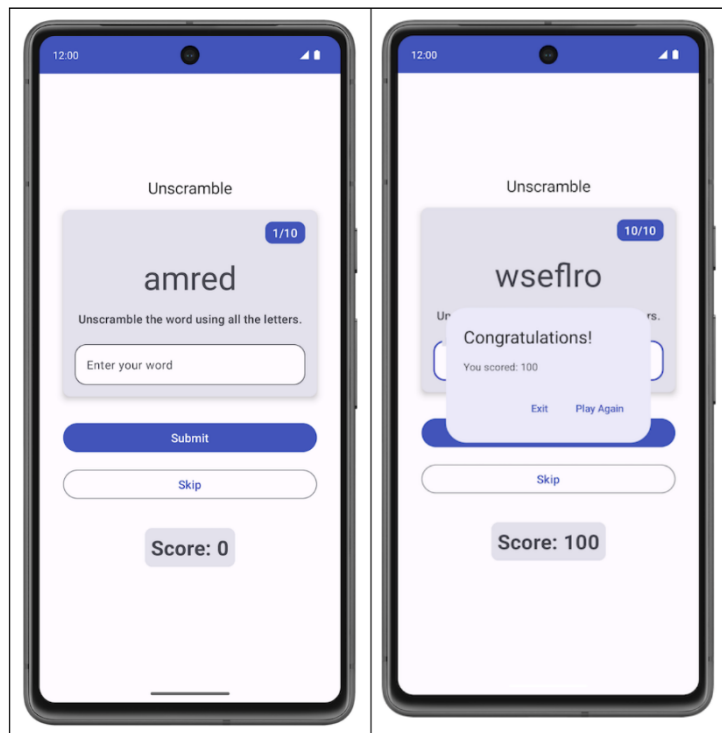


# Write unit tests for ViewModel

## 1. Before you begin

This codelab teaches you to write unit tests to test the `ViewModel` component. You will add unit tests for the Unscramble game app. The Unscramble app is a fun word game where users have to guess a scrambled word and earn points for guessing correctly. The following image shows a preview of the app:



In the Write automated tests codelab, you learned what automated tests are and why they are important. You also learned how to implement unit tests.

You learned:

- Automated testing is code that verifies the accuracy of another piece of code.
- Testing is an important part of the app development process. By running tests against your app consistently, you can verify your app's functional behavior and usability before you release it publicly.
- With unit tests, you can test functions, classes, and properties.
- Local unit tests are executed on your workstation, which means they run in a development environment without the need for an Android device or emulator. In other words, local tests run on your computer.

Before you proceed, make sure that you complete the Write automated tests and ViewModel and State in Compose codelabs.

## Prerequisites

- Knowledge of Kotlin, including functions, lambdas, and stateless composables
- Basic knowledge of how to build layouts in Jetpack Compose
- Basic knowledge of Material Design
- Basic knowledge of how to implement ViewModel

## What you'll learn

- How to add dependencies for unit tests in the app module's `build.gradle.kts` file
- How to create a test strategy to implement unit tests
- How to write unit tests using JUnit4 and understand the test instance lifecycle
- How to run, analyze, and improve code coverage

## What you'll build

- Unit tests for the Unscramble game app

## What you'll need

- The latest version of Android Studio

## Get the starter code

To get started, download the starter code:

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/archive/refs/heads/viewmodel.zip>

Alternatively, you can clone the GitHub repository for the code:

```
$ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git
$ cd basic-android-kotlin-compose-training-unscramble
$ git checkout viewmodel
```

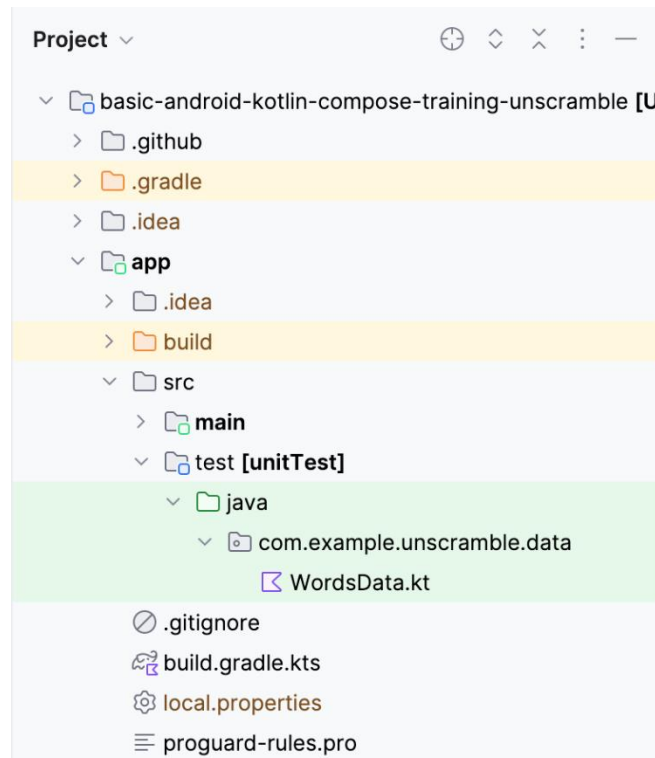
**Note:** The starter code is in the `viewmodel` branch of the downloaded repository.

You can browse the code in the Unscramble GitHub repository.

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble/tree/viewmodel>

## 2. Starter code overview

In Unit 2, you learned to place the unit test code in the **test** source set that is under the **src** folder, as shown in the following image:



The starter code has the following file:

- `WordsData.kt`: This file contains a list of words to use for testing and a `getUnscrambledWord()` helper function to get the unscrambled word from the scrambled word. You don't need to modify this file.

