database class allows other classes easy access to the DAO classes. Add an abstract function that returns a `ScheduleDao`.

```
abstract fun scheduleDao(): ScheduleDao
```

3. When using an `AppDatabase` class, you want to ensure that only one instance of the database exists to prevent race conditions or other potential issues. The instance is stored in the companion object, and you'll also need a method that either returns the existing instance, or creates the database for the first time. This is defined in the companion object. Add the following `companion object` just below the `scheduleDao()` function.

```
companion object {
}
```

In the `companion object`, add a property called `INSTANCE` of type `AppDatabase`. This value is initially set to `null`, so the type is marked with a `?`. This is also marked with a `@Volatile` annotation. While the details about when to use a volatile property are a bit advanced for this lesson, you'll want to use it for your `AppDatabase` instance to avoid potential bugs.

```
@Volatile
private var INSTANCE: AppDatabase? = null
```

Below the `INSTANCE` property, define a function to return the `AppDatabase` instance:

```
fun getDatabase(context: Context): AppDatabase {
    return INSTANCE ?: synchronized(this) {
        val instance = Room.databaseBuilder(
            context,
            AppDatabase::class.java,
            "app_database")
            .createFromAsset("database/bus_schedule.db")
            .build()
        INSTANCE = instance

        instance
    }
}
```

In the implementation for `getDatabase()`, you use the Elvis operator to either return the existing instance of the database (if it already exists) or create the database for the first time if needed. In this app, since the data is prepopulated. You also call `createFromAsset()` to load the existing data. The `bus_schedule.db` file can be found in the `assets.database` package in your project.

4. Just like the model classes and DAO, the database class requires an annotation providing some specific information. All the entity types (you access the type itself using `ClassName::class`) are listed in an array. The database is also given a version number, which you'll set to 1. Add the `@Database` annotation as follows.

```
@Database(entities = arrayOf(Schedule::class), version = 1)
```

Note: The version number is incremented each time you make a schema change. The app checks this version with the one in the database to determine if and how a migration should be performed.

Now that you've created your `AppDatabase` class, there's just one more step to make it usable. You'll need to provide a custom subclass of the `Application` class, and create a `lazy` property that will hold the result of `getDatabase()`.

5. In the **com.example.busschedule** package, add a new file called `BusScheduleApplication.kt`, and create a `BusScheduleApplication` class that inherits from `Application`.

```
class BusScheduleApplication : Application() {
}
```

6. Add a database property of type `AppDatabase`. The property should be lazy and return the result of calling `getDatabase()` on your `AppDatabase` class.

```
class BusScheduleApplication : Application() {
   val database: AppDatabase by lazy { AppDatabase.getDatabase(this) }
```

7. Finally, to make sure that `BusScheduleApplication` class is used (instead of the default base class `Application`), you need to make a small change to the manifest. In `AndroidMainifest.xml`, set the `android:name` property to `com.example.busschedule.BusScheduleApplication`.

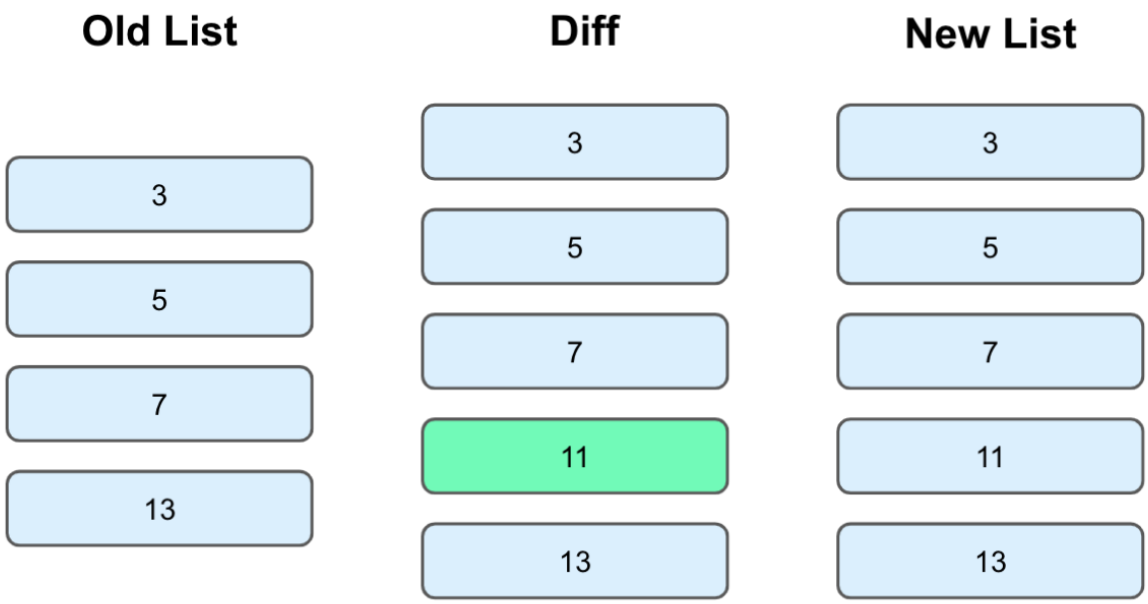
```
<application
    android:name="com.example.busschedule.BusScheduleApplication"
    ...
```

That's it for setting up your app's model. You're all set to start using data from Room in your UI. On the next few pages, you'll create a `ListAdapter` for your app's `RecyclerView` to present the bus schedule data and respond to data changes dynamically.

8. Create the ListAdapter

It's time to take all that hard work and hook up the model to the view. Previously, when using a `RecyclerView`, you would use a `RecyclerView.Adapter` to present a static list of data. While this will certainly work for an app like Bus Schedule, a common scenario when working with databases is to handle changes to the data in real time. Even if only one item's contents change, the entire recycler view is refreshed. This won't be sufficient for the majority of apps using persistence.

An alternative for a dynamically changing list is called `ListAdapter`. `ListAdapter` uses AsyncListDiffer to determine the differences between an old list of data and a new list of data. Then, the recycler view is only updated based on the differences between the two lists. The result is that your recycler view is more performant when handling frequently updated data, as you'll often have in a database application.



Because the UI is identical for both screens, you'll just need to create a single `ListAdapter` that can be used with both screens.

1. Create a new file `BusStopAdapter.kt` and a `BusStopAdapter` class as shown. The class extends a generic `ListAdapter` that takes a list of `Schedule` objects and a `BusStopViewHolder` class for the UI. For the `BusStopViewHolder`, you also pass in a `DiffCallback` type which you'll define soon. The `BusStopAdapter` class itself also takes a parameter, `onItemClicked()`. This function will be used to handle navigation when an item is selected on the first screen, but for the second screen, you'll just pass in an empty function.

```
class BusStopAdapter(private val onItemClicked: (Schedule) -> Unit) :
ListAdapter<Schedule, BusStopAdapter.BusStopViewHolder>(DiffCallback) {
}
```

2. Similar to a recycler view adapter, you need a view holder so that you can access views created from your layout file in code. The layout for the cells is already created. Simply, create a `BusStopViewHolder` class as shown and implement the `bind()` function to set `stopNameTextView`'s text to the stop name and the `arrivalTimeTextView`'s text to the formatted date.

```
class BusStopViewHolder(private var binding: BusStopItemBinding):
RecyclerView.ViewHolder(binding.root) {
    @SuppressLint("SimpleDateFormat")
    fun bind(schedule: Schedule) {
        binding.stopNameTextView.text = schedule.stopName
        binding.arrivalTimeTextView.text = SimpleDateFormat(
            "h:mm a").format(Date(schedule.arrivalTime.toLong() * 1000)
        )
```

```
    }
}
```

3. Override and implement `onCreateViewHolder()` and inflate the layout and set the `onClickListener()` to call `onItemClicked()` for the item at the current position.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
BusStopViewHolder {
   val viewHolder = BusStopViewHolder(
       BusStopItemBinding.inflate(
           LayoutInflater.from( parent.context),
           parent,
           false
       )
   )
   viewHolder.itemView.setOnClickListener {
       val position = viewHolder.adapterPosition
       onItemClicked(getItem(position))
   }
   return viewHolder
}
```

4. Override and implement `onBindViewHolder()` and to bind the view at the specified position.

```
override fun onBindViewHolder(holder: BusStopViewHolder, position: Int) {
   holder.bind(getItem(position))
}
```

5. Remember that `DiffCallback` class you specified for the `ListAdapter`? This is just an object that helps the `ListAdapter` determine which items in the new and old lists are different when updating the list. There are two methods: `areItemsTheSame()` checks if the object (or row in the database in your case) is the same by only checking the ID. `areContentsTheSame()` checks if all properties, not just the ID, are the same. These methods allow the `ListAdapter` to determine which items have been inserted, updated, and deleted so that the UI can be updated accordingly.

Add a companion object and implement `DiffCallback` as shown.

```
companion object {
   private val DiffCallback = object : DiffUtil.ItemCallback<Schedule>() {
       override fun areItemsTheSame(oldItem: Schedule, newItem: Schedule):
Boolean {
           return oldItem.id == newItem.id
       }

       override fun areContentsTheSame(oldItem: Schedule, newItem: Schedule):
Boolean {
           return oldItem == newItem
       }
   }
}
```

That's all there is to setting up the adapter. You'll use it in both screens of the app.

1. First, in `FullScheduleFragment.kt`, you need to get a reference to the view model.

```
private val viewModel: BusScheduleViewModel by activityViewModels {
   BusScheduleViewModelFactory(
       (activity?.application as
BusScheduleApplication).database.scheduleDao()
   )
}
```

2. Then in `onViewCreated()`, add the following code to set up the recycler view and assign its layout manager.

```
recyclerView = binding.recyclerView
recyclerView.layoutManager = LinearLayoutManager(requireContext())
```

3. Then assign the adapter property. The action passed in will use the `stopName` to navigate the selected next screen so that the list of bus stops can be filtered.

```
val busStopAdapter = BusStopAdapter({
   val action =
FullScheduleFragmentDirections.actionFullScheduleFragmentToStopScheduleFragme
nt(
       stopName = it.stopName
   )
   view.findNavController().navigate(action)
})
recyclerView.adapter = busStopAdapter
```

4. Finally, to update a list view, call `submitList()`, passing in the list of bus stops from the view model.

```
// submitList() is a call that accesses the database. To prevent the
// call from potentially locking the UI, you should use a
// coroutine scope to launch the function. Using GlobalScope is not
// best practice, and in the next step we'll see how to improve this.
GlobalScope.launch(Dispatchers.IO) {
   busStopAdapter.submitList(viewModel.fullSchedule())
}
```

5. Do the same in `StopScheduleFragment`. First, get a reference to the view model.

```
private val viewModel: BusScheduleViewModel by activityViewModels {
   BusScheduleViewModelFactory(
       (activity?.application as
BusScheduleApplication).database.scheduleDao()
   )
}
```

6. Then configure the recycler view in `onViewCreated()`. This time you just need to pass in an empty block (function) with `{}`. You don't actually want anything to happen when rows on this screen are tapped.

```
recyclerView = binding.recyclerView
recyclerView.layoutManager = LinearLayoutManager(requireContext())
val busStopAdapter = BusStopAdapter({})
recyclerView.adapter = busStopAdapter
// submitList() is a call that accesses the database. To prevent the
// call from potentially locking the UI, you should use a
// coroutine scope to launch the function. Using GlobalScope is not
// best practice, and in the next step we'll see how to improve this.
GlobalScope.launch(Dispatchers.IO) {
   busStopAdapter.submitList(viewModel.scheduleForStopName(stopName))
}
```

7. Now that you've set up the adapter, you're done integrating Room into the Bus Schedule app. Take a moment to run the app and you should see a list of arrival times. Tapping on a row should navigate to the detail screen.

9. Respond to data changes using Flow

While your list view is set up to efficiently handle data changes whenever `submitList()` is called, your app won't be able to handle dynamic updates just yet. To see for yourself, try opening the Database Inspector and running the following query to insert a new item into the schedule table.

```
INSERT INTO schedule
VALUES (null, 'Winding Way', 1617202500)
```

You'll notice that in the emulator, however, nothing happens. The user is going to assume that the data is unchanged. You'll need to re-run your app in order to see the changes.

The problem is that the `List<Schedule>` is returned from each of the DAO functions only once. Even if the underlying data is updated, `submitList()` won't be called to update the UI, and from the user's perspective, it will look like nothing has changed.

To fix this, you can take advantage of a Kotlin feature called *asynchronous flow* (often just called *flow*) that will allow the DAO to continuously emit data from the database. If an item is inserted, updated, or deleted, the result will be sent back to the fragment. Using a function called `collect()`, you can call `submitList()` using the new value emitted from the flow so that your `ListAdapter` can update the UI based on the new data.

1. To use flow in Bus Schedule, open up `ScheduleDao.kt`. To convert the DAO functions to return a `Flow`, simply change the return type of the `getAll()` function to `Flow<List<Schedule>>`.

```
fun getAll(): Flow<List<Schedule>>
```

2. Likewise, update the return value of the `getByStopName()` function.

```
fun getByStopName(stopName: String): Flow<List<Schedule>>
```

3. The functions in the view model that access the DAO also need to be updated. Update the return values to `Flow<List<Schedule>>` for both `fullSchedule()` and `scheduleForStopName()`.

```
class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel()
{

   fun fullSchedule(): Flow<List<Schedule>> = scheduleDao.getAll()

   fun scheduleForStopName(name: String): Flow<List<Schedule>> =
scheduleDao.getByStopName(name)
}
```

4. Finally, in `FullScheduleFragment.kt`, the `busStopAdapter` should be updated when you call `collect()` on the query results. Because `fullSchedule()` is a suspend function, it needs to be called from a coroutine. Replace the line.

```
busStopAdapter.submitList(viewModel.fullSchedule())
```

With this code that uses the flow returned from `fullSchedule()`.

```
lifecycle.coroutineScope.launch {
   viewModel.fullSchedule().collect() {
       busStopAdapter.submitList(it)
   }
}
```

5. Do the same in `StopScheduleFragment`, but replace the call to `scheduleForStopName()`, with the following.

```
lifecycle.coroutineScope.launch {
   viewModel.scheduleForStopName(stopName).collect() {
       busStopAdapter.submitList(it)
   }
}
```

6. Once you've made the above changes, you can re-run the app to verify that data changes are now handled in real time. Once the app is running, return to the Database Inspector, and send the following query to insert a new arrival time before 8:00 AM.

```
INSERT INTO schedule
VALUES (null, 'Winding Way', 1617202500)
```

The new item should appear at the top of the list.

That's it for the Bus Schedule app. Great job making it this far. You should now have a solid foundation in working with Room. In the next pathway, you'll dive deeper into Room with a new sample app and learn how to save user-created data on a device.

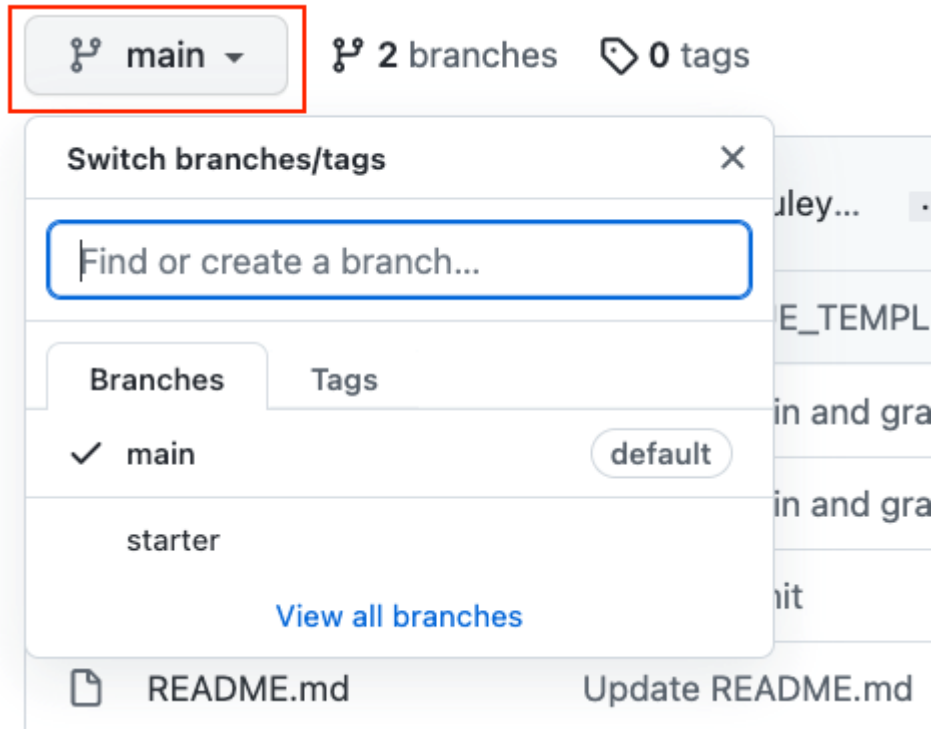
10. Solution code

The solution code for this codelab is in the project and module shown below.

Solution Code URL: https://github.com/google-developer-training/android-basics-kotlin-bus-schedule-app

Branch name: `main`

1. Navigate to the provided GitHub repository page for the project.
2. Verify that the branch name matches the branch name specified in the codelab. For example, in the following screenshot the branch name is **main**.



3. On the GitHub page for the project, click the **Code** button, which brings up a popup.

Lab 9.3 Make Starter Project

Link:

https://github.com/google-developer-training/android-basics-kotlin-inventory-app/tree/starter

Bước 1. Thiết lập phiên bản SDK 35

```
android {
    namespace = "com.example.inventory"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.example.inventory"
        minSdk = 26
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"
```

Bước 2. Thiết lập các thư viện cần thiết

Cập nhật file project/ build.gradle.kts

// Top-level build file where you can add configuration options common to all sub-projects/modules.

