Test the Cupcake App

1. Introduction

In the Navigate between screens with Compose codelab, you learned how to add navigation to a Compose app using the Jetpack Navigation Compose component.

The Cupcake app has multiple screens to navigate through and a variety of actions that can be taken by the user. This app provides a great opportunity for you to hone your automated testing skill! In this codelab, you will write a number of UI tests for the Cupcake app and learn how to approach maximizing test coverage.

Prerequisites

- Familiarity with the Kotlin language, including function types, lambdas, and scope functions
- Completion of the Navigate between screens with Compose codelab

What you'll learn

- Test the Jetpack Navigation component with Compose.
- Create a consistent UI state for each UI test.
- Create helper functions for tests.

What you'll build

- UI tests for the Cupcake app

What you'll need

- The latest version of Android Studio
- An internet connection to download the starter code

2. Download the starter code

Starter code URL: https://github.com/google-developer-training/basic-android-kotlin-compose-training-cupcake

Branch name with starter code: `navigation`

1. In Android Studio, open the basic-android-kotlin-compose-training-cupcake folder.
2. Open the Cupcake app code in Android Studio.

3. Set up Cupcake for UI tests

Add the androidTest dependencies

The Gradle build tool enables you to add dependencies for specific modules. This functionality prevents dependencies from being compiled unnecessarily. You are already familiar with the `implementation` configuration when including dependencies in a project. You've used this keyword to import dependencies in the app module's `build.gradle.kts` file. Using the `implementation` keyword makes that dependency available to all source sets in that module; at this point in the course, you gained experience with the `main`, `test`, and `androidTest` source sets.

UI tests are contained in their own source sets called `androidTest`. Dependencies that are only needed for this module don't need to be compiled for other modules, such as the `main` module, where the app code is contained. When adding a dependency that is only used by UI tests, use the `androidTestImplementation` keyword to declare the dependency in the app module's `build.gradle.kts` file. Doing so ensures that the UI test dependencies are compiled only when you run UI tests.

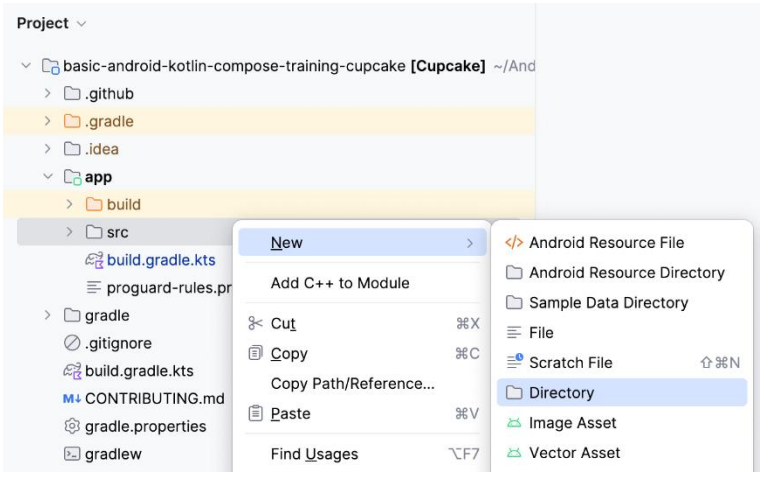
Complete the following steps to add the dependencies necessary to write UI tests:

1. Open the `build.gradle.kts(Module :app)` file.
2. Add the following dependencies to the `dependencies` section of the file:

```
androidTestImplementation(platform("androidx.compose:compose-bom:2023.05.01"))
androidTestImplementation("androidx.compose.ui:ui-test-junit4")
androidTestImplementation("androidx.navigation:navigation-testing:2.6.0")
androidTestImplementation("androidx.test.espresso:espresso-intents:3.5.1")
androidTestImplementation("androidx.test.ext:junit:1.1.5")
```