**Note:** The `allWords` property in the test source set `WordsData.kt` file will be used by the `GameViewModel`. When running tests, this replaces the `allWords` property available in the app source set `WordsData.kt` file.

In the next unit you will learn about a new technique, called Dependency Injection that promotes loose coupling and helps you to use different resources while running tests.

## 3. Add test dependencies

In this codelab, you use the JUnit framework to write unit tests. To use the framework, you need to add it as a dependency in your app module's `build.gradle.kts` file.

You use the `implementation` configuration to specify the dependencies that your app requires. For example, to use the `ViewModel` library in your application, you must add a dependency to `androidx.lifecycle:lifecycle-viewmodel-compose`, as shown in the following code snippet:

```
dependencies {  
  
    ...  
    implementation("androidx.lifecycle:lifecycle-viewmodel-  
compose:2.6.1")  
}
```

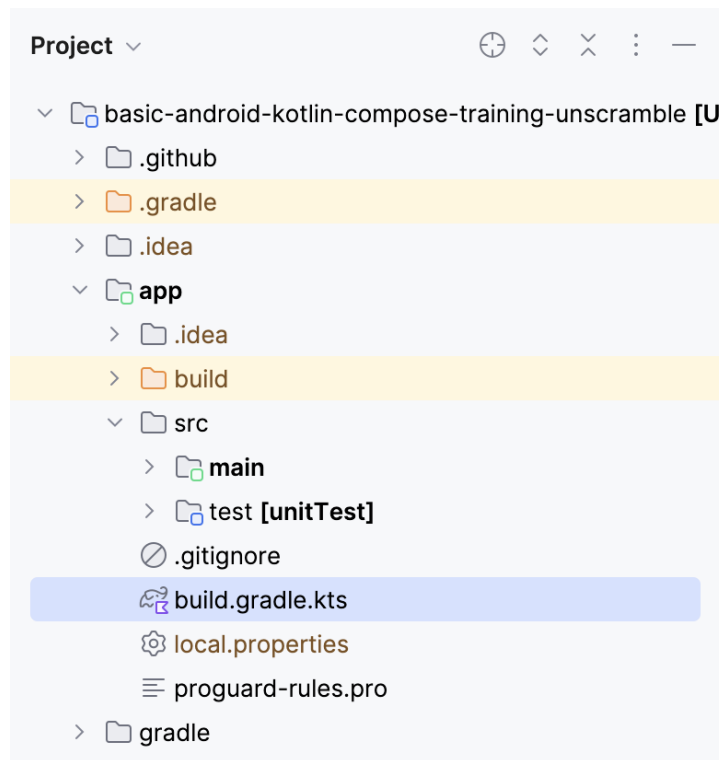
You can now use this library in your app's source code, and Android studio will help to add it to the generated Application Package File (APK) file. However, you do not want your unit test code to be part of your APK file. The test code doesn't add any functionality that the user would use, and the code also has an impact on the APK size. The same goes for the dependencies required by your test code; you should keep them separate. To do so, you use the `testImplementation` configuration, which indicates that the configuration applies to the local test source code and not the application code.

**Note:** Each Android application is compiled and packaged in a single file called an Application Package File (APK) that includes all of the application's code, resources, assets, and the manifest file. For convenience, an application package file is often referred to as an APK, and the file has the extension **.apk**. Android-powered devices use this file to install the app.

To add a dependency to your project, specify a dependency configuration (such as `implementation` or `testImplementation`) in the dependencies block of your `build.gradle.kts` file. Each dependency configuration provides Gradle with different instructions about how to use the dependency.

To add a dependency:

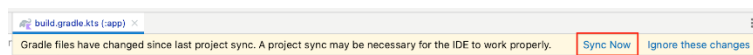
1. Open the app module's `build.gradle.kts` file, located in the `app` directory in the **Project** pane.



2. Inside the file, scroll down until you find the `dependencies{}` block. Add a dependency using `testImplementation` config for `junit`.

```
plugins {  
    ...  
}  
  
android {  
    ...  
}  
  
dependencies {  
    ...  
    testImplementation("junit:junit:4.13.2")  
}
```

3. In the notification bar at the top of the `build.gradle.kts` file, click **Sync Now** to let the import and build finish as shown in the following screenshot:



## Compose Bill of Materials (BOM)

Compose BOM is the recommended way to manage Compose library versions. BOM lets you manage all of your Compose library versions by specifying only the BOM's version.

Notice the dependency section in the `app module's build.gradle.kts` file.

```
// No need to copy over
// This is part of starter code
dependencies {

    // Import the Compose BOM
    implementation (platform("androidx.compose:compose-
bom:2023.06.01"))
    ...
    implementation("androidx.compose.material3:material3")
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.ui:ui-graphics")
    implementation("androidx.compose.ui:ui-tooling-preview")
    ...
}
```

Observe the following:

- Compose library version numbers are not specified.
- BOM is imported using `implementation`  
`platform("androidx.compose:compose-bom:2023.06.01")`

This is because the BOM itself has links to the latest stable versions of the different Compose libraries, in such a way that they work well together. When using the BOM in your app, you don't need to add any version to the Compose library dependencies themselves. When you update the BOM version, all the libraries that you're using are automatically updated to their new versions.