```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
   id("com.android.library") version "8.1.1" apply false
}
```

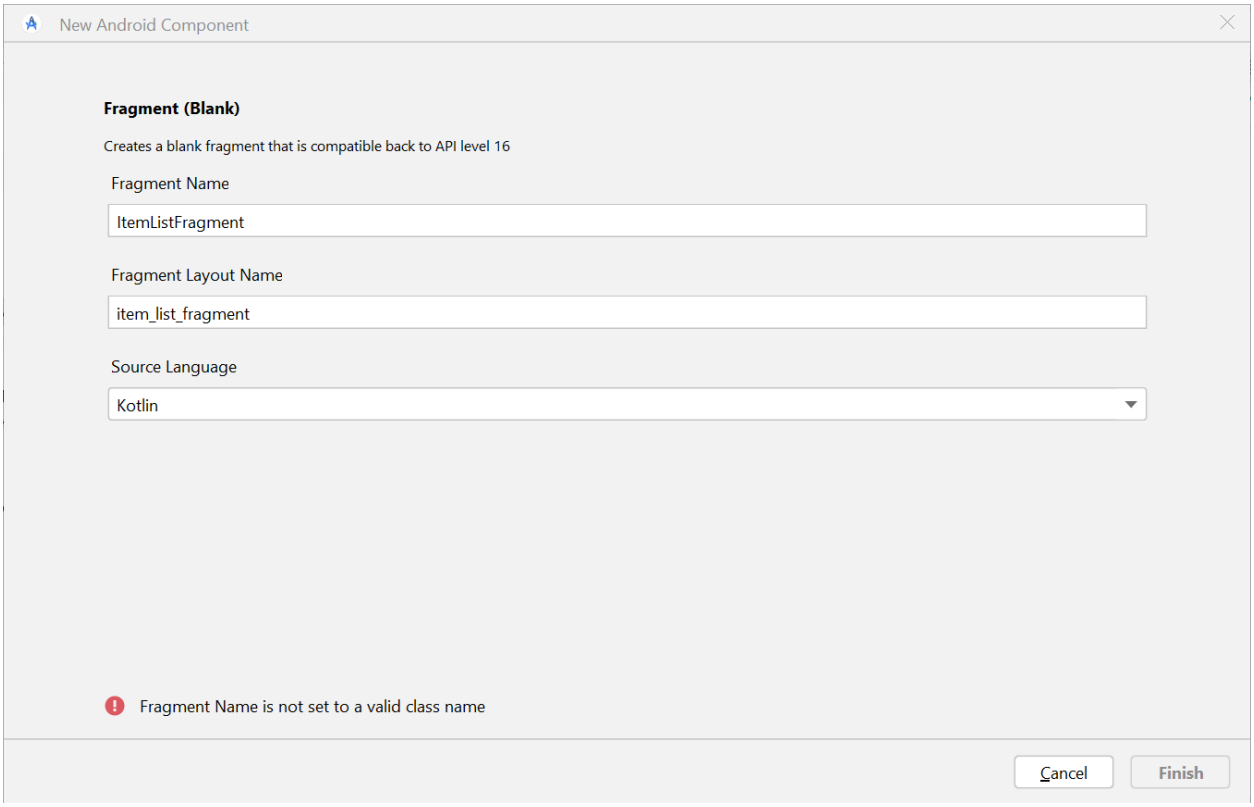
Cập nhật file app/build.gradle.kts

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("androidx.navigation.safeargs.kotlin")
```

```
    id("kotlin-kapt")
}

android {
    namespace = "com.example.inventory"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.example.inventory"
        minSdk = 26
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = "1.8"
    }

    buildFeatures {
        viewBinding = true
    }
}

dependencies {
    val room_version = "2.6.1"

    implementation("androidx.room:room-runtime:$room_version")
```

```
    kapt("androidx.room:room-compiler:$room_version")
    implementation("androidx.room:room-ktx:$room_version")

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.activity)
    implementation(libs.androidx.constraintlayout)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)

    implementation("androidx.navigation:navigation-fragment-ktx:2.8.3")
    implementation("androidx.navigation:navigation-ui-ktx:2.8.3")
}
```

Bước 3. Thêm ItemListFragment



Bước 4. Cập nhật giao diện của ItemListFragment

Mở file item_list_fragment.xml

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="@dimen/margin"
    tools:context=".ItemListFragment">

    <TextView
        android:id="@+id/item_name"
        style="@style/Widget.Inventory.Header"
        android:layout_marginStart="@dimen/margin_between_elements"
        android:text="@string/item"
        app:layout_constraintEnd_toStartOf="@+id/item_price"
        app:layout_constraintHorizontal_weight="2"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/item_price"
        style="@style/Widget.Inventory.Header"
        android:layout_below="@+id/item_name"
        android:layout_marginStart="@dimen/margin_between_elements"
        android:text="@string/price"
        android:textAlignment="center"
        app:layout_constraintEnd_toStartOf="@+id/item_quantity"
        app:layout_constraintHorizontal_weight="1"
```

```
    app:layout_constraintStart_toEndOf="@+id/item_name"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/item_quantity"
    style="@style/Widget.Inventory.Header"
    android:layout_alignParentEnd="true"
    android:layout_marginEnd="@dimen/margin_between_elements"
    android:text="@string/quantity_in_stock"
    android:textAlignment="center"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_weight="1"
    app:layout_constraintStart_toEndOf="@+id/item_price"
    app:layout_constraintTop_toTopOf="parent" />

<View
    android:id="@+id/divider"
    style="@style/Divider"
    android:layout_marginTop="@dimen/margin_between_elements"
    app:layout_constraintBottom_toTopOf="@+id/recyclerView"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/item_quantity" />

<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
```

```
        android:scrollbars="vertical"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/divider" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/floatingActionButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="@dimen/margin_between_elements"
        android:layout_marginBottom="@dimen/margin_between_elements"
        android:contentDescription="@string/add_new_item"
        android:src="@android:drawable/ic_input_add"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:tint="@android:color/white" />

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

Bước 5. Cập nhật file res/values/strings.xml

```
<string name="sell">Sell</string>
    <string name="quantity">Quantity in Stock:</string>
    <string name="quantity_req">Quantity in Stock <font color='#FF0000'>*</font></string>
    <string name="item_name_req">Item Name <font color='#FF0000'>*</font></string>
    <string name="item_price_req">Item Price <font color='#FF0000'>*</font></string>
    <string name="save_action">Save</string>
    <string name="item_detail_fragment_title">Item Details</string>
    <string name="add_fragment_title">Add Item</string>
```

```
    <string name="edit_fragment_title">Edit Item</string>
    <string name="price">Price</string>
    <string name="quantity_in_stock">Quantity\nIn Stock</string>
    <string name="item">Item</string>
    <string name="delete">Delete</string>
    <string name="delete_question">Are you sure you want to delete?</string>
    <string name="no">No</string>
    <string name="yes">Yes</string>
    <string name="currency_symbol">$</string>
    <string name="edit_item">Edit Item</string>
    <string name="add_new_item">Add new item</string>
```

Bước 6. Bổ sung file res/values/styles.xml

```
<style name="Widget.Inventory.ListItemTextView"
parent="Widget.MaterialComponents.TextView">
        <item name="android:gravity">center_vertical</item>
        <item name="android:layout_height">48dp</item>
        <item name="android:textAppearance">?attr/textAppearanceBody1</item>
    </style>

    <style name="Widget.Inventory.TextView"
parent="Widget.MaterialComponents.TextView">
        <item name="android:textSize">16sp</item>
        <item name="android:layout_height">wrap_content</item>
        <item
name="android:textAppearance">@style/TextAppearance.AppCompat.Body1</item>
    </style>

    <style name="Widget.Inventory.Header" parent="Widget.MaterialComponents.TextView">
```

```
        <item name="android:gravity">center_vertical</item>
        <item name="android:textSize">14sp</item>
        <item name="android:layout_marginTop">8dp</item>
        <item name="android:layout_width">0dp</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textAppearance">?attr/textAppearanceOverline</item>
    </style>

    <style name="Divider">
        <item name="android:layout_width">match_parent</item>
        <item name="android:layout_height">1dp</item>
        <item name="android:background">?android:attr/listDivider</item>
    </style>

    <style name="Widget.Inventory.TextInputLayout.OutlinedBox"
        parent="Widget.MaterialComponents.TextInputLayout.OutlinedBox">
        <item name="boxStrokeErrorColor">@color/red_700</item>
        <item name="errorIconTint">@color/red_700</item>
        <item name="errorTextColor">@color/red_700</item>
        <item name="errorIconDrawable">@android:drawable/stat_notify_error</item>
        <item
name="helperTextTextAppearance">@style/TextAppearance.MaterialComponents.Subtitle1
        </item>
    </style>
```

Bước 7. Bổ sung file dimens.xml và cập nhật file colors.xml

```
<dimen name="margin">16dp</dimen>
    <dimen name="margin_between_elements">8dp</dimen>
```

Cập nhật file colors.xml

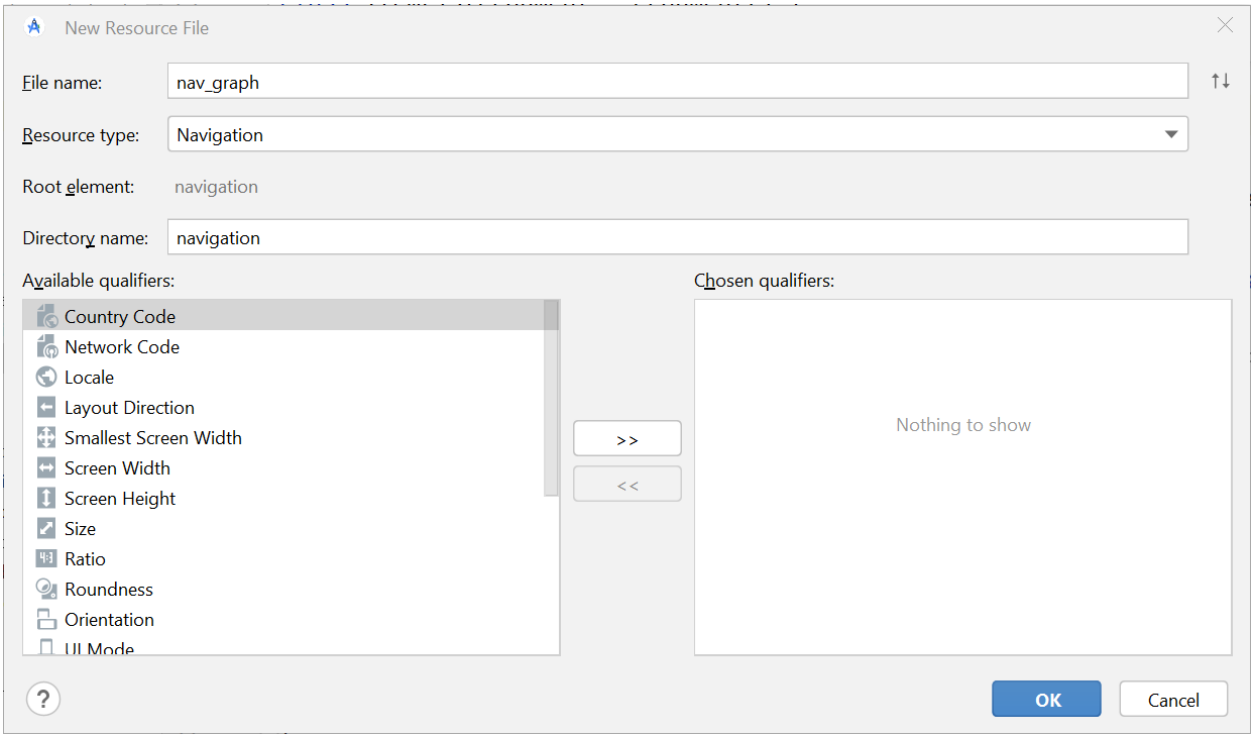
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="red_700">#FFD32F2F</color>
</resources>
```

Bước 8. Cập nhật mã nguồn của ItemListFragment

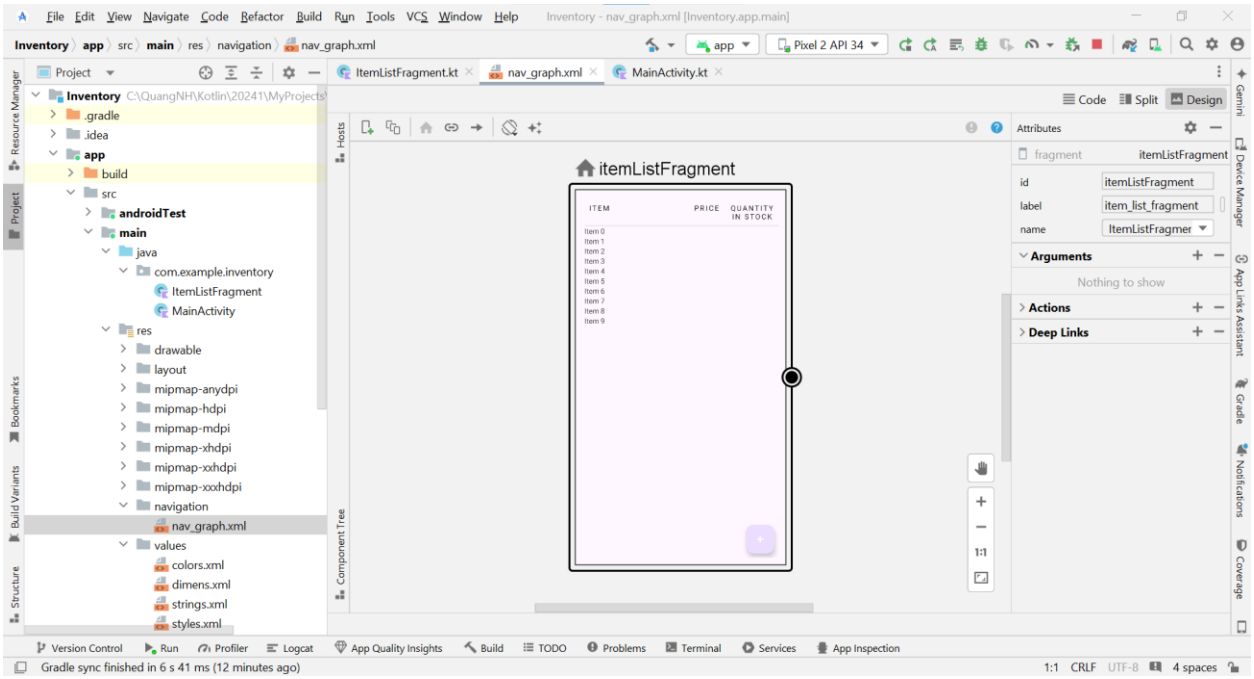
```
package com.example.inventory


import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import androidx.navigation.fragment.findNavController

import androidx.recyclerview.widget.LinearLayoutManager

import com.example.inventory.databinding.ItemListFragmentBinding



class ItemListFragment : Fragment() {

    private var _binding: ItemListFragmentBinding? = null

    private val binding get() = _binding!!
```

```
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        _binding = ItemListFragmentBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.recyclerView.layoutManager = LinearLayoutManager(this.context)
        binding.floatingActionButton.setOnClickListener {
            /*val action = ItemListFragmentDirections.actionItemListFragmentToAddItemFragment(
                getString(R.string.add_fragment_title)
            )
            this.findNavController().navigate(action)
            */
        }
    }
}
```

Bước 9. Tạo file res/navigation/nav_graph.xml



Thêm ItemListFragment vào nav_graph.xml

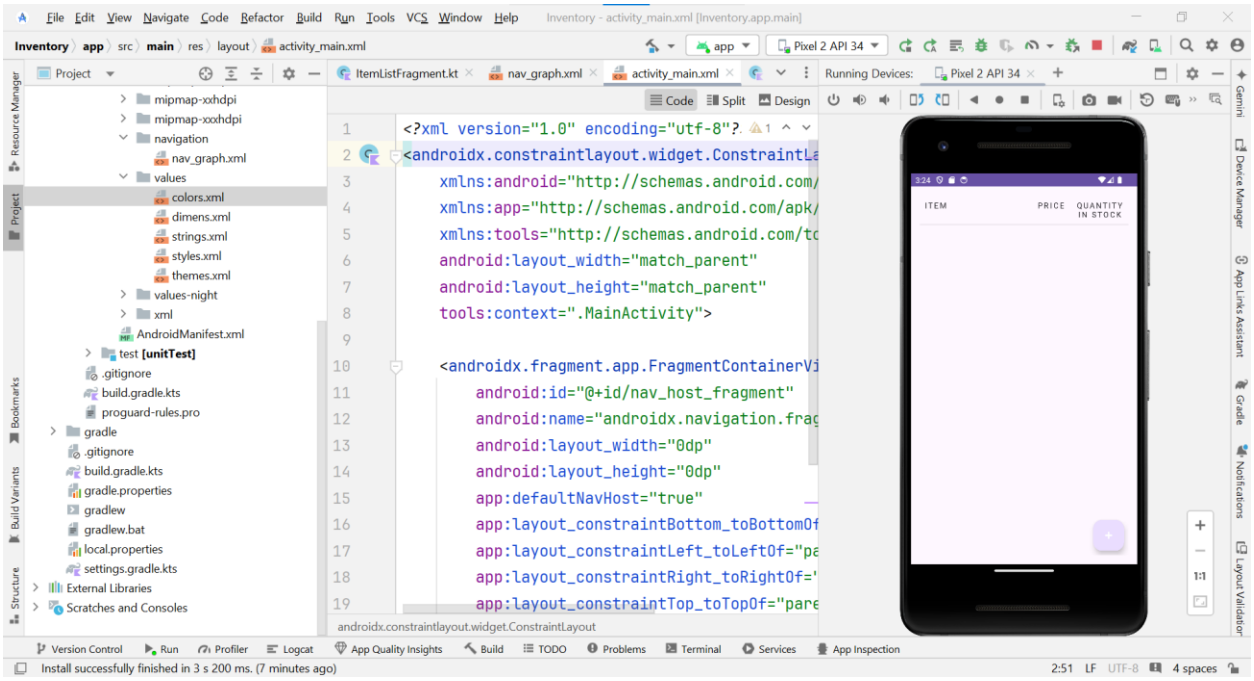


Bước 10. Bổ sung NavHostFragment vào activity_main.xml

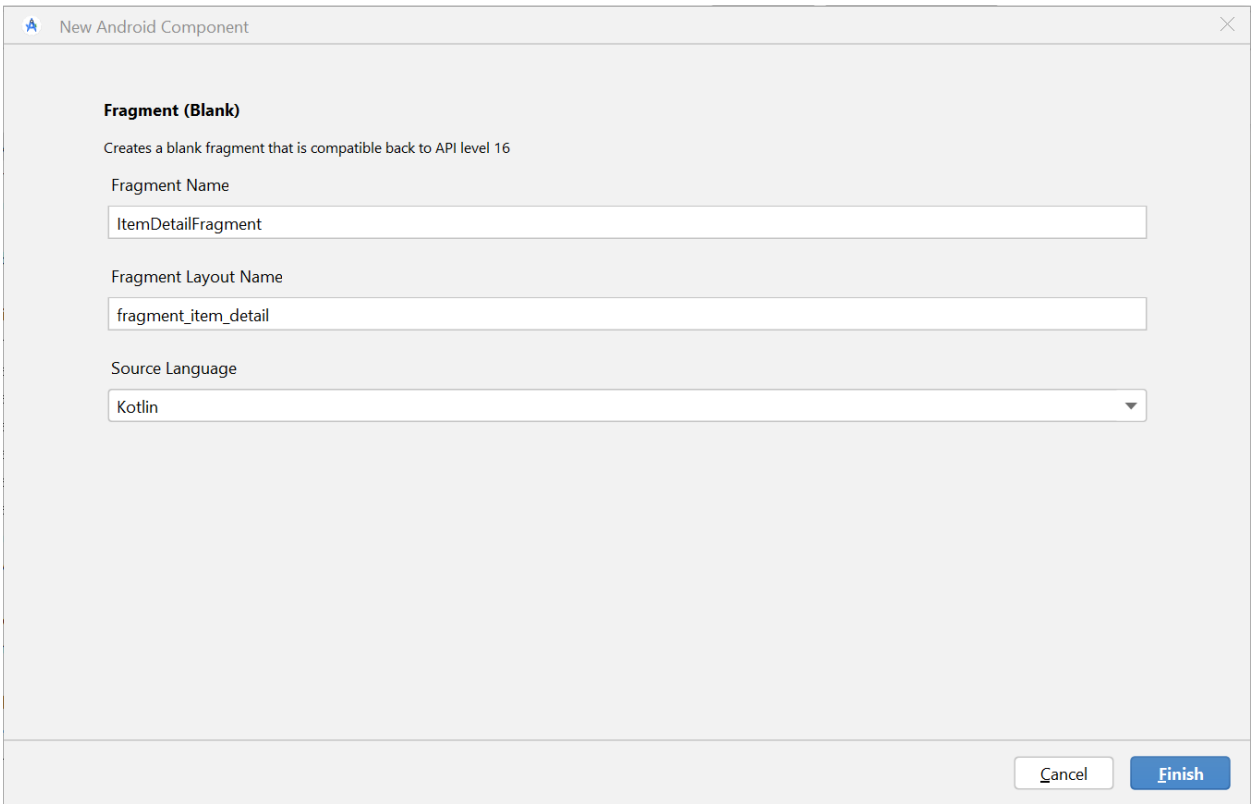
<?xml version="1.0" encoding="utf-8"?>

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

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:defaultNavHost="true"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:navGraph="@navigation/nav_graph" />

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

Chạy thử ứng dụng Inventory:



Bước 11. Thêm ItemDetailFragment



Bước 12. Cập nhật giao diện fragment_item_detail.xml

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="@dimen/margin"
    tools:context=".ItemDetailFragment">

    <TextView
        android:id="@+id/item_name"
        style="@style/Widget.Inventory.TextView"
        android:layout_width="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="Screwdrivers" />

    <TextView
        android:id="@+id/item_price"
        style="@style/Widget.Inventory.TextView"
        android:layout_width="wrap_content"
        android:layout_marginTop="@dimen/margin"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/item_name"
        tools:text="$5.50" />
```

```
<TextView
    android:id="@+id/item_count_label"
    style="@style/Widget.Inventory.TextView"
    android:layout_width="wrap_content"
    android:text="@string/quantity"
    app:layout_constraintBaseline_toBaselineOf="@+id/item_count"
    app:layout_constraintEnd_toStartOf="@+id/item_count"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent" />

<TextView
    android:id="@+id/item_count"
    style="@style/Widget.Inventory.TextView"
    android:layout_width="0dp"
    android:layout_marginStart="@dimen/margin_between_elements"
    android:layout_marginTop="@dimen/margin"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/item_count_label"
    app:layout_constraintTop_toBottomOf="@+id/item_price"
    tools:text="5" />

<Button
    android:id="@+id/sell_item"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin"
    android:text="@string/sell"
    app:layout_constraintBottom_toTopOf="@+id/delete_item"
```

```
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/item_count" />

    <Button
        android:id="@+id/delete_item"
        style="?attr/materialButtonOutlinedStyle"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin"
        android:text="@string/delete"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/sell_item" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/edit_item"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="@dimen/margin_between_elements"
        android:layout_marginBottom="@dimen/margin_between_elements"
        android:contentDescription="@string/edit_item"
        android:src="@drawable/ic_edit"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:tint="@android:color/white" />

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

Bước 13. Thêm ic_edit.xml vào res/drawable

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:tint="#FFFFFF"
    android:viewportWidth="24"
    android:viewportHeight="24">
    <path
        android:fillColor="@android:color/white"
        android:pathData="M3,17.25V21h3.75L17.81,9.94l-3.75,-
3.75L3,17.25zM20.71,7.04c0.39,-0.39 0.39,-1.02 0,-1.41l-2.34,-2.34c-0.39,-0.39 -1.02,-0.39 -
1.41,0l-1.83,1.83 3.75,3.75 1.83,-1.83z" />
</vector>
```