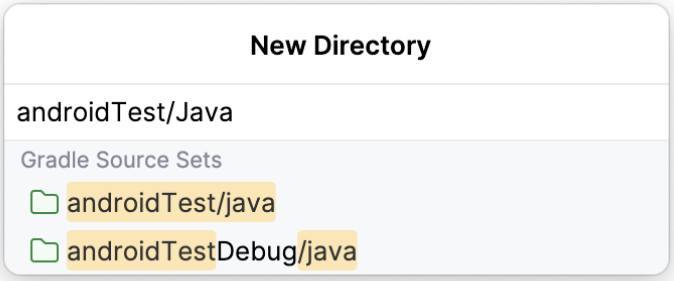
Create the UI test directory

1. Right-click the `src` directory in the project view and select **New** > **Directory**.



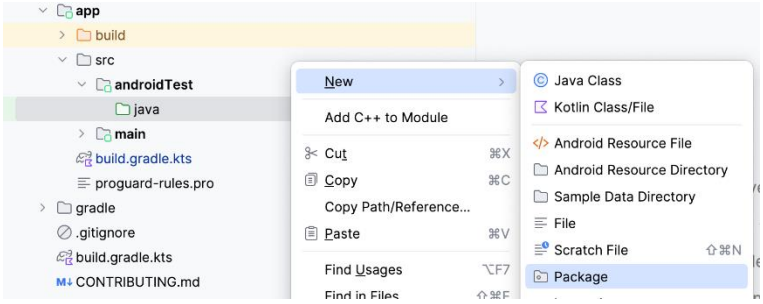
2. Select the **androidTest/java** option.

Note: You may have to scroll down to find this option.

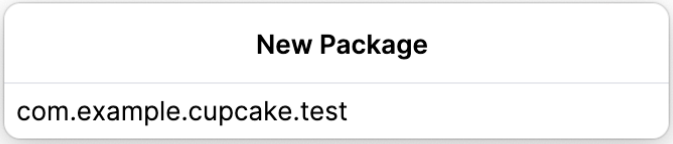


Create the test package

1. Right-click the `androidTest/java` directory in the project window, and select **New** > **Package**.

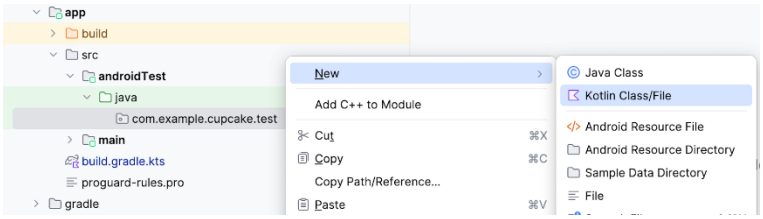


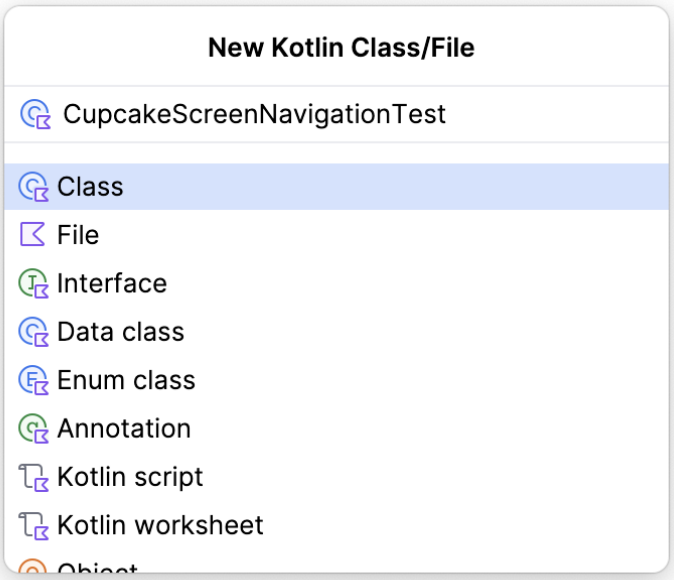
2. Name the package c**om.example.cupcake.test**.



Create the navigation test class

In the `test` directory, create a new Kotlin class called `CupcakeScreenNavigationTest`.





4. Set up the nav host

In a previous codelab, you learned that UI tests in Compose require a Compose test rule. The same is true of testing Jetpack Navigation. However, testing navigation requires some additional setup through the Compose test rule.

When testing Compose Navigation, you won't have access to the same `NavHostController` that you do in the app code. However, you can use a `TestNavHostController` and configure the test rule with this nav controller. In this section, you learn how to configure and reuse the test rule for navigation tests.

1. In `CupcakeScreenNavigationTest.kt`, create a test rule using `createAndroidComposeRule` and passing `ComponentActivity` as the type parameter.

```
import androidx.activity.ComponentActivity
import
androidx.compose.ui.test.junit4.createAndroidComposeRule
```

```
import org.junit.Rule

@get:Rule
val composeTestRule =
createAndroidComposeRule<ComponentActivity>()
```

To make sure that your app navigates to the correct place, you need to reference a `TestNavHostController` instance to check the navigation route of the nav host when the app takes actions to navigate.

2. Instantiate a `TestNavHostController` instance as a `lateinit` variable. In Kotlin, the `lateinit` keyword is used to declare a property that can be initialized after the object has been declared.

```
import androidx.navigation.testing.TestNavHostController

private lateinit var navController: TestNavHostController
```

Next, specify the composable you want to use for the UI tests.