To use BOM with the compose testing libraries(instrumented tests) you need to import `androidTestImplementation`  
`platform("androidx.compose:compose-bom:xxxx.xx.xx")`. You could create a variable and reuse it for `implementation` and `androidTestImplementation` as shown.

```
// Example, not need to copy over
dependencies {

    // Import the Compose BOM
    implementation(platform("androidx.compose:compose-
bom:2023.06.01"))
    implementation("androidx.compose.material:material")
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.ui:ui-tooling-preview")

    // ...
    androidTestImplementation(platform("androidx.compose:compo
se-bom:2023.06.01"))
    androidTestImplementation("androidx.compose.ui:ui-test-
```

```
junit4")  
}
```

**Note:** Compose BOM is only for Compose libraries, not for other libraries such as lifecycle `androidx.lifecycle` library.

Great! You successfully added test dependencies to the app and learned about BOM. You are now ready to add some unit tests.

## 4. Test strategy

A good test strategy revolves around covering different paths and boundaries of your code. At a very basic level, you can categorize the tests in three scenarios: success path, error path, and boundary case.

- **Success path:** The success path tests - also known as happy path tests, focus on testing the functionality for a positive flow. A positive flow is a flow that has no exception or error conditions. Compared to the error path and boundary case scenarios, it is easy to create an exhaustive list of scenarios for success paths, since they focus on intended behavior for your app.

An example of a success path in the Unscramble app is the correct update of the score, word count, and the scrambled word when the user enters a correct word and clicks the **Submit** button.

- **Error path:** The error path tests focus on testing the functionality for a negative flow—that is, to check how the app responds to error conditions or invalid user input. It is quite challenging to determine all the possible error flows because there are lots of possible outcomes when intended behavior is not achieved.

One piece of general advice is to list all the possible error paths, write tests for them, and keep your unit tests evolving as you discover different scenarios.

An example of an error path in the Unscramble app is the user enters an incorrect word and clicks on the **Submit** button, which causes an error message to appear and the score and word count to not update.

- **Boundary case:** A boundary case focuses on testing boundary conditions in the app. In the Unscramble app, a boundary is checking the UI state when the app loads and the UI state after the user plays a maximum number of words.

Creating test scenarios around these categories can serve as guidelines for your test plan.

## Create tests

A good unit test typically has following four properties:

- **Focused:** It should focus on testing a unit, such as a piece of code. This piece of code is often a class or a method. The test should be narrow and focus on validating the correctness of individual pieces of code, rather than multiple pieces of code at the same time.
- **Understandable:** It should be simple and easy to understand when you read the code. At a glance, a developer should be able to immediately understand the intention behind the test.
- **Deterministic:** It should consistently pass or fail. When you run the tests any number of times, without making any code changes, the test should yield the same result. The test shouldn't be flaky, with a failure in one instance and a pass in another instance despite no modification to the code.
- **Self-contained:** It does not require any human interaction or setup and runs in isolation.

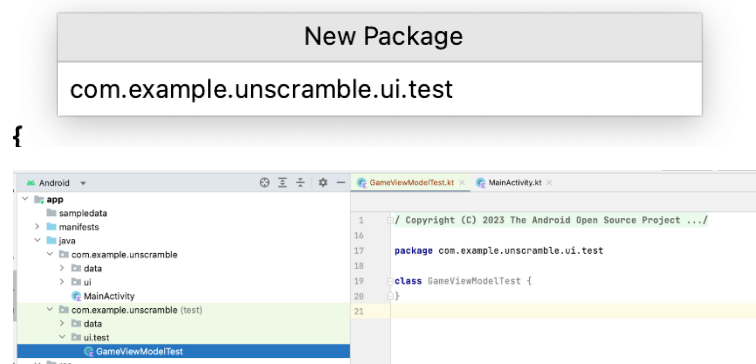
## Success path

To write a unit test for the success path, you need to assert that, given that an instance of the `GameViewModel` was initialized, when the `updateUserGuess()` method is called with correct guess word followed by a call to `checkUserGuess()` method, then:

- The correct guess is passed to the `updateUserGuess()` method.
- The `checkUserGuess()` method is called.
- The value for the `score` and `isGuessedWordWrong` status updates correctly.

Complete the following steps to create the test:

1. Create a new package `com.example.android.unscramble.ui.test` under test source set and add file as shown in the following screenshot:





To write a unit test for the `GameViewModel` class, you need an instance of the class so that you can call the class's methods and verify the state.

2. In the body of the `GameViewModelTest` class, declare a `viewModel` property and assign an instance of the `GameViewModel` class to it.

```
class GameViewModelTest {  
    private val viewModel = GameViewModel()  
}
```

3. To write an unit test for success path, create a `gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset()` function and annotate it with `@Test` annotation.

```
class GameViewModelTest {  
    private val viewModel = GameViewModel()  
  
    @Test  
    fun  
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset  
() {  
    }  
}
```

4. Import the following:

```
import org.junit.Test
```

**Note:** The code above uses the `thingUnderTest_TriggerOfTest_ResultOfTest` format to name the test function name:

- `thingUnderTest` = `gameViewModel`
- `TriggerOfTest` = `CorrectWordGuessed`
- `ResultOfTest` = `ScoreUpdatedAndErrorFlagUnset`

To pass a correct player word to the `viewModel.updateUserGuess()` method, you need to get the correct unscrambled word from the scrambled word in `GameUiState`. To do so, first get the current game ui state.

5. In the function body, create a `currentGameUiState` variable, and assign `viewModel.uiState.value` to it.

```
@Test  
fun  
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset  
() {
```

```
    var currentGameState = viewModel.uiState.value
}
```

**Warning:** This way to retrieve the `uiState` works because you used `MutableStateFlow`. In upcoming units, you will learn about advanced usages of `StateFlow` that create a stream of data, and you need to react to handle the stream. For those scenarios, you will write unit tests using different methods and approaches.