Bước 14. Cập nhật mã nguồn của ItemDetailFragment

```
private val navigationArgs: ItemDetailFragmentArgs by navArgs()


    private var _binding: FragmentItemDetailBinding? = null
    private val binding get() = _binding!!


    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentItemDetailBinding.inflate(inflater, container, false)
        return binding.root
```

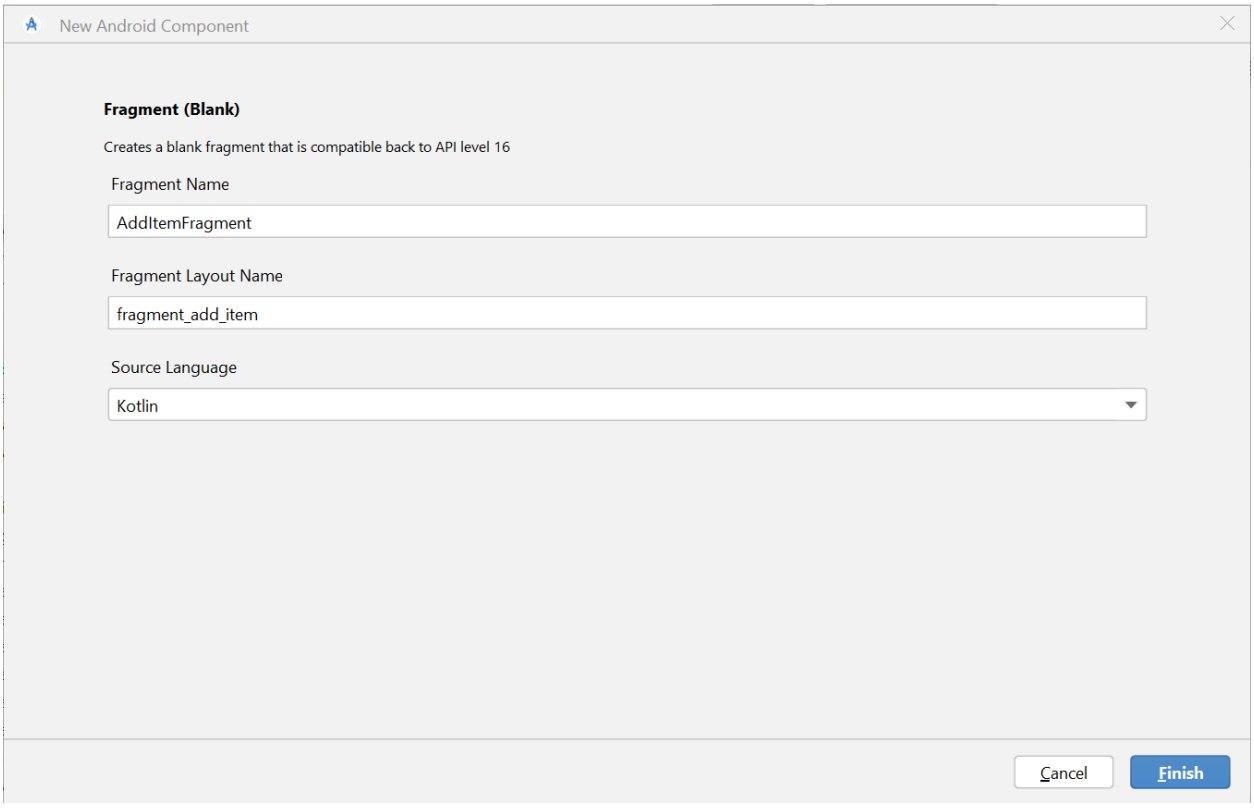
```
}

/**
 * Displays an alert dialog to get the user's confirmation before deleting the item.
 */
private fun showConfirmationDialog() {
    MaterialAlertDialogBuilder(requireContext())
        .setTitle(getString(android.R.string.dialog_alert_title))
        .setMessage(getString(R.string.delete_question))
        .setCancelable(false)
        .setNegativeButton(getString(R.string.no)) { _, _ -> }
        .setPositiveButton(getString(R.string.yes)) { _, _ ->
            deleteItem()
        }
        .show()
}

/**
 * Deletes the current item and navigates to the list fragment.
 */
private fun deleteItem() {
    findNavController().navigateUp()
}

/**
 * Called when fragment is destroyed.
 */
override fun onDestroyView() {
```

```
        super.onDestroyView()
        _binding = null
    }
```

Bước 15. Bổ sung ItemDetailFragment vào nav_graph.xml

```
<fragment
        android:id="@+id/itemDetailFragment"
        android:name="com.example.inventory.ItemDetailFragment"
        android:label="fragment_item_detail"
        tools:layout="@layout/fragment_item_detail" >
        <argument
            android:name="item_id"
            app:argType="integer" />
    </fragment>
```

Bước 16. Thêm AddItemFragment.xml



Bước 17. Cập nhật giao diện fragment_add_item.xml

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/margin">

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/item_name_label"
```

```
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin"
    android:hint="@string/item_name_req"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/item_name"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="textAutoComplete|textCapWords"
        android:singleLine="true" />
</com.google.android.material.textfield.TextInputLayout>

<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/item_price_label"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin"
    android:hint="@string/item_price_req"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/item_name_label"
    app:prefixText="@string/currency_symbol">

    <com.google.android.material.textfield.TextInputEditText
```

```
        android:id="@+id/item_price"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="numberDecimal"
        android:singleLine="true" />
</com.google.android.material.textfield.TextInputLayout>

<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/item_count_label"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin"
    android:hint="@string/quantity_req"
    app:layout_constraintBottom_toTopOf="@+id/save_action"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/item_price_label">

    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/item_count"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="number"
        android:singleLine="true" />
</com.google.android.material.textfield.TextInputLayout>

<Button
    android:id="@+id/save_action"
```

```
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginTop="32dp"
            android:text="@string/save_action"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/item_count_label" />

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

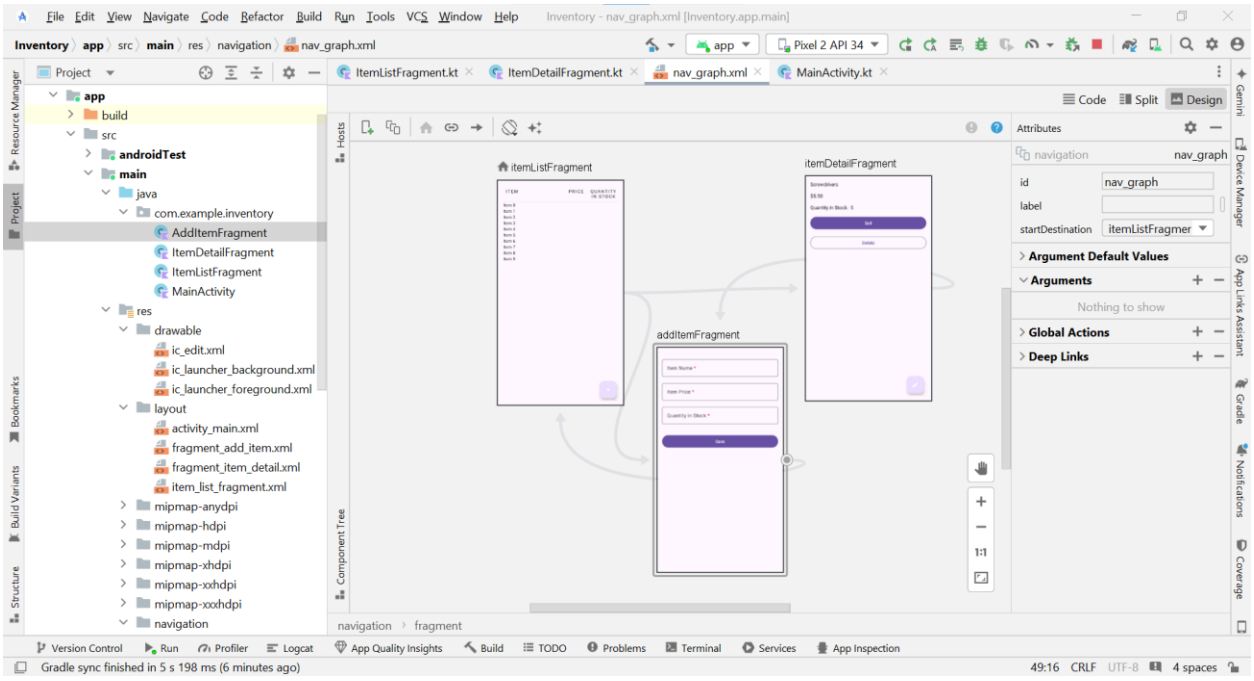
Bước 18. Cập nhật mã nguồn của AddItemFragment

```
package com.example.inventory

import android.content.Context.INPUT_METHOD_SERVICE
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.view.inputmethod.InputMethodManager
import androidx.navigation.fragment.navArgs
import com.example.inventory.databinding.FragmentAddItemBinding


class AddItemFragment : Fragment() {

    private val navigationArgs: ItemDetailFragmentArgs by navArgs()
```

```
    // Binding object instance corresponding to the fragment_add_item.xml layout
    // This property is non-null between the onCreateView() and onDestroyView() lifecycle callbacks,
    // when the view hierarchy is attached to the fragment
    private var _binding: FragmentAddItemBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentAddItemBinding.inflate(inflater, container, false)
        return binding.root
    }

    /**
     * Called before fragment is destroyed.
     */
    override fun onDestroyView() {
        super.onDestroyView()
        // Hide keyboard.
        val inputMethodManager = requireActivity().getSystemService(INPUT_METHOD_SERVICE) as
                InputMethodManager

inputMethodManager.hideSoftInputFromWindow(requireActivity().currentFocus?.windowToken, 0)
        _binding = null
    }
```

}

Bước 19. Thêm AddItemFragment vào nav_graph.xml



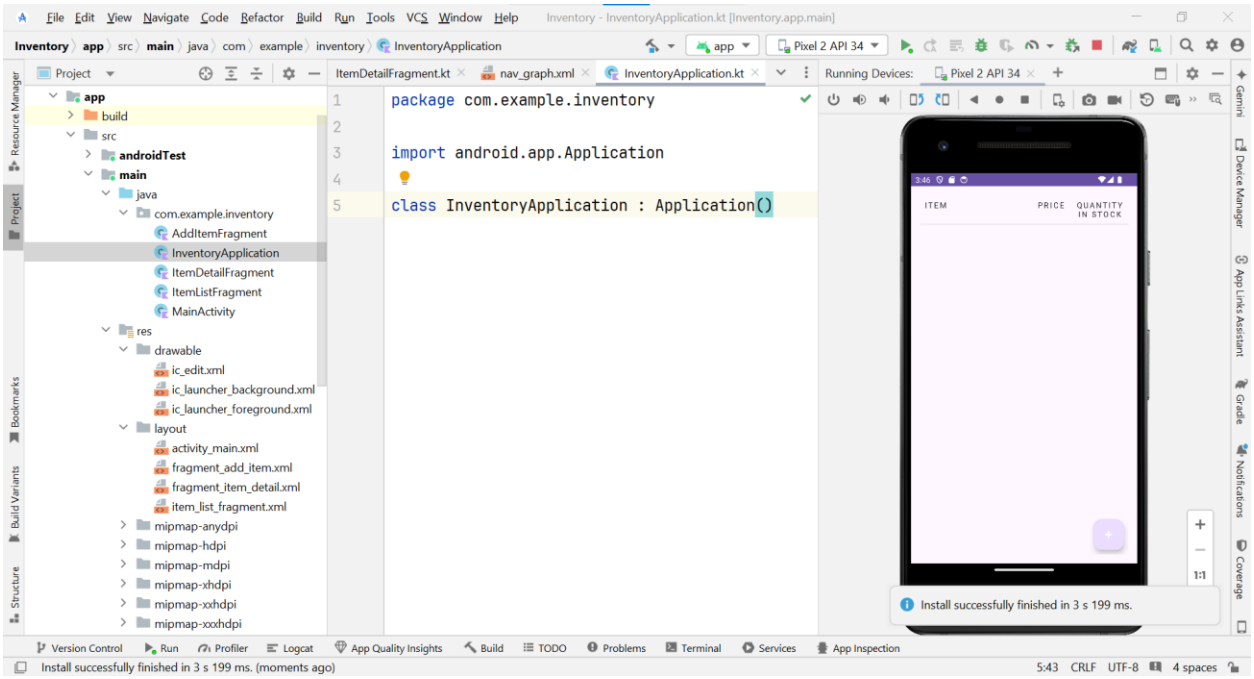

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/itemListFragment">

    <fragment
        android:id="@+id/itemListFragment"
        android:name="com.example.inventory.ItemListFragment"
        android:label="item_list_fragment"
        tools:layout="@layout/item_list_fragment" >
        <action
            android:id="@+id/action_itemListFragment_to_itemDetailFragment"
            app:destination="@id/itemDetailFragment" />
        <action
            android:id="@+id/action_itemListFragment_to_addItemFragment"
            app:destination="@id/addItemFragment" />
    </fragment>
    <fragment
        android:id="@+id/itemDetailFragment"
        android:name="com.example.inventory.ItemDetailFragment"
```

```
        android:label="fragment_item_detail"
        tools:layout="@layout/fragment_item_detail" >
        <argument
            android:name="item_id"
            app:argType="integer" />
        <action
            android:id="@+id/action_itemDetailFragment_to_addItemFragment"
            app:destination="@id/addItemFragment" />
    </fragment>
    <fragment
        android:id="@+id/addItemFragment"
        android:name="com.example.inventory.AddItemFragment"
        android:label="{title}"
        tools:layout="@layout/fragment_add_item">
        <argument
            android:name="title"
            app:argType="string" />
        <argument
            android:name="item_id"
            android:defaultValue="-1"
            app:argType="integer" />
        <action
            android:id="@+id/action_addItemFragment_to_itemListFragment"
            app:destination="@id/itemListFragment"
            app:popUpTo="@id/itemListFragment"
            app:popUpToInclusive="true" />
    </fragment>
</navigation>
```

Bước 20. Bổ sung lớp InventoryApplication