3. Create a method called `setupCupcakeNavHost()`.
4. In the `setupCupcakeNavHost()` method, call the `setContent()` method on the Compose test rule you created.
5. Inside of the lambda passed to the `setContent()` method, call the `CupcakeApp()` composable.

```
import com.example.cupcake.CupcakeApp

fun setupCupcakeNavHost() {
    composeTestRule.setContent {
        CupcakeApp()
    }
}
```

You now need to create the `TestNavHostContoller` object in the test class. You use this object later to determine the navigation state, as the app uses the controller to navigate the various screens in the Cupcake app.

6. Set up the nav host by using the lambda you created previously. Initialize the `navController` variable you created, register a navigator, and pass that `TestNavHostController` to the `CupcakeApp` composable.

```
import androidx.compose.ui.platform.LocalContext

fun setupCupcakeNavHost() {
    composeTestRule.setContent {
        navController =
```

```
TestNavHostController(LocalContext.current).apply {
            navigatorProvider.addNavigator(ComposeNavigator())
        }
        CupcakeApp(navController = navController)
    }
}
```

Every test in the `CupcakeScreenNavigationTest` class involves testing an aspect of navigation. Therefore, each test depends on the `TestNavHostController` object you created. Instead of having to manually call the `setupCupcakeNavHost()` function for every test to set up the nav controller, you can make that happen automatically using the `@Before` annotation provided by the **junit** library. When a method is annotated with `@Before`, it runs before every method annotated with `@Test`.

7. Add the `@Before` annotation to the `setupCupcakeNavHost()` method.

```
import org.junit.Before

@Before
fun setupCupcakeNavHost() {
    composeTestRule.setContent {
        navController =
TestNavHostController(LocalContext.current).apply {
            navigatorProvider.addNavigator(ComposeNavigator())
        }
        CupcakeApp(navController = navController)
    }
}
```

5. Write navigation tests

Verify the start destination

Recall that, when you built the Cupcake app, you created an `enum` class called `CupcakeScreen` that contained constants to dictate navigation of the app.

CupcakeScreen.kt

```
/**
* enum values that represent the screens in the app
*/
enum class CupcakeScreen(@StringRes val title: Int) {
   Start(title = R.string.app_name),
```

```
    Flavor(title = R.string.choose_flavor),
    Pickup(title = R.string.choose_pickup_date),
    Summary(title = R.string.order_summary)
}
```

All apps that have a UI have a home screen of some kind. For Cupcake, that screen is the **Start Order Screen**. The navigation controller in the `CupcakeApp` composable uses the `Start` item of the `CupcakeScreen` enum to determine when to navigate to this screen. When the app starts, if a destination route doesn't already exist, the nav host destination route is set to `CupcakeScreen.Start.name`.

You first need to write a test to verify that the **Start Order Screen** is the current destination route when the app starts.

1. Create a function called `cupcakeNavHost_verifyStartDestination()` and annotate it with `@Test`.

```
import org.junit.Test

@Test
fun cupcakeNavHost_verifyStartDestination() {
}
```

You now must confirm that the nav controller's initial destination route is the **Start Order Screen**.

2. Assert that the expected route name (in this case, `CupcakeScreen.Start.name`) is equal to the destination route of the nav controller's current back stack entry.

```
import org.junit.Assert.assertEquals
...

@Test
fun cupcakeNavHost_verifyStartDestination() {
    assertEquals(CupcakeScreen.Start.name,
navController.currentBackStackEntry?.destination?.route)
}
```

Note: The `AndroidComposeTestRule` you created automatically launches the app, displaying the `CupcakeApp` composable before the execution of any `@Test` method. Therefore, you do not need to take any extra steps in the test methods to launch the app.