6. To get the correct player guess, use the `getUnscrambledWord()` function, which takes in the `currentGameState.currentScrambledWord` as an argument and returns the unscrambled word. Store this returned value in a new read-only variable named `correctPlayerWord` and assign the value returned by the `getUnscrambledWord()` function.

```
@Test
fun
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset
() {
    var currentGameState = viewModel.uiState.value
    val correctPlayerWord =
getUnscrambledWord(currentGameState.currentScrambledWord)
}
```

7. To verify if the guessed word is correct, add a call to the `viewModel.updateUserGuess()` method and pass the `correctPlayerWord` variable as an argument. Then add a call to `viewModel.checkUserGuess()` method to verify the guess.

```
@Test
fun
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset
() {
    var currentGameState = viewModel.uiState.value
    val correctPlayerWord =
getUnscrambledWord(currentGameState.currentScrambledWord)

    viewModel.updateUserGuess(correctPlayerWord)
    viewModel.checkUserGuess()
}
```

You are now ready to assert that the game state is what you expect.

8. Get the instance of the `GameState` class from the value of the `viewModel.uiState` property and store it in the `currentGameState` variable.

```

@Test
fun
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset
() {
    var currentGameUiState = viewModel.uiState.value
    val correctPlayerWord =
getUnscrambledWord(currentGameUiState.currentScrambledWord)
    viewModel.updateUserGuess(correctPlayerWord)
    viewModel.checkUserGuess()

    currentGameUiState = viewModel.uiState.value
}

```

9. To check the guessed word is correct and score is updated, use the `assertFalse()` function to verify that the `currentGameUiState.isGuessedWordWrong` property is false and the `assertEquals()` function to verify that the value of the `currentGameUiState.score` property is equal to 20.

```

@Test
fun
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset
() {
    var currentGameUiState = viewModel.uiState.value
    val correctPlayerWord =
getUnscrambledWord(currentGameUiState.currentScrambledWord)
    viewModel.updateUserGuess(correctPlayerWord)
    viewModel.checkUserGuess()

    currentGameUiState = viewModel.uiState.value
    // Assert that checkUserGuess() method updates
isGuessedWordWrong is updated correctly.
    assertFalse(currentGameUiState.isGuessedWordWrong)
    // Assert that score is updated correctly.
    assertEquals(20, currentGameUiState.score)
}

```

10. Import the following:

```

import org.junit.Assert.assertEquals
import org.junit.Assert.assertFalse

```

11. To make the value 20 readable and reusable, create a companion object and assign 20 to a `private` constant named `SCORE_AFTER_FIRST_CORRECT_ANSWER`. Update the test with the newly created constant.

```

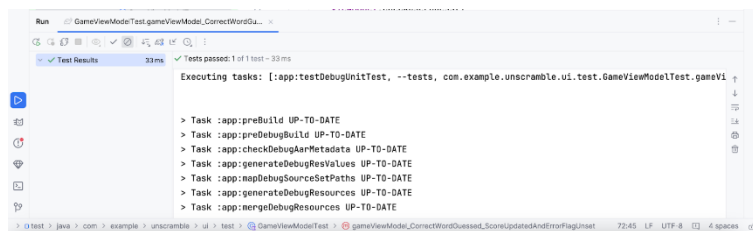
class GameViewModelTest {
    ...
    @Test
    fun
gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset
() {
    ...
    // Assert that score is updated correctly.
    assertEquals(SCORE_AFTER_FIRST_CORRECT_ANSWER,
currentGameUiState.score)
    }

    companion object {
        private const val SCORE_AFTER_FIRST_CORRECT_ANSWER =
SCORE_INCREASE
    }
}

```

12. Run the test.

The test should pass, since all the assertions were valid, as shown in the following screenshot:



## Error path

To write a unit test for the error path, you need to assert that when an incorrect word is passed as an argument to the `viewModel.updateUserGuess()` method and the `viewModel.checkUserGuess()` method is called, then the following happens:

- The value of the `currentGameUiState.score` property remains unchanged.
- The value of the `currentGameUiState.isGuessedWordWrong` property is set to true because the guess is wrong.

Complete the following steps to create the test:

1. In the body of the `GameViewModelTest` class, create a `gameViewModel_IncorrectGuess_ErrorFlagSet()` function and annotate it with the `@Test` annotation.

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

2. Define an `incorrectPlayerWord` variable and assign the "and" value to it, which should not exist in the list of words.

```
@Test
fun gameViewModel_IncorrectGuess_ErrorFlagSet() {
    // Given an incorrect word as input
    val incorrectPlayerWord = "and"
}
```

3. Add a call to the `viewModel.updateUserGuess()` method and pass the `incorrectPlayerWord` variable as an argument.
4. Add a call to the `viewModel.checkUserGuess()` method to verify the guess.

```
@Test
fun gameViewModel_IncorrectGuess_ErrorFlagSet() {
    // Given an incorrect word as input
    val incorrectPlayerWord = "and"

    viewModel.updateUserGuess(incorrectPlayerWord)
    viewModel.checkUserGuess()
}
```

5. Add a `currentGameUiState` variable and assign the value of the `viewModel.uiState.value` state to it.
6. Use assertion functions to assert that the value of the `currentGameUiState.score` property is 0 and the value of the `currentGameUiState.isGuessedWordWrong` property is set to true.

```
@Test
fun gameViewModel_IncorrectGuess_ErrorFlagSet() {
    // Given an incorrect word as input
    val incorrectPlayerWord = "and"

    viewModel.updateUserGuess(incorrectPlayerWord)
    viewModel.checkUserGuess()

    val currentGameUiState = viewModel.uiState.value
```

```
// Assert that score is unchanged
assertEquals(0, currentGameState.score)
// Assert that checkUserGuess() method updates
isGuessedWordWrong correctly
assertTrue(currentGameState.isGuessedWordWrong)
}
```

7. Import the following:

```
import org.junit.Assert.assertTrue
```

8. Run the test to confirm that it passes.

## Boundary case

To test the initial state of the UI, you need to write a unit test for the `GameViewModel` class. The test must assert that when the `GameViewModel` is initialized, then the following is true:

- `currentWordCount` property is set to 1.
- `score` property is set to 0.
- `isGuessedWordWrong` property is set to false.
- `isGameOver` property is set to false.