```
package com.example.inventory

import android.app.Application

class InventoryApplication : Application()
```

Bước 21. Chạy thử ứng dụng



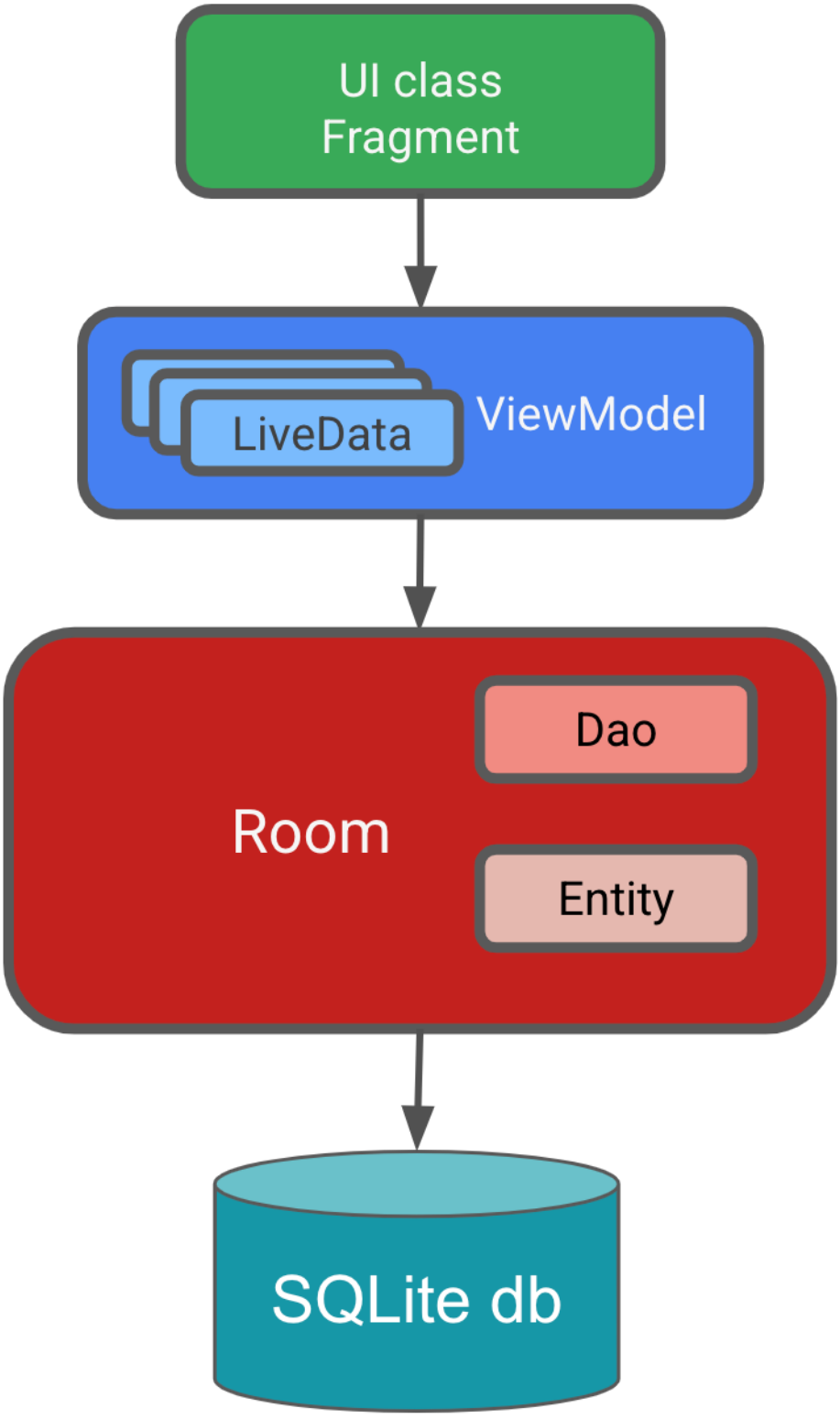
Persist data with Room

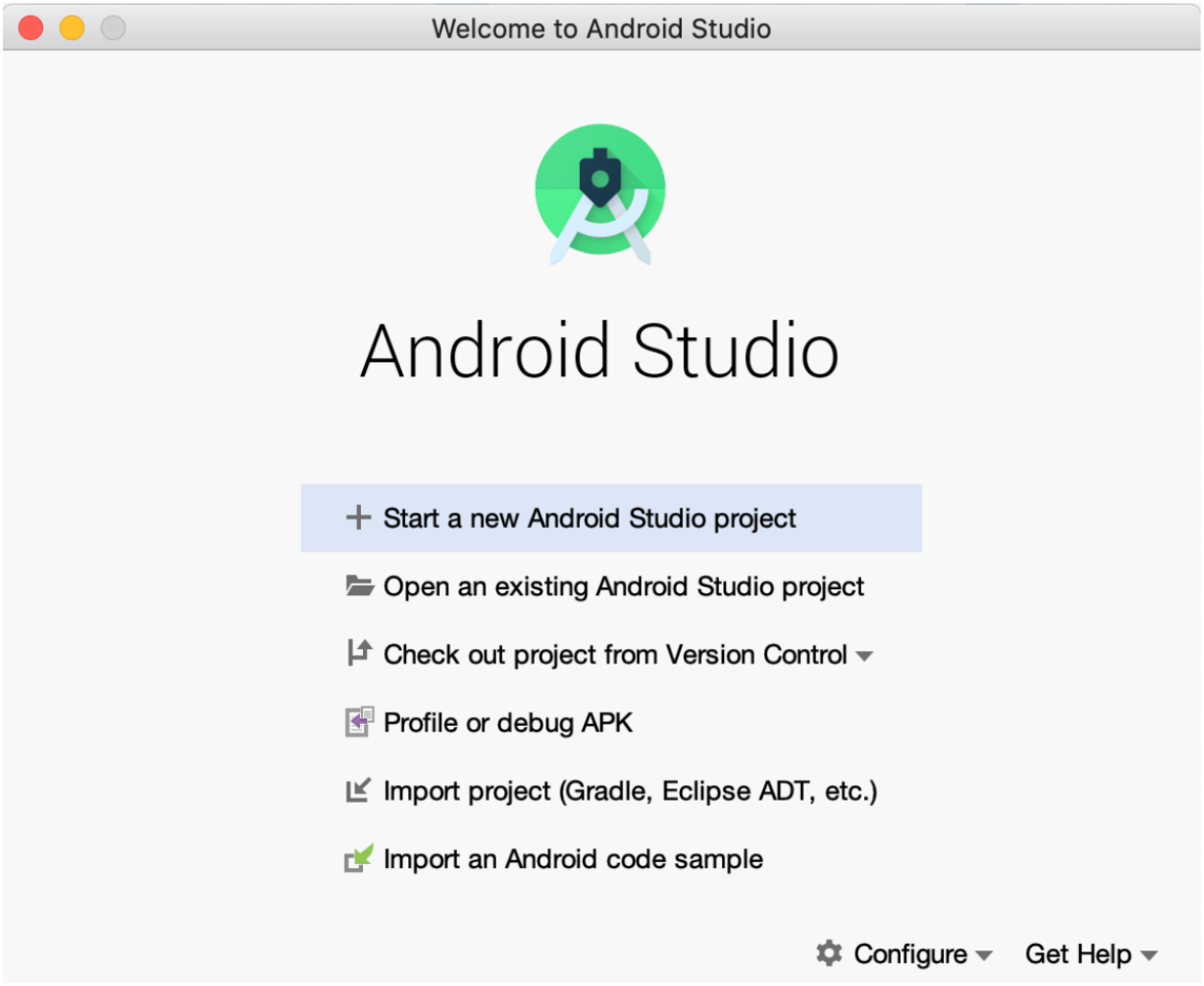
1. Before you begin

Most production quality apps have data that needs to be saved, even after the user closes the app. For example, the app might store a playlist of songs, items on a to-do list, records of expenses and income, a catalog of constellations, or a history of personal data. For most of these cases, you use a database to store this persistent data.

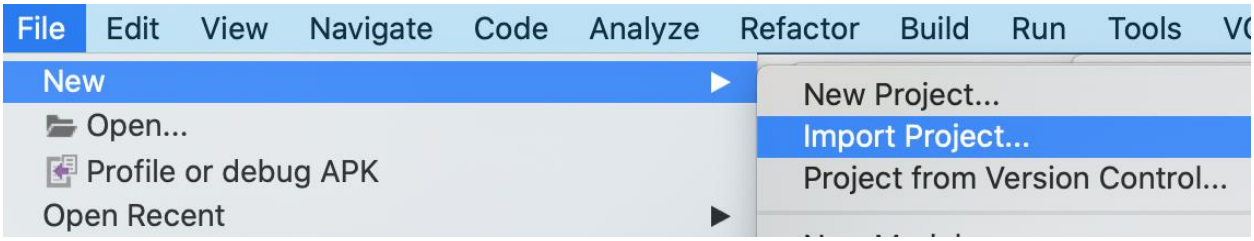
Room is a persistence library that's part of Android Jetpack. Room is an abstraction layer on top of a SQLite database. SQLite uses a specialized language (SQL) to perform database operations. Instead of using SQLite directly, Room simplifies the chores of setting up, configuring, and interacting with the database. Room also provides compile-time checks of SQLite statements.

The image below shows how Room fits in with the overall architecture recommended in this course.

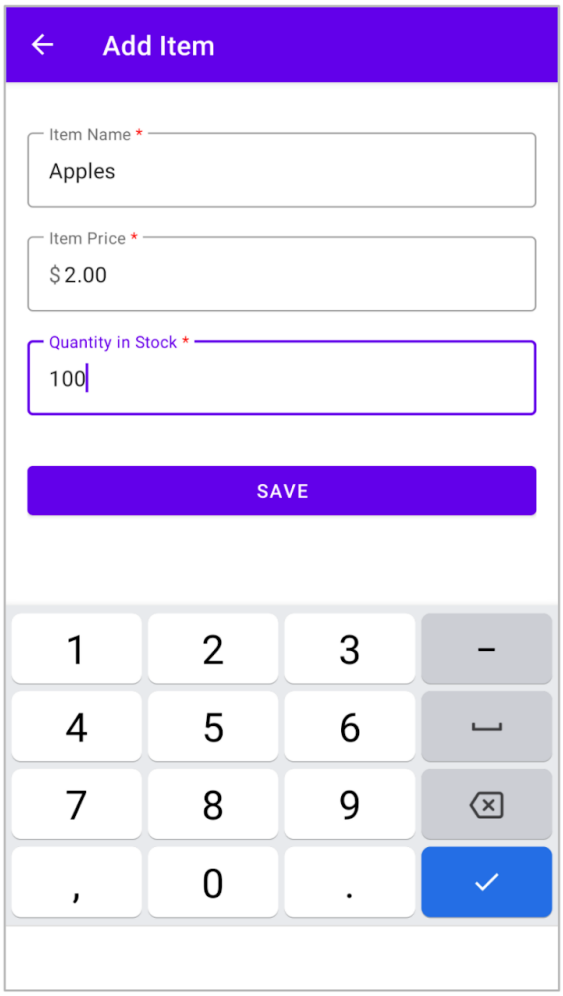

UI class
Fragment
LiveData
ViewModel
Room
Dao
Entity
SQLite db

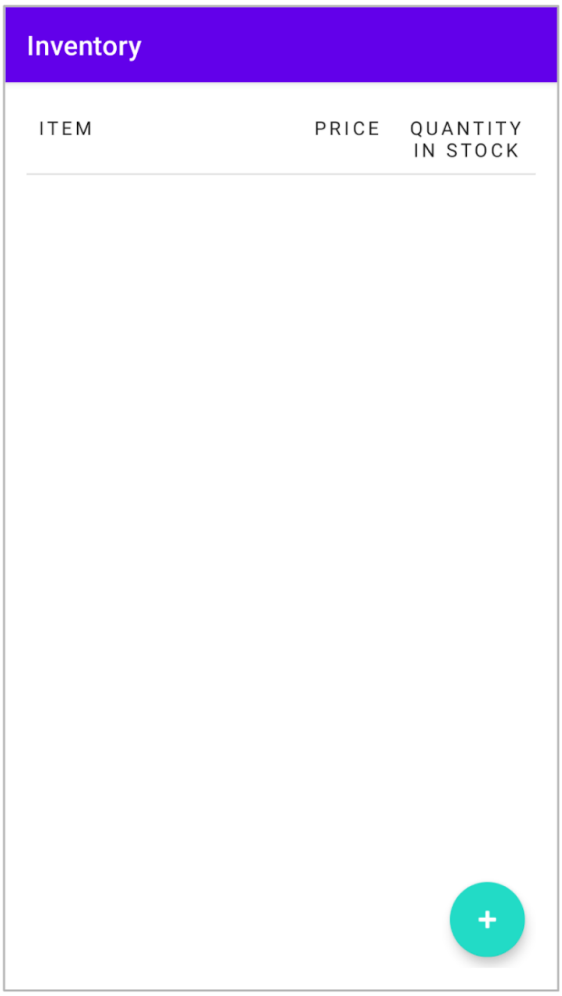


Note: If Android Studio is already open, instead, select the **File** > **New** > **Import Project** menu option.



3. In the **Import Project** dialog, navigate to where the unzipped project folder is located (likely in your **Downloads** folder).
4. Double-click on that project folder.
5. Wait for Android Studio to open the project.
6. Click the **Run** button  to build and run the app. Make sure it builds as expected.





In this codelab, you will add the database portion of an app that saves the inventory details in the SQLite database. You will be using the Room persistence library to interact with the SQLite database.

Code walkthrough

The starter code you downloaded has the screen layouts pre-designed for you. In this pathway, you will focus on implementing the database logic. Here is a brief walkthrough of some of the files to get you started.

main_activity.xml

The main activity that hosts all the other fragments in the app. The `onCreate()` method retrieves `NavController` from the `NavHostFragment` and sets up the action bar for use with the `NavController`.

item_list_fragment.xml

The first screen shown in the app. It mainly contains a RecyclerView and a FAB. You will implement the RecyclerView later in the pathway.

fragment_add_item.xml

This layout contains text fields for entering the details of the new inventory item to be added.

ItemListFragment.kt

This fragment contains mostly boilerplate code. In the `onViewCreated()` method, click listener is set on FAB to navigate to the add item fragment.

AddItemFragment.kt

This fragment is used to add new items into the database. The `onCreateView()` function initializes the binding variable and the `onDestroyView()` function hides the keyboard before destroying the fragment.

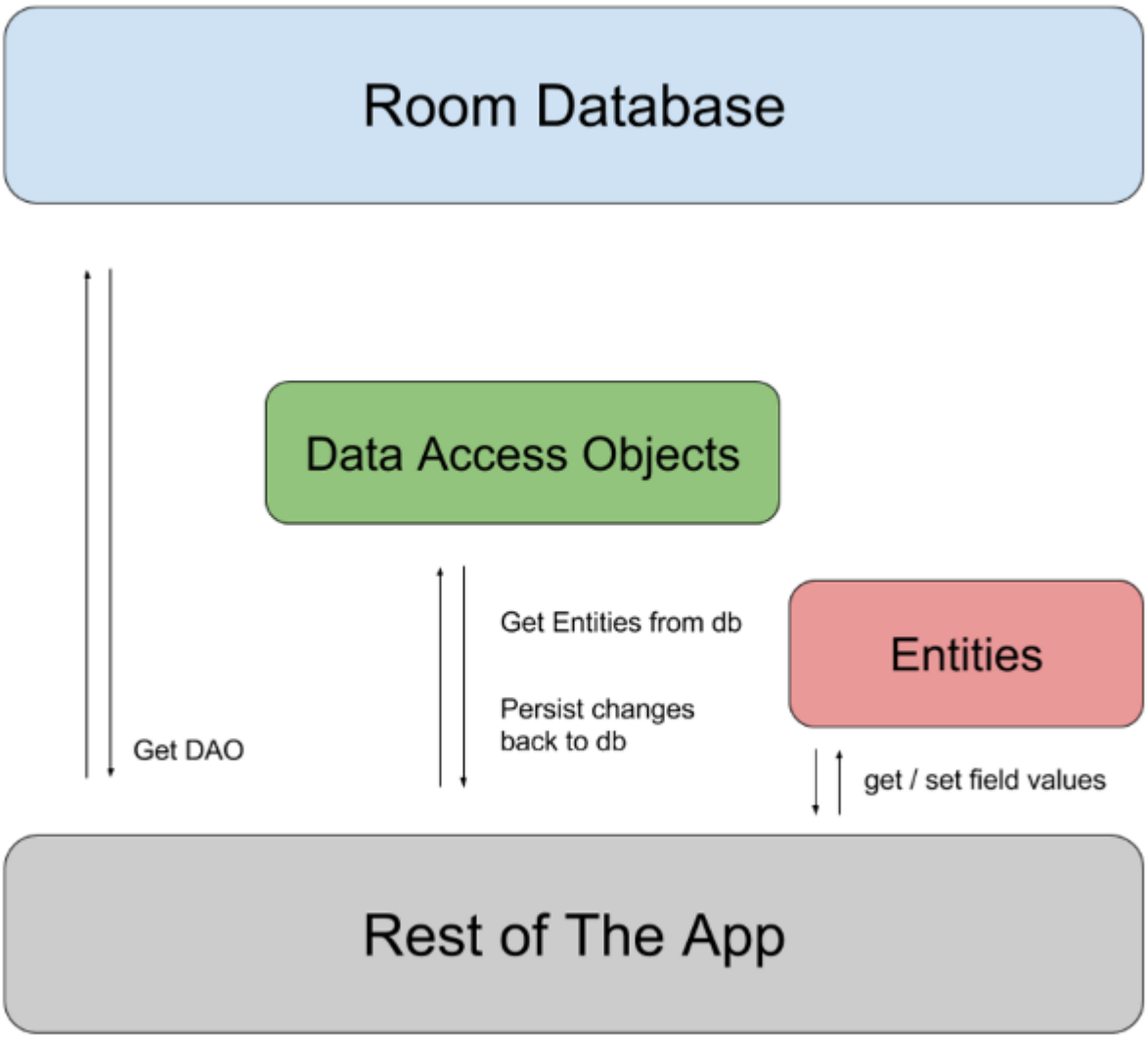
4. Main components of Room

Kotlin provides an easy way to deal with data by introducing data classes. This data is accessed and possibly modified using function calls. However, in the database world, you need *tables* and *queries* to access and modify data. The following components of Room make these workflows seamless.

There are three major components in Room:

- Data entities represent tables in your app's database. They are used to update the data stored in rows in tables, and to create new rows for insertion.
- Data access objects (DAOs) provide methods that your app uses to retrieve, update, insert, and delete data in the database.
- Database class holds the database and is the main access point for the underlying connection to your app's database. The database class provides your app with instances of the DAOs associated with that database.

You will implement and learn more about these components later in the codelab. The following diagram demonstrates how the components of the Room work together to interact with the database.



Add Room libraries

In this task, you'll add the required Room component libraries to your Gradle files.

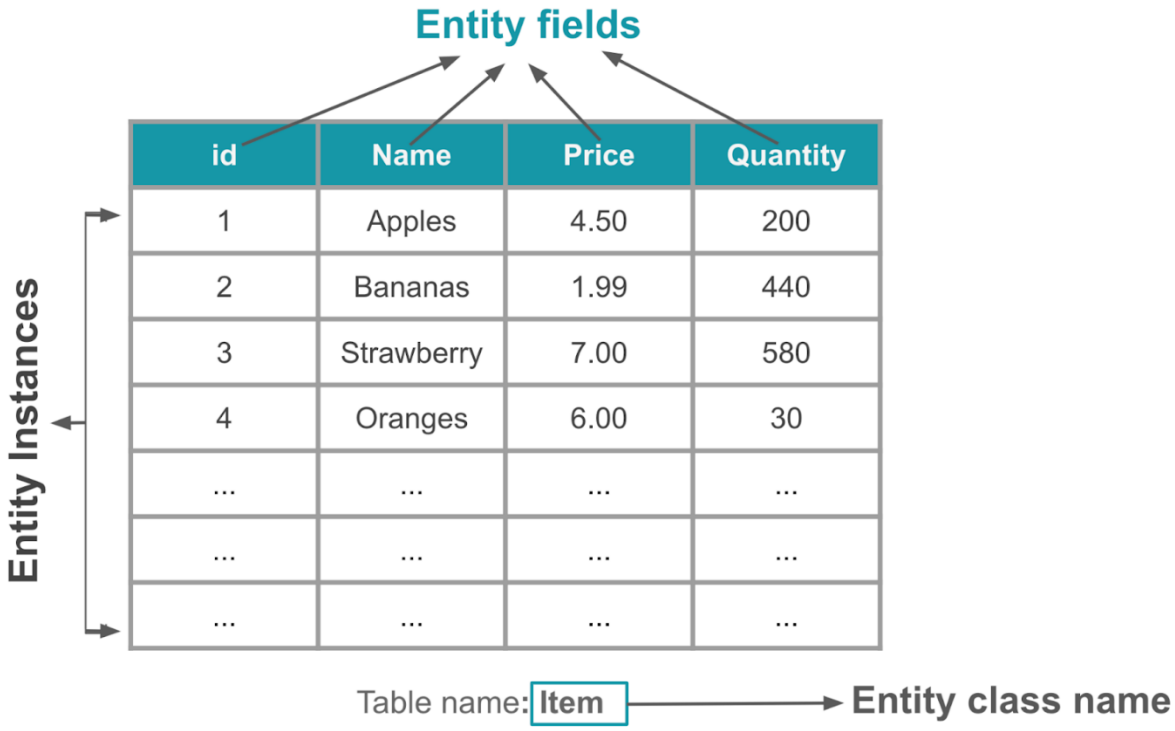
1. Open module level gradle file, `build.gradle (Module: InventoryApp.app)` In the `dependencies` block, add the following dependencies for the Room library.

```
// Room
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
implementation "androidx.room:room-ktx:$room_version"
```

Note: For the library dependencies in your gradle file, always use the most current stable release version numbers from the AndroidX releases page.

5. Create an item Entity

Entity class defines a table, and each instance of this class represents a row in the database table. The entity class has mappings to tell Room how it intends to present and interact with the information in the database. In your app, the entity is going to hold information about inventory items such as item name, item price and stock available.

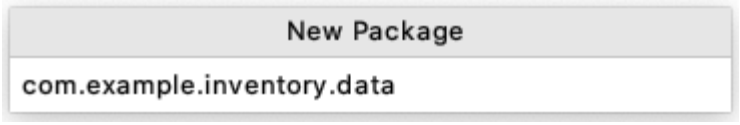


`@Entity` annotation marks a class as a database Entity class. For each Entity class a database table is created to hold the items. Each field of the Entity is represented as a column in the database, unless it is denoted otherwise (see Entity docs for details). Every entity instance that is stored in the database must have a primary key. The primary key is used to uniquely identify every record/entry in your database tables. Once assigned, the primary key cannot be modified, it represents the entity object as long as it exists in the database.

In this task, you will create an Entity class. Define fields to store the following inventory information for each item.

- An `Int` to store the primary key.
- A `String` to store the item name.
- A `double` to store the item price.
- An `Int` to store the quantity in stock.

1. Open starter code in the Android Studio.
2. Create a package called `data` under `com.example.inventory` base package.



3. Inside the `data` package, create a Kotlin class called `Item`. This class will represent a database entity in your app. In the next step you will add corresponding fields to store inventory information.
4. Update the `Item` class definition with the following code. Declare `id` of type `Int`, `itemName` of type `String`, `itemPrice` of type `Double`, and `quantityInStock` of type `Int` as parameters for the primary constructor. Assign a default value of `0` to `id`. This will be the primary key, an ID to uniquely identify every record/entry in your `Item` table.