Create helper methods

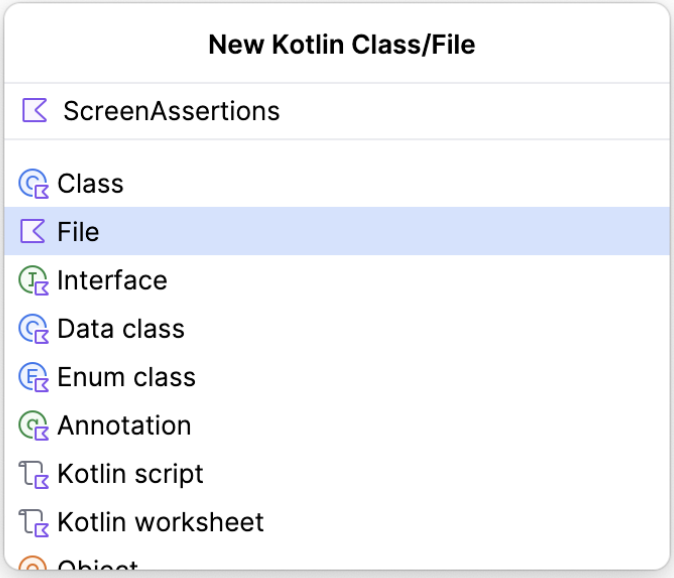
UI tests often require the repeating of steps to put the UI in a state in which a particular piece of the UI can be tested. A custom UI can also require complex assertions that require multiple lines of code. The assertion you wrote in the previous section requires a lot of code, and you are using this same assertion many times as you test navigation in the Cupcake app. In these situations, writing helper methods in your tests saves you from writing duplicate code.

For each navigation test you write, you use the `name` property of the `CupcakeScreen` enum items to check that the current destination route of the navigation controller is correct. You write a helper function that you can call whenever you want to make such an assertion.

Complete the following steps to create this helper function:

1. Create an empty Kotlin file in the `test` directory called `ScreenAssertions`.

Note: Make sure to create a **File** and not a **Class**. This file will be used to create an extension function which will require an empty Kotlin file.



2. Add an extension function to the `NavController` class called `assertCurrentRouteName()` and pass a string for the expected route name in the method signature.

```
fun NavController.assertCurrentRouteName(expectedRouteName:
String) {

}
```

3. In this function, assert that the `expectedRouteName` is equal to the destination route of the nav controller's current back stack entry.

```
import org.junit.Assert.assertEquals
...

fun NavController.assertCurrentRouteName(expectedRouteName:
String) {
    assertEquals(expectedRouteName,
currentBackStackEntry?.destination?.route)
}
```

4. Open the `CupcakeScreenNavigationTest` file and modify the `cupcakeNavHost_verifyStartDestination()` function to use your new extension function instead of the lengthy assertion.

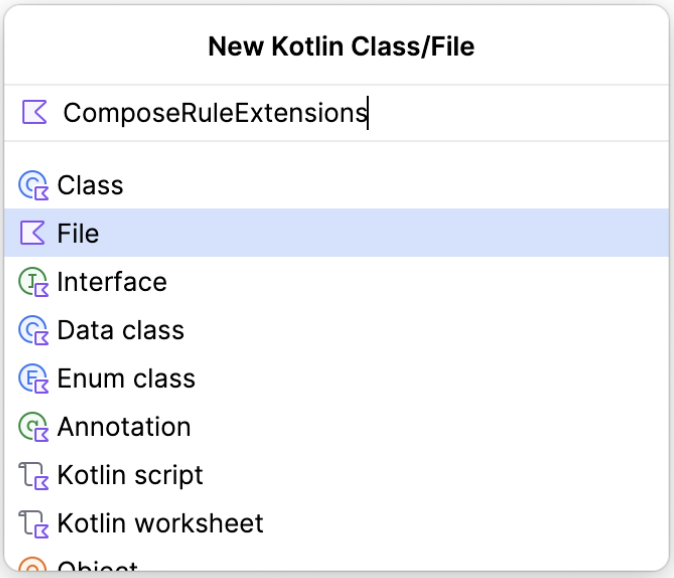
```
@Test
fun cupcakeNavHost_verifyStartDestination() {
    navController.assertCurrentRouteName(CupcakeScreen.Start.n
ame)
}
```

A number of tests also require interacting with UI components. In this codelab, those components are often found using a resource string. You can access a composable by its resource string with the `Context.getString()` method, which you can read about here (https://developer.android.com/reference/android/content/Context#getString(int)). When writing a UI test in compose, implementing this method looks like this:

```
composeTestRule.onNodeWithText(composeTestRule.activity.getStr
ing(R.string.my_string)
```

This is a verbose instruction and it can be simplified with the addition of an extension function.

1. Create a new file in the `com.example.cupcake.test` package called `ComposeRuleExtensions.kt`. Make sure this is a plain Kotlin file.