Complete the following steps to add the test:

1. Create a `gameViewModel_Initialization_FirstWordLoaded()` method and annotate it with the `@Test` annotation.

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

2. Access `viewModel.uiState.value` property to get the initial instance of the `GameState` class. Assign it to a new `gameUiState` read-only variable.

```
@Test
fun gameViewModel_Initialization_FirstWordLoaded() {
    val gameUiState = viewModel.uiState.value
}
```

3. To get the correct player word, use the `getUnscrambledWord()` function, which takes in the `gameUiState.currentScrambledWord` word and

returns the unscrambled word. Assign the returned value to a new read-only variable named `unScrambledWord`.

```
@Test
fun gameViewModel_Initialization_FirstWordLoaded() {
    val gameUiState = viewModel.uiState.value
    val unScrambledWord =
        getUnscrambledWord(gameUiState.currentScrambledWord)
}
```

4. To verify the state is correct, add the `assertTrue()` functions to assert that the `currentWordCount` property is set to 1, and the score property is set to 0.
5. Add `assertFalse()` functions to verify that the `isGuessedWordWrong` property is false and that the `isGameOver` property is set to false.

```
@Test
fun gameViewModel_Initialization_FirstWordLoaded() {
    val gameUiState = viewModel.uiState.value
    val unScrambledWord =
        getUnscrambledWord(gameUiState.currentScrambledWord)

    // Assert that current word is scrambled.
    assertNotEquals(unScrambledWord,
        gameUiState.currentScrambledWord)
    // Assert that current word count is set to 1.
    assertTrue(gameUiState.currentWordCount == 1)
    // Assert that initially the score is 0.
    assertTrue(gameUiState.score == 0)
    // Assert that the wrong word guessed is false.
    assertFalse(gameUiState.isGuessedWordWrong)
    // Assert that game is not over.
    assertFalse(gameUiState.isGameOver)
}
```

6. Import the following:

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

7. Run the test to confirm that it passes.

Another boundary case is to test the UI state after the user guesses all the words. You need to assert that when the user guesses all the words correctly, then the following is true:

- The score is up-to-date;

- The `currentGameState.currentWordCount` property is equal to the value of the `MAX_NO_OF_WORDS` constant.
- The `currentGameState.isGameOver` property is set to `true`.

Complete the following steps to add the test:

1. Create a `gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly()` method and annotate it with the `@Test` annotation. In the method, create an `expectedScore` variable and assign 0 to it.

```
@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
}
```

2. To get the initial state, add a `currentGameState` variable, and assign the value of the `viewModel.uiState.value` property to the variable.

```
@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
    var currentGameState = viewModel.uiState.value
}
```

3. To get the correct player word, use the `getUnscrambledWord()` function, which takes in the `currentGameState.currentScrambledWord` word and returns the unscrambled word. Store this returned value in a new read-only variable named `correctPlayerWord` and assign the value returned by the `getUnscrambledWord()` function.

```
@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
    var currentGameState = viewModel.uiState.value
    var correctPlayerWord =
        getUnscrambledWord(currentGameState.currentScrambledWord)
}
```

4. To test if the user guesses all the answers, use a `repeat` block to repeat the execution of the `viewModel.updateUserGuess()` method and the `viewModel.checkUserGuess()` method `MAX_NO_OF_WORDS` times.

```
@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
    var currentGameState = viewModel.uiState.value
```



```

        var correctPlayerWord =
        getUnscrambledWord(currentGameState.currentScrambledWord)
        repeat(MAX_NO_OF_WORDS) {

        }
    }
}

```

5. In the `repeat` block, add the value of the `SCORE_INCREASE` constant to the `expectedScore` variable to assert that the score increases after each correct answer.
6. Add a call to the `viewModel.updateUserGuess()` method and pass the `correctPlayerWord` variable as an argument.
7. Add a call to the `viewModel.checkUserGuess()` method to trigger the check for the user guess.

```

@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
    var currentGameState = viewModel.uiState.value
    var correctPlayerWord =
    getUnscrambledWord(currentGameState.currentScrambledWord)
    repeat(MAX_NO_OF_WORDS) {
        expectedScore += SCORE_INCREASE
        viewModel.updateUserGuess(correctPlayerWord)
        viewModel.checkUserGuess()
    }
}

```

8. Update the current player word, use the `getUnscrambledWord()` function, which takes in the `currentGameState.currentScrambledWord` as an argument and returns the unscrambled word. Store this returned value in a new read-only variable named `correctPlayerWord`. To verify the state is correct, add the `assertEquals()` function to check if the value of the `currentGameState.score` property is equal to the value of the `expectedScore` variable.

```

@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
    var currentGameState = viewModel.uiState.value
    var correctPlayerWord =
    getUnscrambledWord(currentGameState.currentScrambledWord)
    repeat(MAX_NO_OF_WORDS) {
        expectedScore += SCORE_INCREASE
        viewModel.updateUserGuess(correctPlayerWord)
        viewModel.checkUserGuess()
        currentGameState = viewModel.uiState.value
        correctPlayerWord =
    }
}