```
class Item(
    val id: Int = 0,
    val itemName: String,
    val itemPrice: Double,
    val quantityInStock: Int
)
```

Refresher on primary constructor: The primary constructor is part of the class header in a Kotlin class: it goes after the class name (and optional type parameters).

Data classes

Data classes are primarily used to hold data in Kotlin. They are marked with the keyword `data`. Kotlin data class objects have some extra benefits, the compiler automatically generates utilities for comparing, printing and copying such as `toString()`, `copy()`, and `equals()`.

Example:

```
// Example data class with 2 properties.
data class User(val first_name: String, val last_name: String){
}
```

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

- The primary constructor needs to have at least one parameter.
- All primary constructor parameters need to be marked as `val` or `var`.
- Data classes cannot be `abstract`, `open`, `sealed` or `inner`.

Warning: The compiler only uses the properties defined inside the primary constructor for the automatically generated functions. The properties declared inside the class body are excluded from the generated implementations.

To learn more about Data classes, check out the documentation.

5. Convert the `Item` class to a data class by prefixing its class definition with `data` keyword.

```
data class Item(
    val id: Int = 0,
    val itemName: String,
    val itemPrice: Double,
    val quantityInStock: Int
)
```

6. Above the `Item` class declaration, annotate the data class with `@Entity`. Use `tableName` argument to give the `item` as the SQLite table name.

```
@Entity(tableName = "item")
data class Item(
    ...
)
```

Important: When prompted by Android Studio, import `Entity` and all other Room annotations (which you will use later in the codelab) from the `androidx` library. For example, `androidx.room.Entity`.

Note: `@Entity` annotation has several possible arguments. By default (no arguments to `@Entity`), the table name will be the same as the class. The `tableName` argument let's you give a different or a more helpful table name. This argument for the `tableName` is optional, but highly recommended. For simplicity you will give the same name as the class name, that is `item`. There are several other arguments for `@Entity` you can investigate in the documentation.

7. To identify the `id` as the primary key, annotate the `id` property with `@PrimaryKey`. Set the parameter `autoGenerate` to `true` so that `Room` generates the ID for each entity. This guarantees that the ID for each item is unique.

```
@Entity(tableName = "item")
data class Item(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    ...
)
```

8. Annotate the remaining properties with `@ColumnInfo`. The `ColumnInfo` annotation is used to customise the column associated with the particular field. For example, when using the `name` argument, you can specify a different column name for the field rather than the variable name. Customize the property names using parameters as shown below. This approach is similar to using `tableName` to specify a different name to the database.

```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity
data class Item(
```

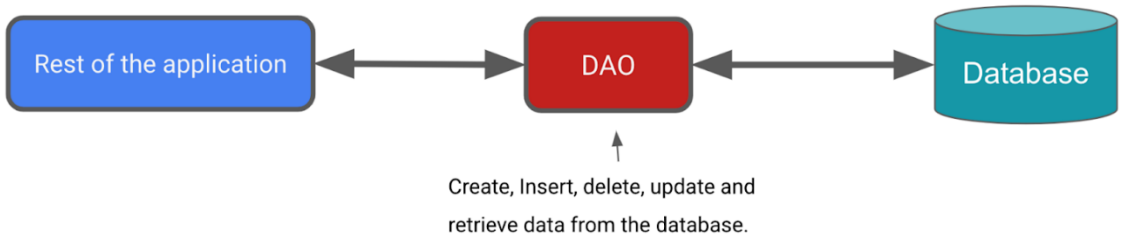
```
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    @ColumnInfo(name = "name")
    val itemName: String,
    @ColumnInfo(name = "price")
    val itemPrice: Double,
    @ColumnInfo(name = "quantity")
    val quantityInStock: Int
)
```

6. Create the item DAO

Data Access Object (DAO)

The Data Access Object (DAO) is a pattern used to separate the persistence layer with the rest of the application by providing an abstract interface. This isolation follows the single responsibility principle, which you have seen in the previous codelabs.

The functionality of the DAO is to hide all the complexities involved in performing the database operations in the underlying persistence layer from the rest of the application. This allows the data access layer to be changed independently of the code that uses the data.



In this task, you define a Data Access Object (DAO) for the Room. Data access objects are the main components of Room that are responsible for defining the interface that accesses the database.

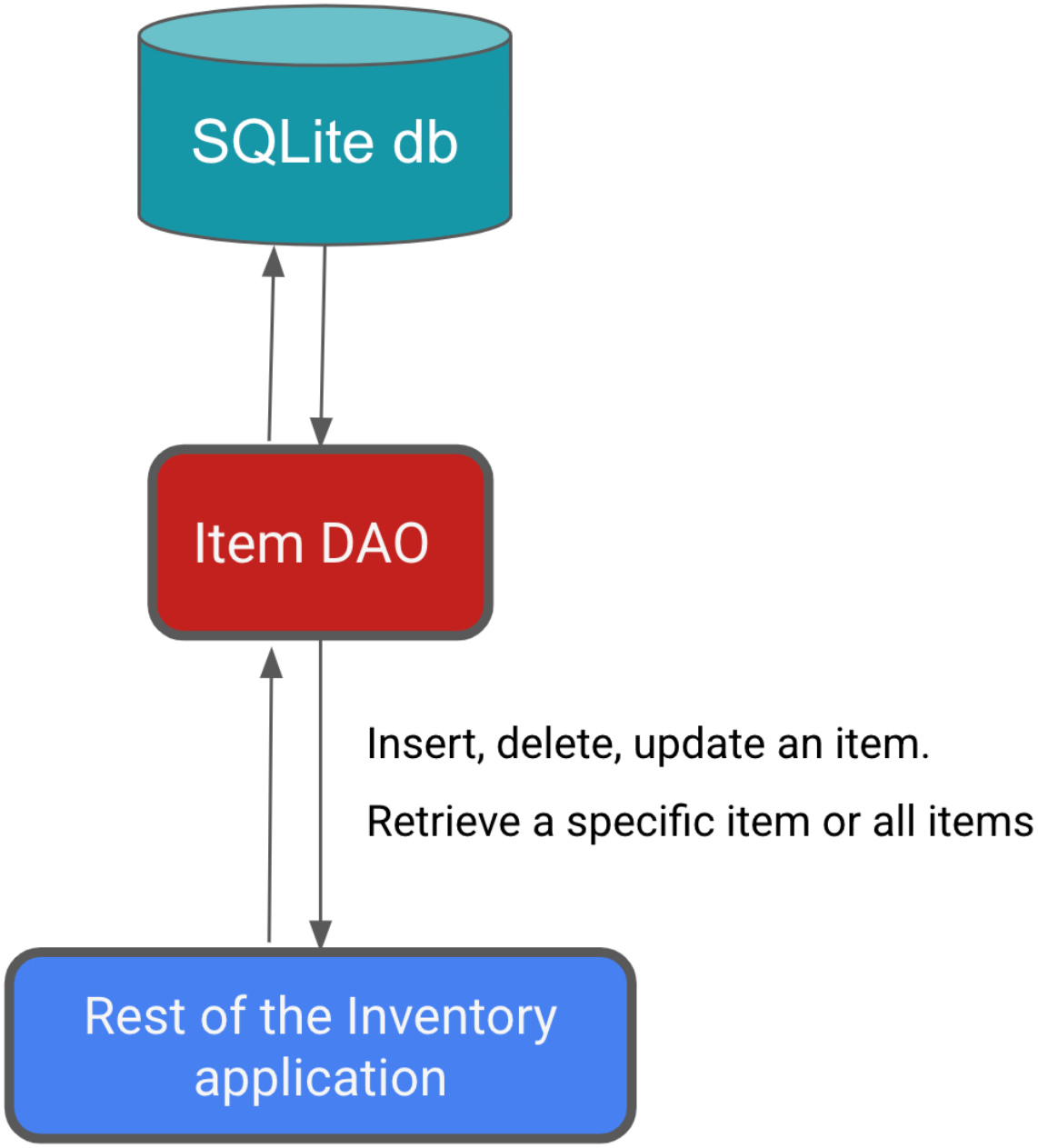
The DAO you will create will be a custom interface providing convenience methods for querying/retrieving, inserting, deleting, and updating the database. Room will generate an implementation of this class at compile time.

For common database operations, the `Room` library provides convenience annotations, such as `@Insert`, `@Delete`, and `@Update`. For everything else, there is the `@Query` annotation. You can write any query that's supported by SQLite.

As an added bonus, as you write your queries in Android Studio, the compiler checks your SQL queries for syntax errors.

For the inventory app, you need to be able to do the following:

- **Insert** or add a new item.
- **Update** an existing item to update name, price, and quantity.
- **Get** a specific item based on its primary key, `id`.
- **Get all items**, so you can display them.
- **Delete** an entry in the database.



Now, implement the item DAO in your app:

1. In the `data` package, create Kotlin class `ItemDao.kt`.
2. Change the class definition to `interface` and annotate with `@Dao`.

```
@Dao
interface ItemDao {
}
```

3. Inside the body of the interface, add an `@Insert` annotation. Below the `@Insert`, add an `insert()` function that takes an instance of the `Entity` class `item` as its argument. The database operations can take a long time to execute, so they should run on a separate thread. Make the function a suspend function, so that this function can be called from a coroutine.

```
@Insert
suspend fun insert(item: Item)
```

4. Add an argument `OnConflict` and assign it a value of `OnConflictStrategy.`*`IGNORE`*. The argument `OnConflict` tells the Room what to do in case of a conflict. The `OnConflictStrategy.`*`IGNORE`* strategy ignores a new item if it's primary key is already in the database. To know more about the available conflict strategies, check out the documentation.

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun insert(item: Item)
```

Now the `Room` will generate all the necessary code to insert the `item` into the database. When you call `insert()` from your Kotlin code, `Room` executes a SQL query to insert the entity into the database. (Note: The function can be named anything you want; it doesn't have to be called `insert()`.)

5. Add an `@Update` annotation with an `update()` function for one `item`. The entity that's updated has the same key as the entity that's passed in. You can update some or all of the entity's other properties. Similar to the `insert()` method, make the following `update()` method `suspend`.

```
@Update
suspend fun update(item: Item)
```

6. Add `@Delete` annotation with a `delete()` function to delete item(s). Make it a suspend method. The `@Delete` annotation deletes one item, or a list of items. (Note: You need to pass the entity(s) to be deleted, if you don't have the entity you may have to fetch it before calling the `delete()` function.)

```
@Delete
suspend fun delete(item: Item)
```

There is no convenience annotation for the remaining functionality, so you have to use the `@Query` annotation and supply SQLite queries.

7. Write a SQLite query to retrieve a particular item from the item table based on the given `id`. You will then add Room annotation and use a modified version of the following query in the later steps. In next steps, you will also change this into a DAO method using Room.
8. Select all columns from the `item`
9. `WHERE` the `id` matches a specific value.

Example:

```
SELECT * from item WHERE id = 1
```

8. Change the above SQL query to use with the Room annotation and an argument. Add a `@Query` annotation, supply the query as a string parameter to the `@Query` annotation. Add a `String` parameter to `@Query` that is a SQLite query to retrieve an item from the item table.
9. Select all columns from the `item`
10. `WHERE` the `id` matches the :`id` argument. Notice the `:id`. You use the colon notation in the query to reference arguments in the function.

```
@Query("SELECT * from item WHERE id = :id")
```

9. Below the `@Query` annotation add `getItem()` function that takes an `Int` argument and returns a `Flow<Item>`.

```
@Query("SELECT * from item WHERE id = :id")
fun getItem(id: Int): Flow<Item>
```

Using `Flow` or `LiveData` as return type will ensure you get notified whenever the data in the database changes. It is recommended to use `Flow` in the persistence layer. The `Room` keeps this `Flow` updated for you, which means you only need to explicitly get the data once. This is helpful to update the inventory list, which you will implement in the next codelab. Because of the `Flow` return type, Room also runs the query on the background thread. You don't need to explicitly make it a `suspend` function and call inside a coroutine scope.

You may need to import `Flow` from `kotlinx.coroutines.flow.Flow`.

10. Add a `@Query` with a `getItems()` function:
11. Have the SQLite query return all columns from the `item` table, ordered in ascending order.
12. Have `getItems()` return a list of `Item` entities as `Flow`. `Room` keeps this `Flow` updated for you, which means you only need to explicitly get the data once.

```
@Query("SELECT * from item ORDER BY name ASC")
fun getItems(): Flow<List<Item>>
```

11. Though you won't see any visible changes, run your app to make sure it has no errors.

7. Create a Database instance

In this task, you create a `RoomDatabase` that uses the `Entity` and DAO that you created in the previous task. The database class defines the list of entities and data access objects. It is also the main access point for the underlying connection.

The `Database` class provides your app with instances of the DAOs you've defined. In turn, the app can use the DAOs to retrieve data from the database as instances of the associated data entity objects. The app can also use the defined data entities to update rows from the corresponding tables, or to create new rows for insertion.

You need to create an abstract `RoomDatabase` class, annotated with `@Database`. This class has one method that either creates an instance of the `RoomDatabase` if it doesn't exist, or returns the existing instance of the `RoomDatabase`.

Here's the general process for getting the `RoomDatabase` instance:

- Create a `public abstract` class that extends `RoomDatabase`. The new abstract class you defined acts as a database holder. The class you defined is abstract, because `Room` creates the implementation for you.
- Annotate the class with `@Database`. In the arguments, list the entities for the database and set the version number.
- Define an abstract method or property that returns an `ItemDao` Instance and the `Room` will generate the implementation for you.
- You only need one instance of the `RoomDatabase` for the whole app, so make the `RoomDatabase` a singleton.
- Use `Room`'s `Room.databaseBuilder` to create your (`item_database`) database only if it doesn't exist. Otherwise, return the existing database.

Tip: The following code can be used as a template for your future projects. The way you create the `RoomDatabase` instance is similar to the process defined above. You may have to replace the entities and Dao's specific to your app.

Create the Database

1. In the `data` package, create a Kotlin class `ItemRoomDatabase.kt`.
2. In the `ItemRoomDatabase.kt` file, make `ItemRoomDatabase` class as an `abstract` class that extends `RoomDatabase`. Annotate the class with `@Database`. You will fix the missing parameters error in the next step.

```
@Database
abstract class ItemRoomDatabase : RoomDatabase() {}
```

3. The `@Database` annotation requires several arguments, so that `Room` can build the database.

- Specify the `Item` as the only class with the list of `entities`.
- Set the `version` as `1`. Whenever you change the schema of the database table, you'll have to increase the version number.
- Set `exportSchema` to `false`, so as not to keep schema version history backups.

```
@Database(entities = [Item::class], version = 1, exportSchema = false)
```

4. The database needs to know about the DAO. Inside the body of the class, declare an abstract function that returns the `ItemDao`. You can have multiple DAOs.

```
abstract fun itemDao(): ItemDao
```

5. Below the abstract function, define a `companion` object. The companion object allows access to the methods for creating or getting the database using the class name as the qualifier.

```
 companion object {}
```

6. Inside the `companion` object, declare a private nullable variable `INSTANCE` for the database and initialize it to `null`. The `INSTANCE` variable will keep a reference to the database, when one has been created. This helps in maintaining a single instance of the database opened at a given time, which is an expensive resource to create and maintain.

Annotate `INSTANCE` with `@Volatile`. The value of a volatile variable will never be cached, and all writes and reads will be done to and from the main memory. This helps make sure the value of `INSTANCE` is always up-to-date and the same for all execution threads. It means that changes made by one thread to `INSTANCE` are visible to all other threads immediately.

```
@Volatile
private var INSTANCE: ItemRoomDatabase? = null
```

7. Below `INSTANCE`, while still inside the `companion` object, define a `getDatabase()` method with a `Context` parameter that the database builder will need. Return a type `ItemRoomDatabase`. You'll see an error because `getDatabase()` isn't returning anything yet.

```
fun getDatabase(context: Context): ItemRoomDatabase {}
```

8. Multiple threads can potentially run into a race condition and ask for a database instance at the same time, resulting in two databases instead of one. Wrapping the code to get the database inside a `synchronized` block means that only one thread of execution at a time can enter this block of code, which makes sure the database only gets initialized once.

Inside `getDatabase()`, return `INSTANCE` variable or if `INSTANCE` is null, initialize it inside a `synchronized{}` block. Use the elvis operator(`?:`) to do this. Pass in `this` the companion object, that you want to be locked inside the function block. You will fix the error in the later steps.

```
return INSTANCE ?: synchronized(this) { }
```

9. Inside the synchronized block, create a `val` instance variable, and use the database builder to get the database. You will still have errors which you will fix in the next steps.

```
val instance = Room.databaseBuilder()
```

10. At the end of the `synchronized` block, return `instance`.

```
return instance
```

11. Inside the `synchronized` block, initialize the `instance` variable, and use the database builder to get a database. Pass in the application context, the database class, and a name for the database, `item_database` to the `Room.databaseBuilder()`.

```
val instance = Room.databaseBuilder(
    context.applicationContext,
    ItemRoomDatabase::class.java,
    "item_database"
)
```

Android Studio will generate a Type Mismatch error. To remove this error, you'll have to add a migration strategy and `build()` in the following steps.