2. Add the following code to that file.

```
import androidx.activity.ComponentActivity
import androidx.annotation.StringRes
import androidx.compose.ui.test.SemanticsNodeInteraction
import androidx.compose.ui.test.junit4.AndroidComposeTestRule
import androidx.compose.ui.test.onNodeWithText
import androidx.test.ext.junit.rules.ActivityScenarioRule

fun <A : ComponentActivity>
AndroidComposeTestRule<ActivityScenarioRule<A>,
A>.onNodeWithStringId(
    @StringRes id: Int
): SemanticsNodeInteraction =
onNodeWithText(activity.getString(id))
```

This extension function allows you to reduce the amount of code you write when finding a UI component by its string resource. Instead of writing this:

```
composeTestRule.onNodeWithText(composeTestRule.activity.getStr
ing(R.string.my_string)
```

You can now use the following instruction:

```
composeTestRule.onNodeWithStringId(R.string.my_string)
```

Verify that the Start screen doesn't have an Up button

The original design of the Cupcake app doesn't have an Up button in the toolbar of the Start screen.

The Start screen lacks a button because there is nowhere to navigate Up from this screen, since it is the initial screen. Follow these steps to create a function that confirms the Start screen doesn't have an Up button:

1. Create a method called `cupcakeNavHost_verifyBackNavigationNotShownOnStartOrderScreen()` and annotate it with `@Test`.

```
@Test
fun
cupcakeNavHost_verifyBackNavigationNotShownOnStartOrderScreen(
) {
}
```

In Cupcake, the Up button has a content description set to the string from the `R.string.back_button` resource.

2. Create a variable in the test function with the value of the `R.string.back_button` resource.

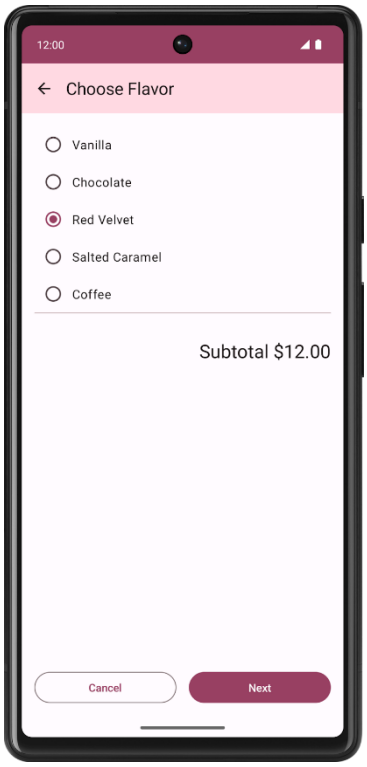
```
@Test
fun
cupcakeNavHost_verifyBackNavigationNotShownOnStartOrderScreen(
) {
    val backText =
composeTestRule.activity.getString(R.string.back_button)
}
```

3. Assert that a node with this content description does not exist on the screen.

```
@Test
fun
cupcakeNavHost_verifyBackNavigationNotShownOnStartOrderScreen(
) {
    val backText =
composeTestRule.activity.getString(R.string.back_button)
    composeTestRule.onNodeWithContentDescription(backText).ass
ertDoesNotExist()
}
```

Verify navigation to the Flavor screen

Clicking on one of the buttons in the Start screen triggers a method that instructs the nav controller to navigate to the Flavor screen.



In this test, you write a command to click a button to trigger this navigation and verify that the destination route is the Flavor screen.

1. Create a function called `cupcakeNavHost_clickOneCupcake_navigatesToSelectFlavorScreen()` and annotate it with `@Test`.

```
@Test
fun
cupcakeNavHost_clickOneCupcake_navigatesToSelectFlavorScreen()
{
}
```

Note: Test method names are different from the function names you use in app code. A good naming convention for test methods is the following: `thingUnderTest_TriggerOfTest_ResultOfTest`. Using the function you just created as an example, it is clear that you are testing the cupcake nav host by clicking the **One Cupcake** button, and the expected result is navigation to the Flavor screen.

2. Find the **One Cupcake** button by its string resource id and perform a click action on it.

```
import com.example.cupcake.R
...

@Test
fun
cupcakeNavHost_clickOneCupcake_navigatesToSelectFlavorScreen()
{
    composeTestRule.onNodeWithStringId(R.string.one_cupcake)
        .performClick()
}
```

3. Assert that the current route name is the Flavor screen name.

```
@Test
fun
cupcakeNavHost_clickOneCupcake_navigatesToSelectFlavorScreen()
{
    composeTestRule.onNodeWithStringId(R.string.one_cupcake)
        .performClick()
    navController.assertCurrentRouteName(CupcakeScreen.Flavor.
name)
}
```

Write more helper methods

The Cupcake app has a mostly linear navigation flow. Short of clicking the **Cancel** button, you can only navigate through the app in one direction. Therefore, as you test screens that are deeper within the app, you can find yourself repeating code to navigate to the areas you want to test. This situation merits the use of more helper methods so that you only have to write that code once.

Now that you tested navigation to the Flavor screen, create a method that navigates to the Flavor screen so that you don't have to repeat that code for future tests.