```

```

getUnscrambledWord(currentGameState.currentScrambledWord)
    // Assert that after each correct answer, score is
    updated correctly.
    assertEquals(expectedScore, currentGameState.score)
}
}

```

9. Add an `assertEquals()` function to assert that the value of the `currentGameState.currentWordCount` property is equal to the value of the `MAX_NO_OF_WORDS` constant and that the value of the `currentGameState.isGameOver` property is set to `true`.

```

@Test
fun gameViewModel_AllWordsGuessed_UiStateUpdatedCorrectly() {
    var expectedScore = 0
    var currentGameState = viewModel.uiState.value
    var correctPlayerWord =
getUnscrambledWord(currentGameState.currentScrambledWord)
    repeat(MAX_NO_OF_WORDS) {
        expectedScore += SCORE_INCREASE
        viewModel.updateUserGuess(correctPlayerWord)
        viewModel.checkUserGuess()
        currentGameState = viewModel.uiState.value
        correctPlayerWord =
getUnscrambledWord(currentGameState.currentScrambledWord)
        // Assert that after each correct answer, score is
        updated correctly.
        assertEquals(expectedScore, currentGameState.score)
    }
    // Assert that after all questions are answered, the
    current word count is up-to-date.
    assertEquals(MAX_NO_OF_WORDS,
currentGameState.currentWordCount)
    // Assert that after 10 questions are answered, the game
    is over.
    assertTrue(currentGameState.isGameOver)
}

```

10. Import the following:

```
import com.example.unscramble.data.MAX_NO_OF_WORDS
```

11. Run the test to confirm that it passes.

## Test instance lifecycle overview

When you take a close look at the way `viewModel` initializes in the test, you might notice that the `viewModel` initializes only once even though all the tests use it. This code snippet shows the definition of the `viewModel` property.

```
class GameViewModelTest {  
    private val viewModel = GameViewModel()  
  
    @Test  
    fun gameViewModel_Initialization_FirstWordLoaded() {  
        val gameUiState = viewModel.uiState.value  
        ...  
    }  
    ...  
}
```

You may wonder the following questions:

- Does it mean that the same instance of `viewModel` is reused for all the tests?
- Will it cause any issues? For example, what if the `gameViewModel_Initialization_FirstWordLoaded` test method executes after the `gameViewModel_CorrectWordGuessed_ScoreUpdatedAndErrorFlagUnset` test method? Will the initialization test fail?

The answer to both questions is no. Test methods are executed in isolation to avoid unexpected side effects from mutable test instance state. By default, before each test method is executed, JUnit creates a new instance of the test class.

Since you have four test methods so far in your `GameViewModelTest` class, the `GameViewModelTest` instantiates four times. Each instance has its own copy of the `viewModel` property. Hence, the sequence of test execution doesn't matter.

**Note:** This "per-method" test instance lifecycle is the default behavior since JUnit4.

## 5. Introduction to code coverage

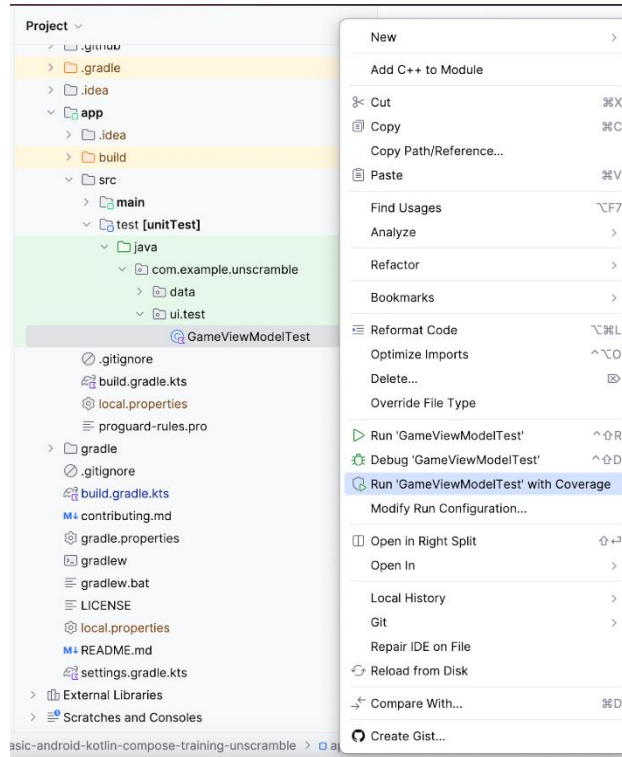
Code coverage plays a vital role to determine if you adequately test the classes, methods, and lines of code that make up your app.

Android Studio provides a test coverage tool for local unit tests to track the percentage and areas of your app code that your unit tests covered.