12. Add the required migration strategy to the builder. Use `.fallbackToDestructiveMigration()`.

Normally, you would have to provide a migration object with a migration strategy for when the schema changes. A *migration object* is an object that defines how you take all rows with the old schema and convert them to rows in the new schema, so that no data is lost. Migration is beyond the scope of this codelab. A simple solution is to destroy and rebuild the database, which means that the data is lost.

```
.fallbackToDestructiveMigration()
```

13. To create the database instance, call `.build()`. This should remove the Android Studio errors.

```
.build()
```

14. Inside the `synchronized` block, assign `INSTANCE = instance`.

```
INSTANCE = instance
```

15. At the end of the `synchronized` block, return `instance`. Your final code should look like this:

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Item::class], version = 1, exportSchema = false)
abstract class ItemRoomDatabase : RoomDatabase() {

    abstract fun itemDao(): ItemDao

    companion object {
        @Volatile
        private var INSTANCE: ItemRoomDatabase? = null
        fun getDatabase(context: Context): ItemRoomDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    ItemRoomDatabase::class.java,
                    "item_database"
                )
                    .fallbackToDestructiveMigration()
                    .build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

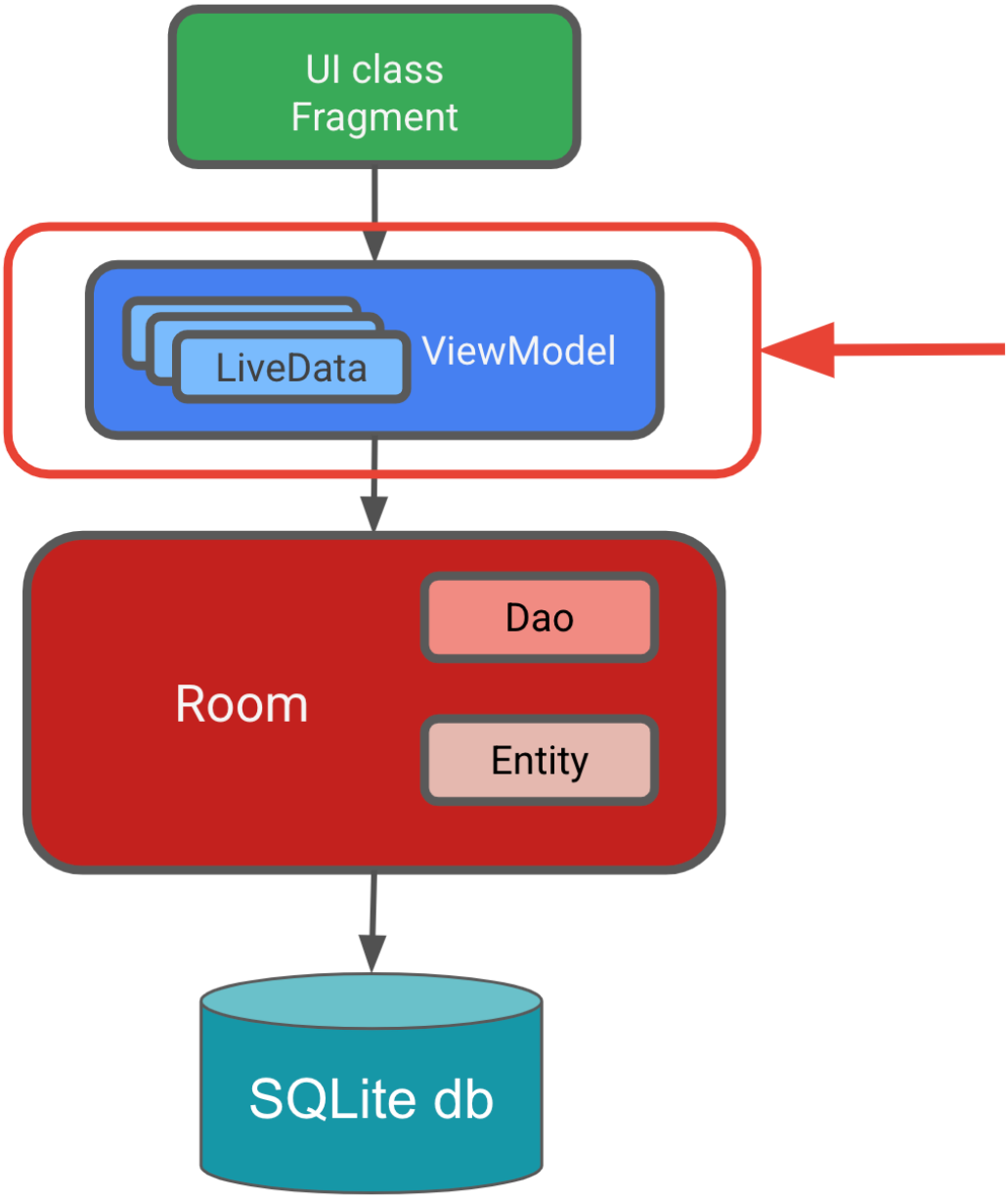
16. Build your code to make sure there are no errors.

Implement Application class

In this task you will instantiate the database instance in the Application class.

1. Open `InventoryApplication.kt`, create a `val` called `database` of the type `ItemRoomDatabase`. Instantiate the `database` instance by calling `getDatabase()` on `ItemRoomDatabase` passing in the context. Use `lazy` delegate so the instance `database` is lazily created when you first need/access the reference (rather than when the app starts). This will create the database (the physical database on the disk) on the first access.

```
import android.app.Application
import com.example.inventory.data.ItemRoomDatabase

class InventoryApplication : Application(){
    val database: ItemRoomDatabase by lazy {
ItemRoomDatabase.getDatabase(this) }
}
```



Create Inventory ViewModel

1. In the `com.example.inventory` package, create a Kotlin class file `InventoryViewModel.kt`.
2. Extend the `InventoryViewModel` class from the `ViewModel` class. Pass in the `ItemDao` object as a parameter to the default constructor.

```
class InventoryViewModel(private val itemDao: ItemDao) : ViewModel() {}
```

3. At the end of the `InventoryViewModel.kt` file outside the class, add `InventoryViewModelFactory` class to instantiate the `InventoryViewModel` instance. Pass in the same constructor parameter as the `InventoryViewModel` that is the `ItemDao` instance. Extend the class from the `ViewModelProvider.Factory` class. You will fix the error regarding the unimplemented methods in the next step.

```
class InventoryViewModelFactory(private val itemDao: ItemDao) :
ViewModelProvider.Factory {
}
```

4. Click on the red bulb and select **Implement Members,** or you can override the `create()` method inside the `ViewModelProvider.Factory` class as follows, which takes any class type as an argument and returns a `ViewModel` object.

```
override fun <T : ViewModel?> create(modelClass: Class<T>): T {
   TODO("Not yet implemented")
}
```

5. Implement the `create()` method. Check if the `modelClass` is the same as the `InventoryViewModel` class and return an instance of it. Otherwise, throw an exception.

```
if (modelClass.isAssignableFrom(InventoryViewModel::class.java)) {
   @Suppress("UNCHECKED_CAST")
   return InventoryViewModel(itemDao) as T
}
throw IllegalArgumentException("Unknown ViewModel class")
```

Tip: The creation of the ViewModel factory is mostly boilerplate code, so you can reuse this code for future ViewModel factories.

Populate the ViewModel

In this task, you will populate the `InventoryViewModel` class to add inventory data to the database. Observe the `Item` entity and **Add Item** screen in the Inventory app.

```
@Entity
data class Item(
   @PrimaryKey(autoGenerate = true)
   val id: Int = 0,
   @ColumnInfo(name = "name")
   val itemName: String,
   @ColumnInfo(name = "price")
   val itemPrice: Double,
   @ColumnInfo(name = "quantity")
   val quantityInStock: Int
)
```

You need the name, price, and stock in hand for that particular item in order to add an entity to the database. Later in the codelab, you will use the **Add Item** screen to get these details from the user. In the current task, you use three strings as input to the ViewModel, convert them to an `Item` entity instance, and save it to the database using the `ItemDao` instance. It's time to implement.

1. In the `InventoryViewModel` class, add a `private` function called `insertItem()` that takes in an `Item` object and adds the data to the database in a non-blocking way.

```
private fun insertItem(item: Item) {
}
```

2. To interact with the database off the main thread, start a coroutine and call the DAO method within it. Inside the `insertItem()` method, use the `viewModelScope.launch` to start a coroutine in the `ViewModelScope`. Inside the launch function, call the suspend function `insert()` on `itemDao` passing in the `item`. The `ViewModelScope` is an extension property to the `ViewModel` class that automatically cancels its child coroutines when the `ViewModel` is destroyed.

```
private fun insertItem(item: Item) {
    viewModelScope.launch {
        itemDao.insert(item)
    }
}
```

Import `kotlinx.coroutines.launch, androidx.lifecycle.`*`viewModelScope`*

`com.example.inventory.data.Item`, if not automatically imported.

Note: Throughout the codelab import `com.example.inventory.data.Item` for `Item` entity, when requested by Android Studio.

3. In the `InventoryViewModel` class, add another private function that takes in three strings and returns an `Item` instance.

```
private fun getNewItemEntry(itemName: String, itemPrice: String, itemCount:
String): Item {
    return Item(
        itemName = itemName,
        itemPrice = itemPrice.toDouble(),
        quantityInStock = itemCount.toInt()
    )
}
```

4. Still inside the `InventoryViewModel` class, add a public function called `addNewItem()` that takes in three strings for item details. Pass in item detail strings to `getNewItemEntry()` function and assign the returned value to a val named `newItem`. Make a call to `insertItem()` passing in the `newItem` to add the new entity to the database. This will be called from the UI fragment to add Item details to the database.

```
fun addNewItem(itemName: String, itemPrice: String, itemCount: String) {
    val newItem = getNewItemEntry(itemName, itemPrice, itemCount)
    insertItem(newItem)
}
```

Notice that you did not use `viewModelScope.launch` for `addNewItem()`, but it is needed above in `insertItem()` when you call a DAO method. The reason is that the s*uspend functions are only allowed to be called from a coroutine or another suspend function*. The function `itemDao.insert(item)` is a suspend function.

You have added all the required functions to add entities to the database. In the next task you will update the **Add Item** fragment to use the above functions.

9. Update AddItemFragment

1. In `AddItemFragment.kt`, at the beginning of the `AddItemFragment` class create a `private val` called `viewModel` of the type `InventoryViewModel`. Use the `by activityViewModels()` Kotlin property delegate to share the `ViewModel` across fragments. You will fix the error in the next step.

```
private val viewModel: InventoryViewModel by activityViewModels {
}
```

2. Inside the lambda, call the `InventoryViewModelFactory()` constructor and pass in the `ItemDao` instance. Use the `database` instance you created in one of the previous tasks to call the `itemDao` constructor.

```
private val viewModel: InventoryViewModel by activityViewModels {
    InventoryViewModelFactory(
        (activity?.application as InventoryApplication).database
            .itemDao()
    )
}
```

Tip: This is mostly boilerplate code, so you can reuse the code for future to create a ViewModel instance using a ViewModel factory.

3. Below the `viewModel` definition, create a `lateinit var` called `item` of the type `Item`.

```
 lateinit var item: Item
```

4. The **Add Item** screen contains three text fields to get the item details from the user. In this step, you will add a function to verify if the text in the TextFields are not empty. You will use this function to verify user input before adding or updating the entity in the database. This validation needs to be done in the `ViewModel` and not in the Fragment. In

the `InventoryViewModel` class, add the following `public` function called `isEntryValid()`.

```
fun isEntryValid(itemName: String, itemPrice: String, itemCount: String):
Boolean {
   if (itemName.isBlank() || itemPrice.isBlank() || itemCount.isBlank()) {
       return false
   }
   return true
}
```

5. In `AddItemFragment.kt`, below the `onCreateView()` function create a `private` function called `isEntryValid()` that returns a `Boolean`. You will fix the missing return value error in the next step.

```
private fun isEntryValid(): Boolean {
}
```

6. In the `AddItemFragment` class, implement the `isEntryValid()` function. Call the `isEntryValid()` function on the `viewModel` instance, passing in the text from the text views. Return the value of the `viewModel.isEntryValid()` function.

```
private fun isEntryValid(): Boolean {
   return viewModel.isEntryValid(
       binding.itemName.text.toString(),
       binding.itemPrice.text.toString(),
       binding.itemCount.text.toString()
   )
}
```

7. In the `AddItemFragment` class below the `isEntryValid()` function, add another `private` function called `addNewItem()` with no parameters and return nothing. Inside the function, call `isEntryValid()` inside the `if` condition.

```
private fun addNewItem() {
   if (isEntryValid()) {
   }
}
```

8. Inside the `if` block, call the `addNewItem()` method on the `viewModel` instance. Pass in the item details entered by the user, use the `binding` instance to read them.

```
if (isEntryValid()) {
   viewModel.addNewItem(
   binding.itemName.text.toString(),
   binding.itemPrice.text.toString(),
   binding.itemCount.text.toString(),
   )
}
```

9. Below the `if` block, create a `val action` to navigate back to the `ItemListFragment`. Call *`findNavController`*`().navigate()`, passing in the `action`.

```
val action =
AddItemFragmentDirections.actionAddItemFragmentToItemListFragment()
findNavController().navigate(action)
```

Import `androidx.navigation.fragment.findNavController`.

10. The complete method should look like the following.

```
private fun addNewItem() {
       if (isEntryValid()) {
           viewModel.addNewItem(
               binding.itemName.text.toString(),
               binding.itemPrice.text.toString(),
               binding.itemCount.text.toString(),
           )
           val action =
AddItemFragmentDirections.actionAddItemFragmentToItemListFragment()
           findNavController().navigate(action)
       }
}
```

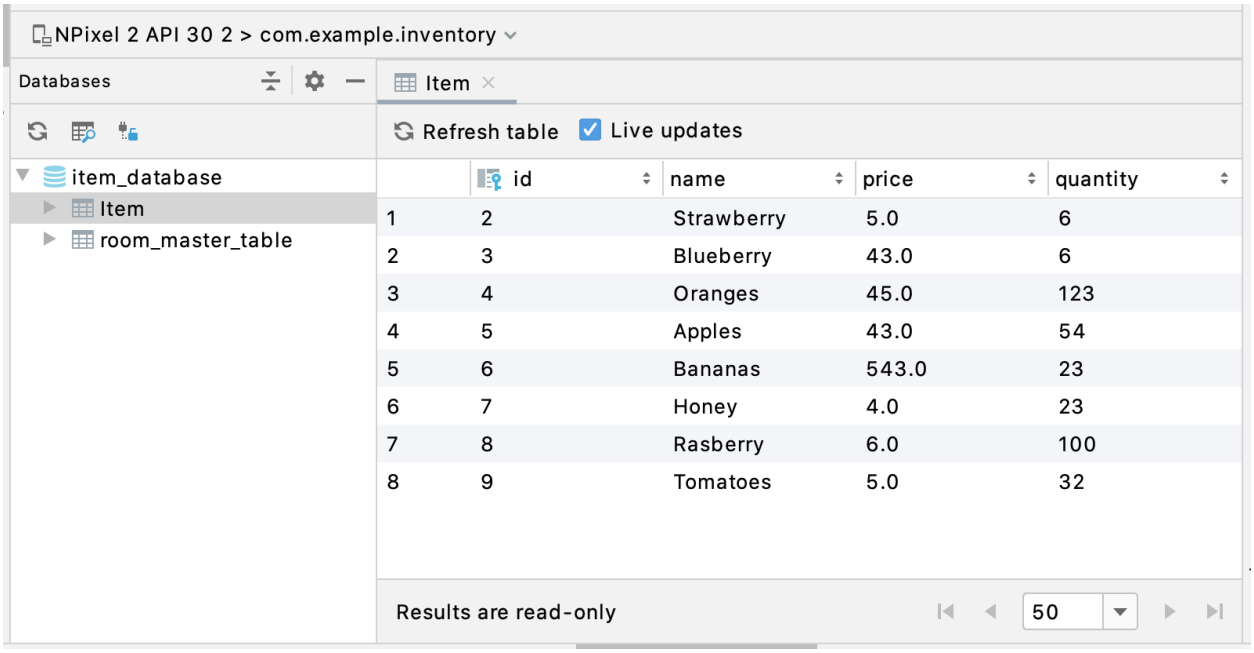
11. To tie everything together, add a click handler to the **Save** button. In the `AddItemFragment` class, above the `onDestroyView()` function, override the `onViewCreated()` function.
12. Inside the `onViewCreated()` function, add a click handler to the save button, and call `addNewItem()` from it.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
   super.onViewCreated(view, savedInstanceState)
   binding.saveAction.setOnClickListener {
       addNewItem()
   }
}
```

13. Build and run your app. Tap the + Fab. In the **Add Item** screen, add the item details and tap **Save**. This action saves the data, but you cannot see anything yet in the app. In the next task, you will use the Database Inspector to view the data you saved.

View the database using Database Inspector

1. Run your app on an emulator or connected device running API level 26 or higher, if you have not done so already. Database Inspector works best on emulator/devices running API level 26.
2. In Android studio, select **View** > **Tool Windows** > **Database Inspector** from the menu bar.
3. In the Database Inspector pane, select the `com.example.inventory` from the dropdown menu.
4. The **item_database** in the Inventory app appears in the **Databases** pane. Expand the node for the **item_database** and select **Item** to inspect. If your **Databases** pane is empty, use your emulator to add some items to the database using the **Add Item** screen.
5. Check the **Live updates** checkbox in the Database Inspector to automatically update the data it presents as you interact with your running app in the emulator or device.



Congratulations! You have created an app that can persist the data using Room. In the next codelab, you will add a `RecyclerView` to your app to display the items on the database and add new features to the app like deleting and updating the entities. See you there!

10. Solution code

The solution code for this codelab is in the GitHub repo and branch shown below.

Solution Code URL:

Practice: Build Bus Schedule app

1. Before you begin

Introduction

In the Persist Data with Room codelab, you learned how to implement a Room database in an Android app. This exercise provides the opportunity to gain more familiarity with the implementation of Room databases through an independently driven set of steps.

In this practice set, you take the concepts you learned from the Persist Data with Room codelab to complete the Bus Schedule app. This app presents the user with a list of bus stops and scheduled departures using data provided from a Room database.

The solution code is available at the end. To make the most of this learning experience, try to implement and troubleshoot as much as you can before you look at the provided solution code. It is during this hands-on time that you learn the most.

Prerequisites

- Android Basics with Compose coursework through the Persist Data with Room codelab

What you'll need

- A computer with internet access and Android Studio
- The Bus Schedule starter code

What you'll build

In this practice set, you complete the Bus Schedule app by implementing a database and then delivering data to the UI using the database. A database file in the asset directory found in the starter code provides data for the app. You load this data into a database and make it available for read usage by the app.

After you complete the app, it shows a list of bus stops and corresponding arrival times. You can click an item in the list to trigger navigation to a detail screen that provides data for that stop.

The completed app shows this data, loaded from a Room database:

3. Add dependencies

Add the following dependencies to the app:

app/build.gradle.kts

```
implementation("androidx.room:room-ktx:${rootProject.extra["room_version"]}")
implementation("androidx.room:room-runtime:${rootProject.extra["room_version"]}")
ksp("androidx.room:room-compiler:${rootProject.extra["room_version"]}")
```

You should get the most current stable version of room from the Room documentation and add the correct version number. At this moment the latest version is:

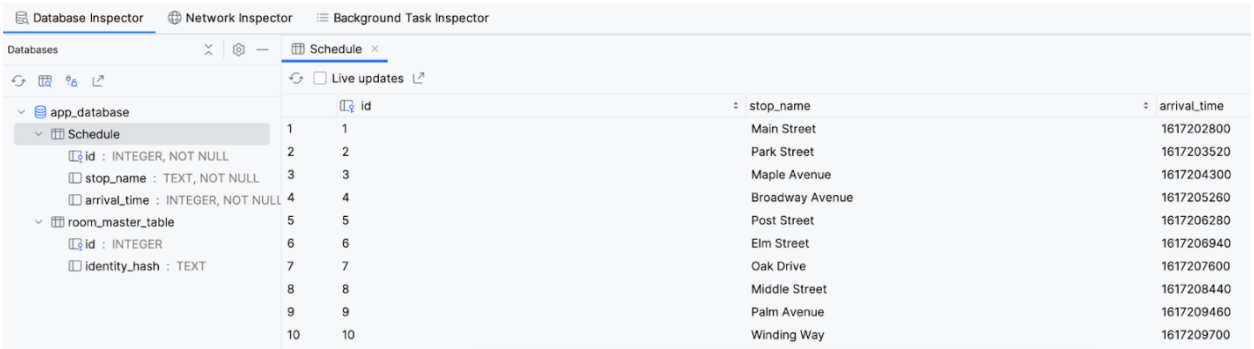
build.gradle.kts