Note: Keep in mind that these are not test methods. They don't test anything and they should only be executed when explicitly called. Therefore, they should not be annotated with `@Test`.

1. Create a method called `navigateToFlavorScreen()`.

```
private fun navigateToFlavorScreen() {
}
```

2. Write a command to find the **One Cupcake** button and perform a click action on it, as you did in the previous section.

```
private fun navigateToFlavorScreen() {
    composeTestRule.onNodeWithStringId(R.string.one_cupcake)
        .performClick()
}
```

Recall that the **Next** button on the Flavor screen will not be clickable until a flavor is selected. This method is only meant to prepare the UI for navigation. After you call this method, the UI should be in a state in which the **Next** button is clickable.

3. Find a node in the UI with the `R.string.chocolate` string and perform a click action on it to select it.

```
private fun navigateToFlavorScreen() {
    composeTestRule.onNodeWithStringId(R.string.one_cupcake)
        .performClick()
    composeTestRule.onNodeWithStringId(R.string.chocolate)
        .performClick()
}
```

See if you can write helper methods that navigate to the Pickup screen and the Summary screen. Give this exercise a try on your own before looking at the solution.

Note: Navigating to the Summary screen first requires the selection of a date from the Pickup screen. You have to generate a date to select in the UI.

Use the following code to accomplish this:

```
private fun getFormattedDate(): String {
    val calendar = Calendar.getInstance()
    calendar.add(java.util.Calendar.DATE, 1)
    val formatter = SimpleDateFormat("E MMM d",
Locale.getDefault())
    return formatter.format(calendar.time)
}
```

```
private fun navigateToPickupScreen() {
    navigateToFlavorScreen()
    composeTestRule.onNodeWithStringId(R.string.next)
        .performClick()
}
```

```
private fun navigateToSummaryScreen() {
    navigateToPickupScreen()
    composeTestRule.onNodeWithText(getFormattedDate())
        .performClick()
    composeTestRule.onNodeWithStringId(R.string.next)
        .performClick()
}
```

As you test screens beyond the Start screen, you need to plan to test the Up button functionality to make sure it directs navigation to the previous screen. Consider making a helper function to find and click the Up button.

```
private fun performNavigateUp() {
    val backText =
composeTestRule.activity.getString(R.string.back_button)
    composeTestRule.onNodeWithContentDescription(backText).per
formClick()
}
```

Maximize test coverage

An app's test suite should test as much of the app functionality as possible. In a perfect world, a UI test suite would cover 100% of the UI functionality. In practice, this amount of test coverage is difficult to achieve because there are many factors external to your app that can affect the UI, such as devices with unique screen sizes, different versions of the Android operating system, and third party apps that can affect other apps on the phone.

One way to help maximize test coverage is to write tests alongside features as you add them. In doing so, you avoid getting too far ahead on new features and having to go back to remember all possible scenarios. Cupcake is a fairly small app at this point, and you already tested a significant portion of the app's navigation! However, there are more navigation states to test.

See if you can write the tests to verify the following navigation states. Try implementing them on your own before looking at the solution.

- Navigating to the Start screen by clicking the Up button from the Flavor screen
- Navigating to the Start screen by clicking the Cancel button from the Flavor screen
- Navigating to the Pickup screen
- Navigating to the Flavor screen by clicking the Up button from the Pickup screen
- Navigating to the Start screen by clicking the Cancel button from the Pickup screen

- Navigating to the Summary screen
- Navigating to the Start screen by clicking the Cancel button from the Summary screen

```
@Test
fun
cupcakeNavHost_clickNextOnFlavorScreen_navigatesToPickupScreen
() {
    navigateToFlavorScreen()
    composeTestRule.onNodeWithStringId(R.string.next)
        .performClick()
    navController.assertCurrentRouteName(CupcakeScreen.Pickup.
name)
}

@Test
fun
cupcakeNavHost_clickBackOnFlavorScreen_navigatesToStartOrderSc
reen() {
    navigateToFlavorScreen()
    performNavigateUp()
    navController.assertCurrentRouteName(CupcakeScreen.Start.n
ame)
}

@Test
fun
cupcakeNavHost_clickCancelOnFlavorScreen_navigatesToStartOrder
Screen() {
    navigateToFlavorScreen()
    composeTestRule.onNodeWithStringId(R.string.cancel)
        .performClick()
    navController.assertCurrentRouteName(CupcakeScreen.Start.n
ame)
}

@Test
fun
cupcakeNavHost_clickNextOnPickupScreen_navigatesToSummaryScree
n() {
    navigateToPickupScreen()
    composeTestRule.onNodeWithText(getFormattedDate())
        .performClick()
    composeTestRule.onNodeWithStringId(R.string.next)
        .performClick()
    navController.assertCurrentRouteName(CupcakeScreen.Summary
.name)
}
```

```
@Test
fun
cupcakeNavHost_clickBackOnPickupScreen_navigatesToFlavorScreen
() {
    navigateToPickupScreen()
    performNavigateUp()
    navController.assertCurrentRouteName(CupcakeScreen.Flavor.
name)
}

@Test
fun
cupcakeNavHost_clickCancelOnPickupScreen_navigatesToStartOrder
Screen() {
    navigateToPickupScreen()
    composeTestRule.onNodeWithStringId(R.string.cancel)
        .performClick()
    navController.assertCurrentRouteName(CupcakeScreen.Start.n
ame)
}

@Test
fun
cupcakeNavHost_clickCancelOnSummaryScreen_navigatesToStartOrde
rScreen() {
    navigateToSummaryScreen()
    composeTestRule.onNodeWithStringId(R.string.cancel)
        .performClick()
    navController.assertCurrentRouteName(CupcakeScreen.Start.n
ame)
}
```