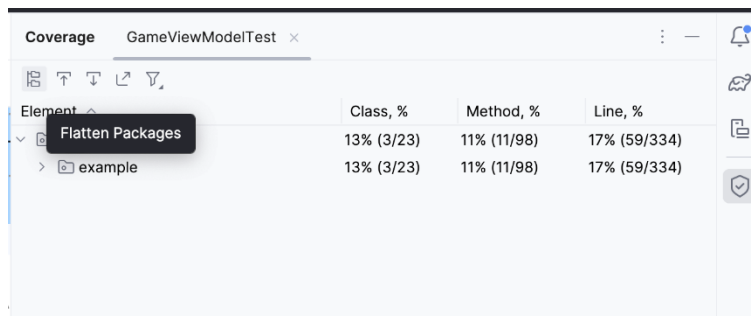
## Run test with coverage using Android Studio

To run tests with coverage:

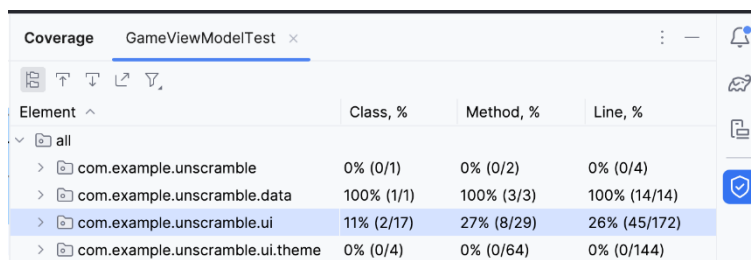
1. Right-click the `GameViewModelTest.kt` file in the project pane and select  **Run 'GameViewModelTest' with Coverage**.



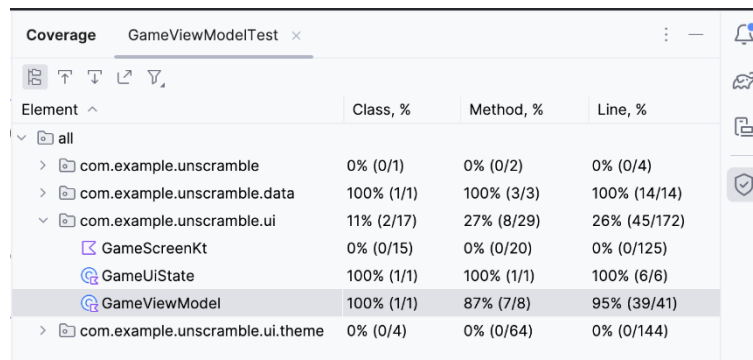
2. After the test execution completes, in the coverage panel on the right, click the **Flatten Packages** option.



3. Notice the `com.example.android.unsramble.ui` package as shown in the following image.



- Double click on the package `com.example.android.unscramble.ui` name, displays the coverage for `GameViewModel` as shown in the following image:



| Element                         | Class, %          | Method, %        | Line, %            |
|---------------------------------|-------------------|------------------|--------------------|
| all                             |                   |                  |                    |
| com.example.unscramble          | 0% (0/1)          | 0% (0/2)         | 0% (0/4)           |
| com.example.unscramble.data     | 100% (1/1)        | 100% (3/3)       | 100% (14/14)       |
| com.example.unscramble.ui       | 11% (2/17)        | 27% (8/29)       | 26% (45/172)       |
| GameScreenKt                    | 0% (0/15)         | 0% (0/20)        | 0% (0/125)         |
| GameUiState                     | 100% (1/1)        | 100% (1/1)       | 100% (6/6)         |
| <b>GameViewModel</b>            | <b>100% (1/1)</b> | <b>87% (7/8)</b> | <b>95% (39/41)</b> |
| com.example.unscramble.ui.theme | 0% (0/4)          | 0% (0/64)        | 0% (0/144)         |

## Analyze test report

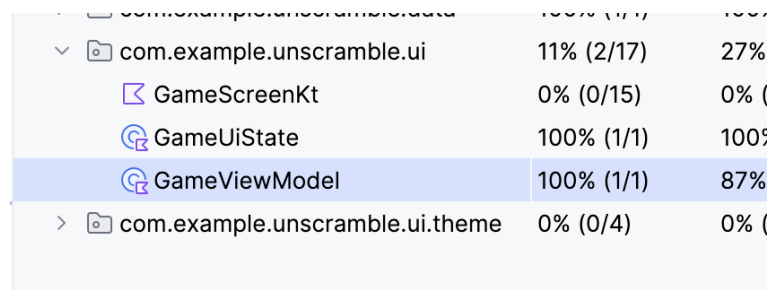
The report shown in the following diagram is broken down into two aspects:

- The percentage of methods covered by the unit tests:** In the example diagram, the tests you wrote, so far, covered 7 out of 8 methods. That is 87% of the total methods.
- The percentage of lines covered by the unit tests:** In the example diagram, the tests you wrote covered 39 out of 41 lines of code. That is 95% of the lines of code.

**Note:** Code coverage tracks the number of lines executed during test execution. Hence, even though there are no verifications on `GameUiState`, the coverage shows 100%.

The reports suggest that the unit tests you wrote so far missed certain parts of the code. To determine which parts were missed, complete the following step:

- Double-click `GameViewModel`.



|                                 |                   |            |
|---------------------------------|-------------------|------------|
| com.example.unscramble.ui       | 11% (2/17)        | 27%        |
| GameScreenKt                    | 0% (0/15)         | 0%         |
| GameUiState                     | 100% (1/1)        | 100%       |
| <b>GameViewModel</b>            | <b>100% (1/1)</b> | <b>87%</b> |
| com.example.unscramble.ui.theme | 0% (0/4)          | 0%         |