```
set("room_version", "2.5.1")
```

4. Create a Room entity

Convert the current Bus Schedule data class into a Room Entity.

The following image shows a sample of what the final data table looks like, including the schema and Entity property.



5. Create a data access object

Create a data access object (DAO) to access the database. The DAO provides a method to retrieve all the items in the database and a method to retrieve a single item with the name of the bus stop. Make sure to order the schedule by arrival time.

6. Create a database instance

Create a Room database that uses the Entity and your DAO. The database initializes itself with data from the assets/database/bus_schedule.db file in the starter code.

7. Update the ViewModel

Update the ViewModel to retrieve data from the DAO and provide it to the UI instead of supplying sample data. Make sure to leverage both of your DAO methods to supply data for the list and for individual stops.

8. Solution code

Solution code URL:

https://github.com/google-developer-training/basic-android-kotlin-compose-training-bus-schedule-app

Branch name with solution code: main

Read and update data with Room

1. Before you begin

You have learned in the previous codelabs how to use a Room persistence library, an abstraction layer on top of a SQLite database to store the app data. In this codelab, you'll add more features to the Inventory app and learn how to read, display, update, and delete data from the SQLite database using Room. You will use a `RecyclerView` to display the data from the database and automatically update the data when the underlying data in the database is changed.

Prerequisites

- You know how to create and interact with the SQLite database using the Room library.
- You know how to create an entity, DAO, and database classes.
- You know how to use a data access object (DAO) to map Kotlin functions to SQL queries.
- You know how to display list items in a `RecyclerView`.
- You've taken the previous codelab in this unit, Persisting data with Room

What you'll learn

- How to read and display entities from a SQLite database.
- How to update and delete entities from a SQLite database using the Room library.

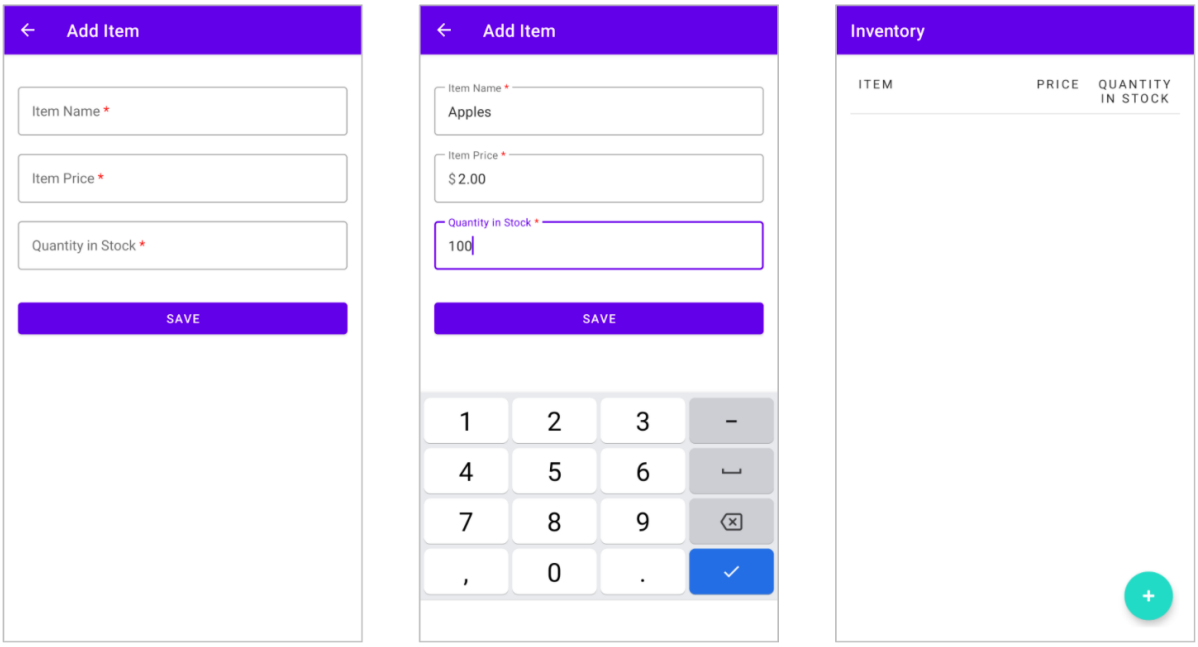
What you'll build

- You'll build an Inventory app that displays a list of inventory items. The app can update, edit, and delete items from the app database using Room.

2. Starter app overview

This codelab uses the Inventory app solution code from the previous codelab as the starter code. The starter app already saves data using the Room persistence library. The user can add data to the app database using the **Add Item** screen.

Note: The current version of the starter app doesn't display the date stored in the database.



In this codelab, you will extend the app to read and display the data, update and delete entities on the database using Room library.

Download the starter code for this codelab

This starter code is the same as the solution code from the previous codelab.

Starter Code URL: https://github.com/google-developer-training/android-basics-kotlin-inventory-app/tree/room

Branch name: `room`

To get the code for this codelab and open it in Android Studio, do the following.

Get the code

1. Click on the provided URL. This opens the GitHub page for the project in a browser.
2. On the GitHub page for the project, click the **Code** button, which brings up a dialog.

Notice that the price is displayed in the currency format. To convert a double value to the desired currency format, you will add an extension function to the `Item` class.

Extension Functions

Kotlin provides an ability to extend a class with new functionality without having to inherit from the class or modify the existing definition of the class. That means you can add functions to an existing class without having to access its source code. This is done via special declarations called *extensions*.

For example, you can write new functions for a class from a third-party library that you can't modify. Such functions are available for calling in the usual way, as if they were methods of the original class. These functions are called *extension functions*. (There are also *extension properties* that let you define new properties for existing classes, but these are outside the scope of this codelab.)

Extension functions don't actually modify the class, but allow you to use the dot-notation when calling the function on objects of that class.

For example, in the following code snippet you have a class called `Square`. This class has a property for the side and a function to calculate the area of the square. Notice the `Square.perimeter()` extension function, the function name is prefixed with the class it operates on. Inside the function, you can reference the public properties of the `Square` class.

Observe the extension function usage in the `main()` function. The created extension function, `perimeter()`, is called as a regular function inside that `Square` class.

Example:

```
class Square(val side: Double){
        fun area(): Double{
        return side * side;
    }
}

// Extension function to calculate the perimeter of the square
fun Square.perimeter(): Double{
        return 4 * side;
}

// Usage
fun main(args: Array<String>){
      val square = Square(5.5);
      val perimeterValue = square.perimeter()
      println("Perimeter: $perimeterValue")
      val areaValue = square.area()
      println("Area: $areaValue")
}
```

In this step, you will format the item price to a currency format string. In general, you don't want to change an entity class that represents data just to format the data (see single responsibility principle), so instead you'll add an extension function.

1. In `Item.kt`, below the class definition, add an extension function called `Item.getFormattedPrice()` that takes no parameters and returns a `String`. Notice the class name and the dot-notation in the function name.

```
fun Item.getFormattedPrice(): String =
   NumberFormat.getCurrencyInstance().format(itemPrice)
```

Import `java.text.NumberFormat`, when prompted by Android Studio.

Add ListAdapter

In this step, you'll add a list adapter to the `RecyclerView`. Since you're familiar with implementing the adapter from previous codelabs, the instructions are summarized below. The completed `ItemListAdapter` file is at the end of this step for your convenience, and to help increase your understanding of the Room concepts in the codelab.

1. In the `com.example.inventory` package, add a Kotlin class named `ItemListAdapter`. Pass in a function called `onItemClicked()` as a constructor parameter that takes in an `Item` object as parameter.
2. Change the `ItemListAdapter` class signature to extend `ListAdapter`. Pass in the `Item` and `ItemListAdapter.ItemViewHolder` as parameters.
3. Add the constructor parameter `DiffCallback`; the `ListAdapter` will use this to figure out what changed in the list.
4. Override the required methods `onCreateViewHolder()` and `onBindViewHolder()`.
5. The `onCreateViewHolder()` method returns a new `ViewHolder` when RecyclerView needs one.
6. Inside the `onCreateViewHolder()` method, create a new `View`, inflate it from the `item_list_item.xml` layout file using the auto generated binding class, `ItemListItemBinding`.
7. Implement the `onBindViewHolder()` method. Get the current item using the method `getItem()`, passing the position.
8. Set the click listener on the `itemView`, call the function `onItemClicked()` inside the listener.
9. Define the `ItemViewHolder` class, extend it from `RecyclerView.ViewHolder`. Override the `bind()` function, pass in the `Item` object.
10. Define a companion object. Inside the companion object, define a `val` of the type `DiffUtil.ItemCallback<Item>()` called `DiffCallback`. Override the required methods `areItemsTheSame()` and `areContentsTheSame()`, and define them.

The finished class should look like the following:

```
package com.example.inventory

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import com.example.inventory.data.Item
import com.example.inventory.data.getFormattedPrice
import com.example.inventory.databinding.ItemListItemBinding

/**
* [ListAdapter] implementation for the recyclerview.
*/

class ItemListAdapter(private val onItemClicked: (Item) -> Unit) :
   ListAdapter<Item, ItemListAdapter.ItemViewHolder>(DiffCallback) {

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ItemViewHolder {
       return ItemViewHolder(
           ItemListItemBinding.inflate(
               LayoutInflater.from(
                   parent.context
               )
           )
       )
   }

   override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
       val current = getItem(position)
       holder.itemView.setOnClickListener {
           onItemClicked(current)
       }
       holder.bind(current)
   }

   class ItemViewHolder(private var binding: ItemListItemBinding) :
       RecyclerView.ViewHolder(binding.root) {

       fun bind(item: Item) {

       }
   }

   companion object {
       private val DiffCallback = object : DiffUtil.ItemCallback<Item>() {
           override fun areItemsTheSame(oldItem: Item, newItem: Item):
Boolean {
               return oldItem === newItem
           }

           override fun areContentsTheSame(oldItem: Item, newItem: Item):
Boolean {
               return oldItem.itemName == newItem.itemName
           }
       }
```

11. In `ItemListAdapter.kt`, implement the `bind()` function in `ItemViewHolder` class. Bind the `itemName` TextView to `item.itemName`. Get the price in currency format using the `getFormattedPrice()` extension function, and bind it to the `itemPrice` TextView. Convert the `quantityInStock` value to `String`, and bind it to the `itemQuantity` TextView. The completed method should look like this:

```
fun bind(item: Item) {
    binding.apply {
        itemName.text = item.itemName
        itemPrice.text = item.getFormattedPrice()
        itemQuantity.text = item.quantityInStock.toString()
    }
}
```

When prompted by Android Studio, import `com.example.inventory.data.getFormattedPrice`.

Use ListAdapter

In this task, you will update the `InventoryViewModel` and the `ItemListFragment` to display the item details on the screen using the list adapter you created in the previous step.

1. At the beginning of the class `InventoryViewModel`, create a `val` named `allItems` of the type `LiveData<List<Item>>` for the items from the database. Don't worry about the error, you will fix it soon.

```
val allItems: LiveData<List<Item>>
```

Import `androidx.lifecycle.LiveData` when prompted by the Android Studio.

2. Call `getItems()` on `itemDao` and assign it to `allItems`. The `getItems()` function returns a `Flow`. To consume the data as a `LiveData` value, use the `asLiveData()` function. The finished definition should look like this:

```
val allItems: LiveData<List<Item>> = itemDao.getItems().asLiveData()
```

Import `androidx.lifecycle.asLiveData`, when prompted by the Android Studio.

3. In `ItemListFragment`, at the beginning of the class, declare a `private` immutable property called `viewModel` of the type `InventoryViewModel`. Use `by` delegate to hand off the property initialization to the `activityViewModels` class. Pass in the `InventoryViewModelFactory` constructor.

```
private val viewModel: InventoryViewModel by activityViewModels {
    InventoryViewModelFactory(
        (activity?.application as InventoryApplication).database.itemDao()
    )
}
```

Import `androidx.fragment.app.activityViewModels` when requested by the Android Studio.

4. Still within the `ItemListFragment`, scroll to function `onViewCreated()`. Below the call to `super.onViewCreated()`, declare a `val` named `adapter`. Initialize the new `adapter` property using the default constructor, `ItemListAdapter{}` passing in nothing.
5. Bind the newly created `adapter` to the `recyclerView` as follows:

```
val adapter = ItemListAdapter {
}
binding.recyclerView.adapter = adapter
```

6. Still inside `onViewCreated()`, after setting the adapter. Attach an observer on the `allItems` to listen for the data changes.
7. Inside the observer, call `submitList()` on the `adapter` and pass in the new list. This will update the RecyclerView with the new items on the list.

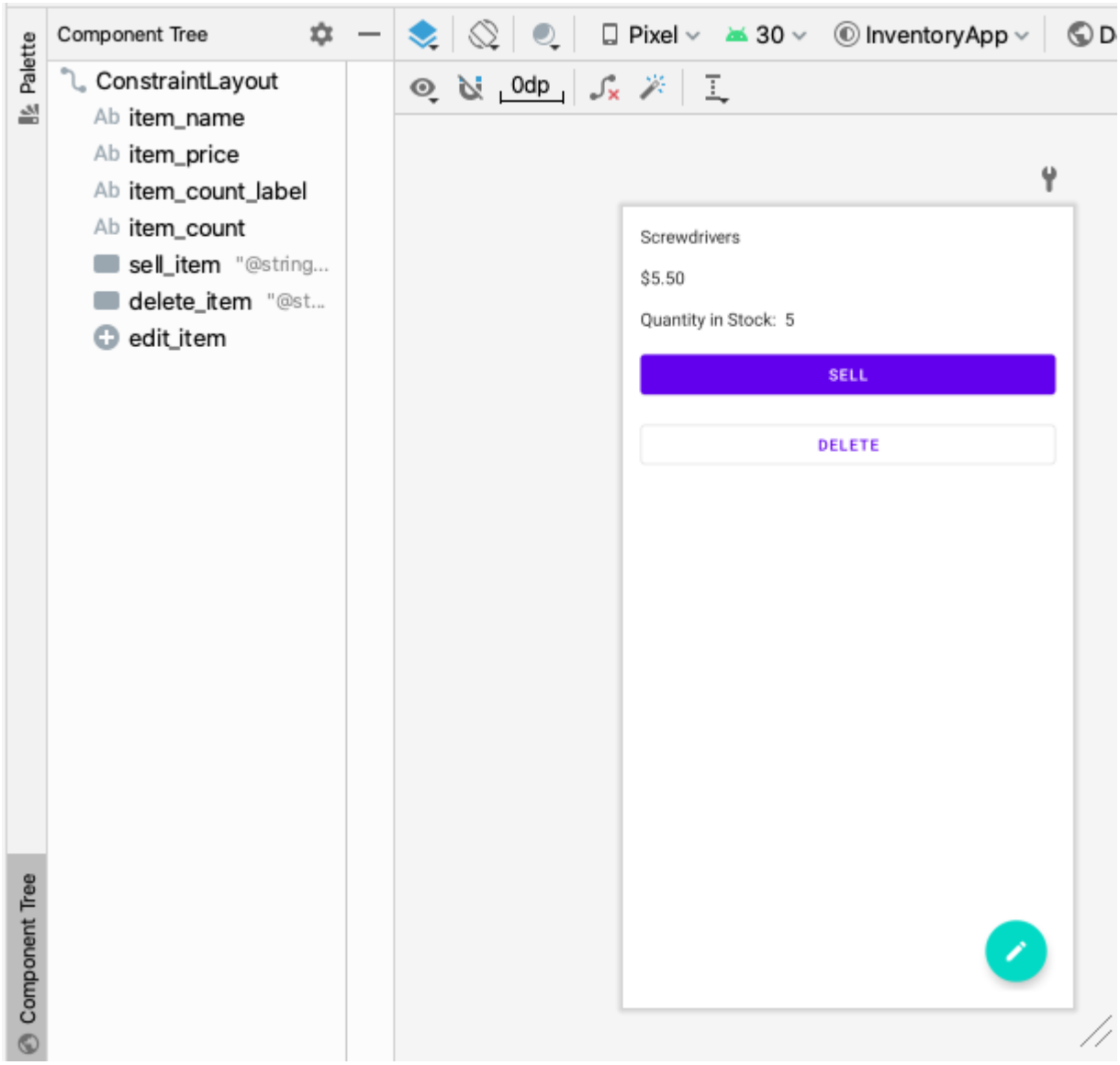
```
viewModel.allItems.observe(this.viewLifecycleOwner) { items ->
    items.let {
        adapter.submitList(it)
    }
}
```

8. Verify that the completed `onViewCreated()` method looks like the below. Run the app. Notice that the inventory list is displayed, if you items saved in your app database. Add some inventory items to the app database if the list is empty.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val adapter = ItemListAdapter {
        }
    binding.recyclerView.adapter = adapter
    viewModel.allItems.observe(this.viewLifecycleOwner) { items ->
        items.let {
            adapter.submitList(it)
        }
    }
    binding.recyclerView.layoutManager = LinearLayoutManager(this.context)
    binding.floatingActionButton.setOnClickListener {
        val action =
ItemListFragmentDirections.actionItemListFragmentToAddItemFragment(
            getString(R.string.add_fragment_title)
        )
        this.findNavController().navigate(action)
    }
}
```

4. Display item details

In this task, you will read and display the entity details on the **Item Details** screen. You will use the primary key (the item `id`) to read the details, such as name, price and quantity from the inventory app database and display them on the **Item Details** screen using the `fragment_item_detail.xml` layout file. The layout file `fragment_item_detail.xml` is predesigned for you and contains three TextViews that display the item details.



You will implement the following steps in this task:

- Add a click handler to the RecyclerView to navigate the app to the **Item Details** screen.
- In the `ItemListFragment` fragment, retrieve the data from the database and display.

- Bind the TextViews to the ViewModel data.

Add a click handler

1. In `ItemListFragment`, scroll to the `onViewCreated()` function to update the adapter definition.
2. Add a lambda as a constructor parameter to the `ItemListAdapter{}`.

```
val adapter = ItemListAdapter {
}
```

3. Inside the lambda, create a `val` called `action`. You will fix the initialization error soon.

```
val adapter = ItemListAdapter {
    val action
}
```

4. Call `actionItemListFragmentToItemDetailFragment()` method on the `ItemListFragmentDirections` passing in the item `id`. Assign the returned `NavDirections` object to `action`.

```
val adapter = ItemListAdapter {
   val action
=    ItemListFragmentDirections.actionItemListFragmentToItemDetailFragment(it
.id)
}
```

5. Below the `action` definition, retrieve a `NavController` instance using `this.`*`findNavController`*`()` and call `navigate()` on it pasing in the `action`. The adapter definition should look like this:

```
val adapter = ItemListAdapter {
   val action
=   ItemListFragmentDirections.actionItemListFragmentToItemDetailFragment(it.
id)
   this.findNavController().navigate(action)
}
```

6. Run the app. Click on an item in the `RecyclerView`. The app navigates to the **Item Details** screen. Notice that the details are blank. Tap on the buttons, nothing happens.

In later steps you will display the entity details on the **Item Details** screen and add functionality to sell and delete buttons.

Retrieve item details

In this step, you will add a new function to the `InventoryViewModel`, to retrieve the item details from the database based on the item `id`. In the next step, you will use this function to display the entity details on the **Item Details** screen.

1. In `InventoryViewModel`, add a function named `retrieveItem()` that takes an `Int` for the item id and returns a `LiveData<Item>`. You will fix the return expression error soon.

```
fun retrieveItem(id: Int): LiveData<Item> {
}
```

2. Inside the new function, call `getItem()` on the `itemDao`, passing in the parameter `id`. The `getItem()` function returns a `Flow`. To consume the `Flow` value as `LiveData` call `asLiveData()` function and use this as the return of `retrieveItem()` function. The completed function should look like the following:

```
fun retrieveItem(id: Int): LiveData<Item> {
   return itemDao.getItem(id).asLiveData()
}
```

Bind data to the TextViews

In this step, you will create a ViewModel instance in the `ItemDetailFragment` and bind the ViewModel data to the TextViews in the **Item Details** screen. You will also attach an observer to the data in the ViewModel to keep your inventory list updated on the screen, if underlying data in the database changes.