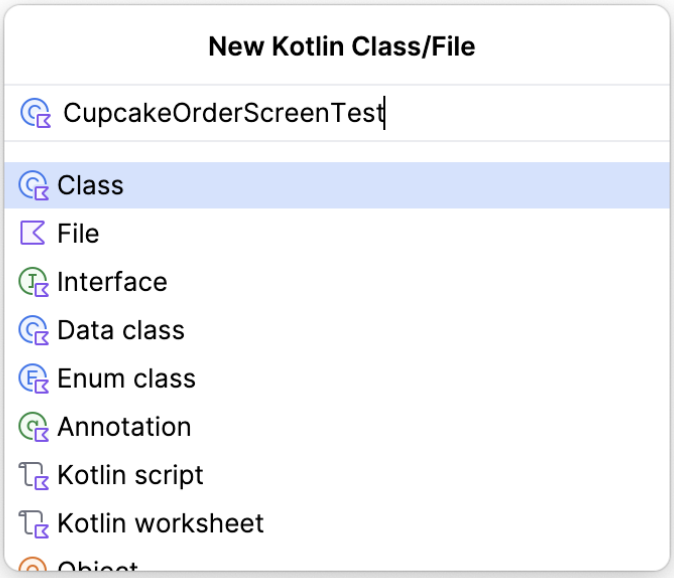
6. Write tests for the Order screen

Navigation is only one aspect of the Cupcake app's functionality. The user also interacts with each of the app screens. You need to verify what appears on these screens and that actions taken on these screens yield the correct results. The **SelectOptionScreen** is an important part of the app.

In this section, you write a test to verify that the content on this screen is correctly set.

Test the Choose Flavor screen content

1. Create a new class inside the `app/src/androidTest` directory called `CupcakeOrderScreenTest`, where your other test files are contained.



2. In this class, create an `AndroidComposeTestRule`.

```
@get:Rule
val composeTestRule =
createAndroidComposeRule<ComponentActivity>()
```

3. Create a function called `selectOptionScreen_verifyContent()` and annotate it with `@Test`.

```
@Test
fun selectOptionScreen_verifyContent() {

}
```

In this function, you ultimately set the Compose rule content to the `SelectOptionScreen`. Doing so ensures that the `SelectOptionScreen` composable launches directly so that navigation is not required. However, this screen requires two parameters: a list of flavor options and a subtotal.

4. Create a list of flavor options and a subtotal to be passed to the screen.

```
@Test
fun selectOptionScreen_verifyContent() {
    // Given list of options
    val flavors = listOf("Vanilla", "Chocolate", "Hazelnut",
"Cookie", "Mango")
    // And subtotal
    val subtotal = "$100"
}
```

5. Set the content to the `SelectOptionScreen` composable using the values you just created.

Note that this approach is similar to launching a composable from the `MainActivity`. The only difference is that the `MainActivity` calls the `CupcakeApp` composable, and here you are calling the `SelectOptionScreen` composable. Being able to change the composable you launch from `setContent()` lets you launch specific composables instead of having the test explicitly step through the app to get to the area you want to test. This approach helps prevent the test from failing in areas of the code that are unrelated to your current test.

```
@Test
fun selectOptionScreen_verifyContent() {
    // Given list of options
    val flavors = listOf("Vanilla", "Chocolate", "Hazelnut",
"Cookie", "Mango")
    // And subtotal
    val subtotal = "$100"

    // When SelectOptionScreen is loaded
    composeTestRule.setContent {
        SelectOptionScreen(subtotal = subtotal, options =
flavors)
    }
}
```

At this point in the test, the app launches the `SelectOptionScreen` composable and you are then able to interact with it through test instructions.