Android Studio displays the `GameViewModel.kt` file with additional color coding on the left side of the window. The bright green color indicates that those lines of code were covered.

```

16
17 package com.example.unscramble.ui
18
19 > import ...
20
21 /**
22  * ViewModel containing the app data and methods to process the data
23  */
24 @Android Dev
25 class GameViewModel : ViewModel() {
26
27     // Game UI state
28     private val _uiState = MutableStateFlow(GameUiState())
29     val uiState: StateFlow<GameUiState> = _uiState.asStateFlow()
30
31     @Android Dev
32     var userGuess by mutableStateOf<String>{ "" }
33     private set
34
35     // Set of words used in the game
36     private var usedWords: MutableSet<String> = mutableSetOf()
37     private lateinit var currentWord: String
38
39     @Android Dev
40     init {
41         resetGame()
42     }
43
44
45
46
47
48
49

```

When you scroll down in the GameViewModel, you might notice that a couple of lines are marked with a light pink color. This color indicates that these lines of code were not covered by the unit tests.

```

86
87     /**
88      * Skip to next word
89      */
90     fun skipWord() {
91         updateGameState(_uiState.value.score)
92         // Reset user guess
93         updateUserGuess( guessedWord: "" )
94     }
95
96

```

## Improve coverage

To improve the coverage, you need to write a test that covers the missing path. You need to add a test to assert that when a user skips a word, then the following is true:

- `currentGameUiState.score` property remains unchanged.
- `currentGameUiState.currentWordCount` property is incremented by one, as shown in the following code snippet.

To prepare to improve coverage, add the following test method to the GameViewModelTest class.

```

@Test
fun
gameViewModel_WordSkipped_ScoreUnchangedAndWordCountIncreased(

```

```

) {
    var currentGameUiState = viewModel.uiState.value
    val correctPlayerWord =
getUnscrambledWord(currentGameUiState.currentScrambledWord)
    viewModel.updateUserGuess(correctPlayerWord)
    viewModel.checkUserGuess()

    currentGameUiState = viewModel.uiState.value
    val lastWordCount = currentGameUiState.currentWordCount
    viewModel.skipWord()
    currentGameUiState = viewModel.uiState.value
    // Assert that score remains unchanged after word is
skipped.
    assertEquals(SCORE_AFTER_FIRST_CORRECT_ANSWER,
currentGameUiState.score)
    // Assert that word count is increased by 1 after word is
skipped.
    assertEquals(lastWordCount + 1,
currentGameUiState.currentWordCount)
}

```

Complete the following steps to re-run the coverage:

1. Right-click the `GameViewModelTest.kt` file and from the menu and select **Run 'GameViewModelTest' with Coverage**.
2. After the build is successful, navigate to the **GameViewModel** element again and confirm that the coverage percentage is 100%. The final coverage report is shown in the following image.

|                           |            |            |              |
|---------------------------|------------|------------|--------------|
| com.example.unscramble.ui | 11% (2/17) | 31% (9/29) | 27% (47/172) |
| GameScreenKt              | 0% (0/15)  | 0% (0/20)  | 0% (0/125)   |
| GameUiState               | 100% (1/1) | 100% (1/1) | 100% (6/6)   |
| GameViewModel             | 100% (1/1) | 100% (8/8) | 100% (41/41) |

1. Navigate to the `GameViewModel.kt` file and scroll down to check whether the previously missed path is now covered.

```

86      /*
87      * Skip to next word
88      */
89      fun skipWord() {
90          updateGameState(_uiState.value.score)
91          // Reset user guess
92          updateUserGuess( guessedWord: "")
93      }

```

You learned how to run, analyze, and improve the code coverage of your application code.

*Does a high code coverage percentage mean high quality of app code?* No. Code coverage indicates the percentage of code covered, or executed, by your unit test. It

doesn't indicate that the code is verified. If you remove all the assertions from your unit test code and run the code coverage, it still shows 100% coverage.

A high coverage doesn't indicate that the tests are designed correctly and that the tests verify the app's behavior. You need to ensure that the tests you wrote have the assertions that verify the behavior of the class being tested. You also don't have to strive to write unit tests to get a 100% test coverage for the entire app. You should test some parts of the app's code, such as Activities, using UI tests instead.

However, a low coverage means that large parts of your code were completely untested. Use the code coverage as a tool to find the parts of code that were not executed by your tests, rather than a tool to measure your code's quality.

## 6. Get the solution code

To download the code for the finished codelab, you can use these git commands:

```
$ git clone https://github.com/google-developer-  
training/basic-android-kotlin-compose-training-unsramble.git  
$ cd basic-android-kotlin-compose-training-unsramble  
$ git checkout main
```

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-unsramble/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, view it on GitHub.

<https://github.com/google-developer-training/basic-android-kotlin-compose-training-unsramble>

## 7. Conclusion

Congratulations! You learned how to define test strategy and implemented unit tests to test the `ViewModel` and `StateFlow` in the Unscramble app. As you continue to build Android apps, make sure that you write tests alongside your app features to confirm that your apps work properly throughout the development process.



## Summary

- Use the `testImplementation` configuration to indicate that the dependencies apply to the local test source code and not the application code.
- Aim to categorize tests in three scenarios: Success path, error path, and boundary case.
- A good unit test has at least four characteristics: they are focused, understandable, deterministic, and self-contained.
- Test methods are executed in isolation to avoid unexpected side effects from mutable test instance state.
- By default, before each test method executes, JUnit creates a new instance of the test class.
- Code coverage plays a vital role to determine whether you adequately tested the classes, methods, and lines of code that make up your app.

## Learn more

|                                      |   |
|--------------------------------------|---|
| Fundamentals of testing Android apps | <a href="https://developer.android.com/training/testing/fundamentals#benefits">https://developer.android.com/training/testing/fundamentals#benefits</a> |
| Using the Bill of Materials          | <a href="https://developer.android.com/jetpack/compose/bom/bom">https://developer.android.com/jetpack/compose/bom/bom</a>                               |