1. In `ItemDetailFragment`, add a mutable property called `item` of the type `Item` entity. You will use this property to store information about a single entity. This property will be initialized later, so prefix it with `lateinit`.

```
lateinit var item: Item
```

Import `com.example.inventory.data.Item`, when prompted by the Android Studio.

2. At the beginning of the class `ItemDetailFragment`, declare a `private` immutable property called `viewModel` of the type `InventoryViewModel`. Use `by` delegate to hand off the property initialization to the `activityViewModels` class. Pass in the `InventoryViewModelFactory` constructor.

```
private val viewModel: InventoryViewModel by activityViewModels {
   InventoryViewModelFactory(
```

```
        (activity?.application as InventoryApplication).database.itemDao()
    )
}
```

Import `androidx.fragment.app.activityViewModels`, if prompted by Android Studio.

3. Still in `ItemDetailFragment`, create a `private` function called `bind()` that takes an instance of the `Item` entity as the parameter and returns nothing.

```
private fun bind(item: Item) {
}
```

4. Implement the `bind()` function, this is similar to what you have done in the `ItemListAdapter`. Set the `text` property of `itemName` TextView to `item.itemName`. Call *`getFormattedPrice()`* on the `item` property to format the price value, and set it to the `text` property of `itemPrice` TextView. Convert the `quantityInStock` to `String` and set it to the `text` property of `itemQuantity` TextView.

```
private fun bind(item: Item) {
    binding.itemName.text = item.itemName
    binding.itemPrice.text = item.getFormattedPrice()
    binding.itemCount.text = item.quantityInStock.toString()
}
```

5. Update the `bind()` function to use the `apply{}` scope function to the code block as shown below.

```
private fun bind(item: Item) {
    binding.apply {
        itemName.text = item.itemName
        itemPrice.text = item.getFormattedPrice()
        itemCount.text = item.quantityInStock.toString()
    }
}
```

6. Still in `ItemDetailFragment`, override `onViewCreated()`.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
}
```

7. In one of the previous steps, you passed item id as a navigation argument to `ItemDetailFragment` from the `ItemListFragment`. Inside `onViewCreated()`, below the call to the super class function, create an immutable variable called `id`. Retrieve and assign the navigation argument to this new variable.

```
val id = navigationArgs.itemId
```

8. Now you'll use this `id` variable to retrieve the item details. Still inside `onViewCreated()`, call the `retrieveItem()` function on the `viewModel` passing in the `id`. Attach an observer to the returned value passing in the `viewLifecycleOwner` and a lambda.

```
viewModel.retrieveItem(id).observe(this.viewLifecycleOwner) {
    }
```

9. Inside the lambda, pass in `selectedItem` as the parameter which contains the `Item` entity retrieved from the database. In the lambda function body, assign `selectedItem` value to `item`. Call `bind()` function passing in the `item`. The completed function should look like the following.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    val id = navigationArgs.itemId
    viewModel.retrieveItem(id).observe(this.viewLifecycleOwner) { selectedItem
->
        item = selectedItem
        bind(item)
    }
}
```

10. Run the app. Click on any list element on the **Inventory** screen, **Item Details** screen is displayed. Notice that now the screen is not blank any more, it displays the entity details retrieved from the inventory database.

11. Tap on the **Sell**, **Delete**, and FAB buttons. Nothing happens! In next tasks, you'll implement the functionality of these buttons.

5. Implement sell item

In this task, you will extend the features of the app, implement sell functionality. Here is a high level gist of the instructions for this step.

- Add a function in the ViewModel to update an entity
- Create a new method to reduce the quantity and update the entity in the app database.
- Attach a click listener to the **Sell** button
- Disable the **Sell** button if the quantity is zero.

Let's code:

1. In `InventoryViewModel`, add a private function called `updateItem()` that takes an instance of the entity class, `Item` and returns nothing.

```
private fun updateItem(item: Item) {
}
```

2. Implement the new method, `updateItem()`. To call `update()` suspend method from the `ItemDao` class, launch a coroutine using the `viewModelScope`. Inside the launch block, make a call to the `update()` function on `itemDao` passing in the `item`. Your completed method should look like the following.

```
private fun updateItem(item: Item) {
    viewModelScope.launch {
        itemDao.update(item)
    }
}
```

3. Still inside the `InventoryViewModel`, add another method called `sellItem()` that takes an instance of the `Item` entity class and returns nothing.

```
fun sellItem(item: Item) {
}
```

4. Inside the `sellItem()` function, add an `if` condition to check whether the `item.quantityInStock` is greater than `0`.

```
fun sellItem(item: Item) {
    if (item.quantityInStock > 0) {
    }
}
```

Inside the `if` block you will use `copy()` function for Data class to update the entity.

Data class: copy()

The `copy()` function is provided by default to all the instances of data classes. This function is used to copy an object for changing some of its properties, but keeping the rest of the properties unchanged.

For example, consider the `User` class and its instance `jack` as shown below. If you want to create a new instance with only updating the `age` property, its implementation would be as follows:

Example

```
// Data class
data class User(val name: String = "", val age: Int = 0)

// Data class instance
val jack = User(name = "Jack", age = 1)

// A new instance is created with its age property changed, rest of the
properties unchanged.
val olderJack = jack.copy(age = 2)
```

5. Back to the `sellItem()` function in the `InventoryViewModel`. Inside the `if` block, create a new immutable property called `newItem`. Call `copy()` function on the `item` instance passing in the updated `quantityInStock`, that is decreasing the stock by `1`.

```
val newItem = item.copy(quantityInStock = item.quantityInStock - 1)
```

6. Below the definition of the `newItem`, make a call to the `updateItem()` function passing in the new updated entity, that is `newItem`. The completed method should look like the following.

```
fun sellItem(item: Item) {
   if (item.quantityInStock > 0) {
       // Decrease the quantity by 1
       val newItem = item.copy(quantityInStock = item.quantityInStock - 1)
       updateItem(newItem)
   }
}
```

7. To add the selling stock feature, go to `ItemDetailFragment`. Scroll to the end of the `bind()` function. Inside the `apply` block, set a click listener to the **Sell** button and call the `sellItem()` function on `viewModel`.

```
private fun bind(item: Item) {
binding.apply {

...
    sellItem.setOnClickListener { viewModel.sellItem(item) }
```

10. To give users better feedback, you might want to disable the **Sell** button when there is no item to sell. In `InventoryViewModel`, add a function to check if the quantity is greater than `0`. Name the function `isStockAvailable()`, that takes an `Item` instance and returns a `Boolean`.

```
fun isStockAvailable(item: Item): Boolean {
   return (item.quantityInStock > 0)
}
```

11. Go to `ItemDetailFragment`, scroll to the `bind()` function. Inside the apply block, call the `isStockAvailable()` function on `viewModel` passing in the `item`. Set the return value to `isEnabled` property of the **Sell** button. Your code should look something like this.

```
private fun bind(item: Item) {
   binding.apply {
        ...
        sellItem.isEnabled = viewModel.isStockAvailable(item)
        sellItem.setOnClickListener { viewModel.sellItem(item) }
   }
}
```

12. Run your app, notice that the **Sell** button is disabled when the quantity in stock is zero. Congratulations on implementing the sell item feature to your app.

Delete item entity

Similar to the previous task, you will extend the features of your app further by implementing delete functionality. Here are the high-level instructions for this step, it's much easier than implementing the sell feature.

- Add a function in the ViewModel to delete an entity from the database
- Add a new method in the `ItemDetailFragment` to call the new delete function and handle navigation.
- Attach a click listener to the **Delete** button.

Let's continue to code:

1. In `InventoryViewModel`, add a new function called `deleteItem()`, which takes an instance of the `Item` entity class called `item` and returns nothing. Inside the `deleteItem()` function, launch a coroutine with `viewModelScope`. Inside the `launch` block call the `delete()` method on `itemDao` passing in the `item`.

```
fun deleteItem(item: Item) {
   viewModelScope.launch {
       itemDao.delete(item)
   }
}
```

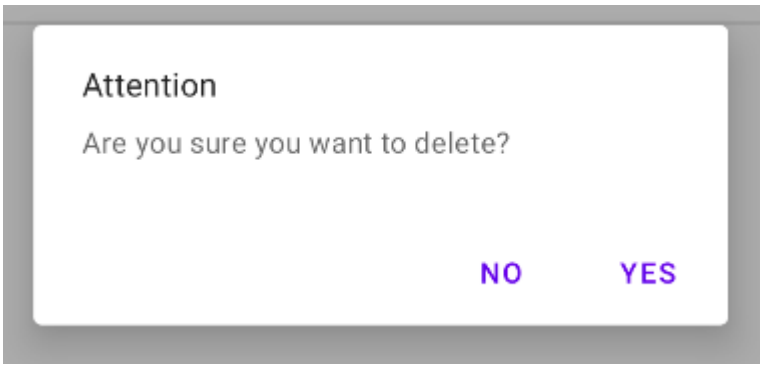
2. In `ItemDetailFragment`, scroll to the beginning of the `deleteItem()` function. Call `deleteItem()` on the `viewModel`, pass in the `item`. The `item` instance contains the entity currently displayed on the **Item Details** screen. Your completed method should look like this.

```
private fun deleteItem() {
   viewModel.deleteItem(item)
   findNavController().navigateUp()
}
```

3. Still within `ItemDetailFragment`, scroll to the `showConfirmationDialog()` function. This function is given for you as part of the starter code. This method displays an alert dialog to get the user's confirmation before deleting the item and calls `deleteItem()` function when the positive button is tapped.

```
private fun showConfirmationDialog() {
        MaterialAlertDialogBuilder(requireContext())
            ...
            .setPositiveButton(getString(R.string.yes)) { _, _ ->
                deleteItem()
            }
            .show()
    }
```

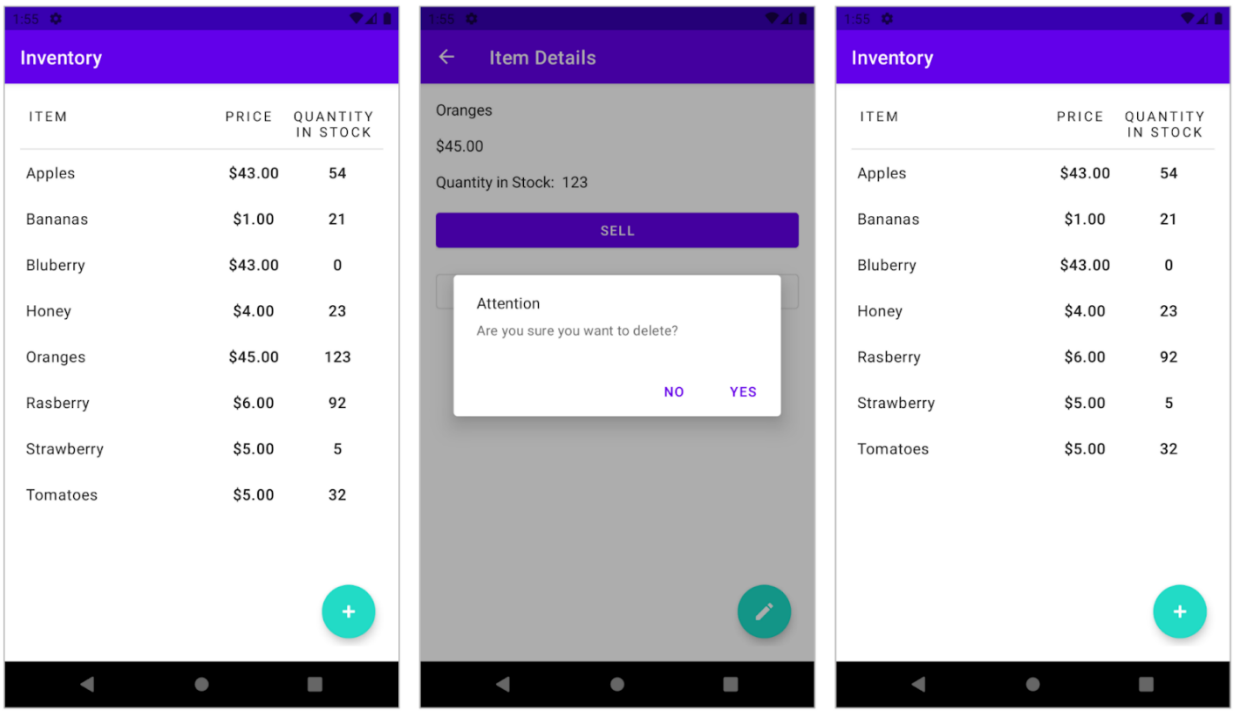
The `showConfirmationDialog()` function displays a alert dialog which looks like the following:



4. In `ItemDetailFragment`, at the end of `bind()` function, inside the `apply` block, set the click listener to the delete button. Call `showConfirmationDialog()` inside the click listener lambda.

```
private fun bind(item: Item) {
    binding.apply {
        ...
        deleteItem.setOnClickListener { showConfirmationDialog() }
    }
}
```

5. Run your app! Select a list element on the Inventory list screen, in the **Item Details** screen tap **Delete** button. Tap Yes, the app navigates back to the Inventory screen. Notice that the entity you deleted is no longer in the app database. Congratulations on implementing the delete feature.



Edit item entity

Similar to the previous tasks, in this task you will add another feature enhancement to the app. You will implement the edit item entity.

Here is a quick run through of the steps to edit an entity in the app database:

- Reuse the **Add Item** screen by updating the fragment title to Edit Item.
- Add click listener to the FAB, to navigate to the **Edit Item** screen.
- Populate the TextViews with the entity details.
- Update the entity in the database using Room.

Add click listener to the FAB

1. In `ItemDetailFragment`, add a new `private` function called `editItem()` that takes no parameters and returns nothing. In the next step, you will be reusing the `fragment_add_item.xml`, by updating the screen title to **Edit Item**. To achieve this you will send the fragment title string along with item id as part of the action.

```
private fun editItem() {
}
```

After you update the fragment title the **Edit Item** screen should look like the following.

2. Inside `editItem()` function, create an immutable variable called `action`. Make a call to `actionItemDetailFragmentToAddItemFragment()` on `ItemDetailFragmentDirections` passing in title string, `edit_fragment_title` and the item `id`. Assign the returned value to `action`. Below the definition of `action`, call `this.findNavController().navigate()` passing in the `action` to navigate to the **Edit Item** screen.

```
private fun editItem() {
   val action =
ItemDetailFragmentDirections.actionItemDetailFragmentToAddItemFragment(
       getString(R.string.edit_fragment_title),
       item.id
   )
   this.findNavController().navigate(action)
}
```

3. Still within `ItemDetailFragment`, scroll to the `bind()` function. Inside the `apply` block, set the click listener to the FAB, call the `editItem()` function from the lambda to navigate to the **Edit Item** screen.

```
private fun bind(item: Item) {
   binding.apply {
       ...
       editItem.setOnClickListener { editItem() }
   }
}
```

4. Run the app. Go to the **Item Details** screen. Click on FAB. Notice the title of the screen is updated to Edit Item, but all text fields are empty. In the next step, you'll fix this.

```
val price = "%.2f".format(item.itemPrice)
```

3. Below the `price` definition, use the `apply` scope function on the `binding` property as shown below.

```
binding.apply {
}
```

4. Inside the `apply` scope function code block, Set `item.itemName` to the text property of the `itemName`. Use `setText()` function and pass in `item.itemName` string and `TextView.BufferType.SPANNABLE` as `BufferType`.

```
binding.apply {
    itemName.setText(item.itemName, TextView.BufferType.SPANNABLE)
}
```

Import `android.widget.TextView`, if prompted by Android Studio.

5. Similar to the above step, set the text property of the price `EditText` as shown below. For setting `text` property of quantity EditText remember to convert the `item.quantityInStock` to `String`. Your completed function should look like this.

```
private fun bind(item: Item) {
    val price = "%.2f".format(item.itemPrice)
    binding.apply {
        itemName.setText(item.itemName, TextView.BufferType.SPANNABLE)
        itemPrice.setText(price, TextView.BufferType.SPANNABLE)
        itemCount.setText(item.quantityInStock.toString(),
TextView.BufferType.SPANNABLE)
    }
}
```

6. Still inside the `AddItemFragment`, scroll to the `onViewCreated()` function. After the call to the super class function. Create a `val` called `id` and retrieve `itemId` from the navigation arguments.

```
val id = navigationArgs.itemId
```

7. Add an `if-else` block with a condition to check whether `id` is greater than zero and move the **Save** button click listener into the `else` block. Inside the `if` block retrieve the entity using the `id` and add an observer on it. Inside the observer, update the `item` property and call `bind()` passing in the `item`. The complete function is provided for you to copy-paste. It is simple and easy to understand; you are left to decipher it on your own.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    val id = navigationArgs.itemId
    if (id > 0) {
        viewModel.retrieveItem(id).observe(this.viewLifecycleOwner) {
selectedItem ->
```

It's coding time again!

1. In `InventoryViewModel`, add a `private` function called `getUpdatedItemEntry()` that takes in an `Int`, and three strings for the entity details named `itemName`, `itemPrice` and `itemCount`. Return an instance of the `Item` from the function. Code is given for your reference.

```
private fun getUpdatedItemEntry(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
): Item {
}
```

2. Inside the `getUpdatedItemEntry()` function create an Item instance using the function parameters, as shown below. Return the `Item` instance from the function.

```
private fun getUpdatedItemEntry(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
): Item {
    return Item(
        id = itemId,
        itemName = itemName,
        itemPrice = itemPrice.toDouble(),
        quantityInStock = itemCount.toInt()
    )
}
```

3. Still inside the `InventoryViewModel`, add another function named `updateItem()`. This function also takes an `Int` and three strings for the entity details and returns nothing. Use the variable names from the following code snippet.

```
fun updateItem(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
) {
}
```

4. Inside the `updateItem()` function make a call to the `getUpdatedItemEntry()` function passing in the entity information, which are passed in as function parameters, as shown below. Assign the returned value to an immutable variable called `updatedItem`.

```
val updatedItem = getUpdatedItemEntry(itemId, itemName, itemPrice, itemCount)
```

5. Just below the call to the `getUpdatedItemEntry()` function, make a call to the `updateItem()` function passing in the `updatedItem`. The completed function looks like this:

```
fun updateItem(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
) {
    val updatedItem = getUpdatedItemEntry(itemId, itemName, itemPrice,
itemCount)
    updateItem(updatedItem)
}
```

6. Go back to `AddItemFragment`, add a private function called `updateItem()` with no parameters and return nothing. Inside the function add an `if` condition to validate the user input by calling the function `isEntryValid()`.

```
private fun updateItem() {
    if (isEntryValid()) {
    }
}
```

7. Inside the `if` block, make a call to `viewModel.updateItem()` passing the entity details. Use the `itemId` from the navigation arguments, and the other entity details like name, price and quantity from the EditTexts as shown below.

```
viewModel.updateItem(
     this.navigationArgs.itemId,
     this.binding.itemName.text.toString(),
     this.binding.itemPrice.text.toString(),
     this.binding.itemCount.text.toString()
)
```

8. Below the `updateItem()` function call, define an `val` called `action`. Call `actionAddItemFragmentToItemListFragment()` on `AddItemFragmentDirections` and assign the returned value to `action`. Navigate to `ItemListFragment`, call `findNavController().navigate()` passing in the `action`.

```
private fun updateItem() {
    if (isEntryValid()) {
        viewModel.updateItem(
            this.navigationArgs.itemId,
            this.binding.itemName.text.toString(),
            this.binding.itemPrice.text.toString(),
            this.binding.itemCount.text.toString()
        )
        val action =
AddItemFragmentDirections.actionAddItemFragmentToItemListFragment()
        findNavController().navigate(action)
```