6. Iterate through the `flavors` list and ensure that each string item in the list is displayed on the screen.
7. Use the `onNodeWithText()` method to find the text on the screen and use the `assertIsDisplayed()` method to verify that the text is displayed in the app.

```
@Test
fun selectOptionScreen_verifyContent() {
    // Given list of options
    val flavors = listOf("Vanilla", "Chocolate", "Hazelnut",
"Cookie", "Mango")
    // And subtotal
    val subtotal = "$100"

    // When SelectOptionScreen is loaded
    composeTestRule.setContent {
        SelectOptionScreen(subtotal = subtotal, options =
flavors)
    }
```

```
    // Then all the options are displayed on the screen.
    flavors.forEach { flavor ->
        composeTestRule.onNodeWithText(flavor).assertIsDisplay
ed()
    }
}
```

8. Using the same technique to verify that the app displays the text, verify that the app displays the correct subtotal string on the screen. Search the screen for the `R.string.subtotal_price` resource id and the correct subtotal value, then assert that the app displays the value.

```
import com.example.cupcake.R
...

@Test
fun selectOptionScreen_verifyContent() {
    // Given list of options
    val flavors = listOf("Vanilla", "Chocolate", "Hazelnut",
"Cookie", "Mango")
    // And subtotal
    val subtotal = "$100"

    // When SelectOptionScreen is loaded
    composeTestRule.setContent {
        SelectOptionScreen(subtotal = subtotal, options =
flavors)
    }

    // Then all the options are displayed on the screen.
    flavors.forEach { flavor ->
        composeTestRule.onNodeWithText(flavor).assertIsDisplay
ed()
    }

    // And then the subtotal is displayed correctly.
    composeTestRule.onNodeWithText(
        composeTestRule.activity.getString(
            R.string.subtotal_price,
            subtotal
        )
    ).assertIsDisplayed()
}
```

Recall that the **Next** button is not enabled until an item is selected. This test only verifies the screen content, so the last thing to test is that the **Next** button is disabled.

9. Find the **Next** button by using the same approach to find a node by string resource id. However, instead of verifying that the app displays the node, use the `assertIsNotEnabled()` method.

```
@Test
fun selectOptionScreen_verifyContent() {
    // Given list of options
    val flavors = listOf("Vanilla", "Chocolate", "Hazelnut",
"Cookie", "Mango")
    // And subtotal
    val subtotal = "$100"

    // When SelectOptionScreen is loaded
    composeTestRule.setContent {
        SelectOptionScreen(subtotal = subtotal, options =
flavors)
    }

    // Then all the options are displayed on the screen.
    flavors.forEach { flavor ->
        composeTestRule.onNodeWithText(flavor).assertIsDisplay
ed()
    }

    // And then the subtotal is displayed correctly.
    composeTestRule.onNodeWithText(
        composeTestRule.activity.getString(
            R.string.subtotal_price,
            subtotal
        )
    ).assertIsDisplayed()

    // And then the next button is disabled
    composeTestRule.onNodeWithStringId(R.string.next).assertIs
NotEnabled()
}
```

Maximize test coverage

The Choose Flavor screen content test only tests one aspect of a single screen. There are a number of additional tests you can write to increase your code coverage. Try writing the following tests on your own before downloading the solution code.

- Verify the Start screen content.
- Verify the Summary screen content.
- Verify that the **Next** button is enabled when an option is selected on the Choose Flavor screen.

As you write your tests, keep in mind any helper functions that might reduce the amount of code you write along the way!

7. Get the solution code

To download the code for the finished codelab, you can use this git command:

```
$ git clone https://github.com/google-developer-
training/basic-android-kotlin-compose-training-cupcake.git
```

Alternatively, you can download the repository as a zip file, unzip it, and open it in Android Studio.

https://github.com/google-developer-training/basic-android-kotlin-compose-training-cupcake/archive/refs/heads/main.zip

Note: The solution code is in the main branch of the downloaded repository.

If you want to see the solution code, https://github.com/google-developer-training/basic-android-kotlin-compose-training-cupcake/tree/main

8. Summary

Congratulations! You've learned how to test the Jetpack Navigation component. You also learned some fundamental skills for writing UI tests, such as writing reusable helper methods, how to leverage `setContent()` to write concise tests, how to set up your tests with the `@Before` annotation, and how to think about maximum test coverage. As you continue to build Android apps, remember to keep writing tests alongside your feature code!