Up to date for Kotlin 1.3

Android Test-Driven Development by Tutorials

FIRST EDITION

Learn Android TDD by Building Real-World Apps

By the raywenderlich Tutorial Team

Lance Gleason, Fernando Sproviero & Victoria Gonda

Table of Contents: Overview

Table of Contents: Extended

Writing apps is hard. Writing testable apps is even harder, but it doesn't have to be. Reading and understanding all the official Google documentation on testing can be time-consuming — and confusing. This is where *Android Test-Driven Development* comes to the rescue! In this book, you'll learn about Android Test-Driven Development the quick and easy way: by following fun and easy-to-read tutorials.

This book is for the intermediate Android developers who already know the basics of Android and Kotlin development but want to learn Android Test-Driven Development.

We are pleased to offer you this sample chapter from the full *Android Test-Driven Development by Tutorials*, "What Is a Test?"

We hope this sample chapter will introduce you to these concepts and give you a chance to practice them in our hands-on By Tutorials style.

The full book is ready for purchase at:

- https://store.raywenderlich.com/products/android-test-driven-development-by-tutorials.

Enjoy!

The *Android Test-Driven Development by Tutorials* Team

Android Test-Driven Development by Tutorials

By Lance Gleason, Victoria Gonda and Fernando Sproviero

Notice of Rights

Notice of Liability

Trademarks

Dedications

"There are many people who helped to make this book possible. My other half Marlene was the one who initially suggested that I try out for the Ray Wenderlich team. She gave me lots of encouragement and, even before the editors saw my work, edited every chapter to make sure my sentences were coherent. My late mother passed on her love of reading and many creative skills for which I will always be grateful. The many strong women and family members in my life who taught me to live life with honesty and conviction and were encouraging of my work. I owe a debt of gratitude to the Ruby community for teaching me about TDD and infecting me with enthusiasm. I'd also like to thank Ray Wenderlich and the team for giving me the chance to share my love of TDD with the world. Finally, I'd like to thank all of the editors and co-authors of this book. It has been a very rewarding experience working with everybody on the team."

— Lance Gleason

"To my family, friends, and especially my partner, who supported me while writing this book. Tyler, thanks for all your encouragement and patience you gave me as I spent evenings in front of my laptop turning thoughts into words."

— Victoria Gonda

"To my girlfriend Romina, who will soon become my wife. Thanks for your support and help, examples and discussions about the topics of this book. Yes, she's a developer too, isn't that great? :] Also, to my family, who had to listen repeatedly about what I was writing, and no, they aren't developers!"

— Fernando Sproviero

What You Need

To follow along with this book, you'll need the following:

- **Kotlin 1.3:** This book uses Kotlin 1.3 throughout. The examples may work with an earlier version of Kotlin, but they are untested.
- **Android Studio 3.5 or later.** Android Studio is the main development tool for Android. You'll need Android Studio 3.5 or later for the tasks in this book. You can download the latest version of Android Studio from Android's developer site here: https://developer.android.com/studio.

If you haven't installed the latest version of Android Studio, be sure to do that before continuing with the book. The code covered in this book depends on Android 10, Kotlin 1.3 and Android Studio 3.5 — you may get lost if you try to work with an older version.

Book License

By purchasing *Android Test-Driven Development by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Android Test-Driven Development by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Android Test-Driven Development by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Android Test-Driven Development by Tutorials*, available at www.raywenderlich.com."
- The source code included in *Android Test-Driven Development by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Android Test-Driven Development by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement.

In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Chapter 2: What Is a Test?

By Fernando Sproviero

A **test** is a manual or automatic procedure used to evaluate if the **System Under Test** (SUT) behaves correctly.

The SUT may be a method, an entire class, a module or even a whole application.

From now on, when mentioning anything related to writing a test this book will be referring to the **automatic** procedure form of a test.

To write a test, you need to understand the feature, specification or requirement of the component you are implementing. That component may be an **Activity**, **Fragment**, **View Model** or several of these components working together. These may come in different forms, such as user stories, use cases or some other kind of documentation.

Testing is an important part of software development. By including tests along with your code, you can ensure that your code works and that later changes to the code won't break it. Tests can give you the peace of mind you need to develop quickly and catch bugs before they're released.

Essentially, there are two approaches to writing tests:

- Write tests **before** you write the feature.
- Write tests **after** you write the feature.

This book primarily focuses on writing tests first versus writing them after a feature has been implemented.

Why should you test?

Writing tests can take more time up front, and it is code you write that the client won't "see", which is why tests are sometimes skipped by developers. However, having tests can speed up development down the road and it presents some advantages.

Change/refactor confidence

You have probably run into a scenario in which you have a section of your application that works correctly before adding new functionality to the application. After adding new functionality, either in **Quality Assurance (QA)** or after it is released to customers you discover that this new functionality broke the previously working section. That is called a **regression**.

Having good, reliable, effective tests would have caught that at the moment the bug was introduced saving time in QA and preventing preventable bugs from making it to your users. Another related scenario is where you have a section of your application that is working correctly, but could use some refactoring to use a new library, break things up to follow a more readable architectural pattern, etc. A good test suite will provide you with the confidence to make those changes without having to do time consuming manual QA regression test cycles to ensure everything is still working correctly.

However, you should always bear in mind that this is not a 100% "insurance". No matter how many tests you write, there could be edge cases that the tests don't catch. Even so, it's absolutely safer to have tests that catch *most* issues than not having them at all!

Usually, you will write tests for the most common scenarios your user may encounter. Whenever someone finds a bug that your tests didn't catch, you should immediately add a test for it.

Documentation

Some companies and developers treat tests as a complementary documentation to explain how the implementation of a feature works. When you have well-written tests, they provide an excellent description of what your code should do. By writing a test, its corresponding implementation and repeating this until a feature is completed, bearing in mind that these tests can be treated as specifications, will help you and your team when a refactor or a modification of the feature is required.

When you're working on a piece of code, you can look at the tests to help you understand what the code does. You can also see what the code *should not* do. Because these are tests rather than a static document, as long as the tests are passing you can be sure this form of documentation is up-to-date!

How to write a test

There are many things to bear in mind when writing a test. You'll understand them by reading this book and practicing writing tests. However, the most important aspects of writing a test are as follows:

- **Naming**: You should give a meaningful name to each test so that it is clearly identifiable in code and in subsequent reports.

For example, consider a quiz game:

```
fun whenAnsweringCorrectly_shouldIncrementCurrentScore() {
  ...
}
```

This test's name represents the state of what you are testing and the expected behavior. This is useful when looking at the report after running a test suite.

- **Short and simple**: You should aim to write tests that focus on a narrow piece of functionality. As a rule of thumb, if your test methods get long, and have multiple **assertion** statements to check conditions of the system, it may be trying to test too many things. In that scenario it may be a good idea to break up that test into multiple, more narrowly focused tests. Take a look at this test:

```
fun whenIncrementingScore_shouldIncrementCurrentScore() {
  val score = Score(0)

  score.increment()

  if (score.current == 1) {
    print("Success")
  } else {
    throw AssertionError("Invalid score")
  }
}
```

The test only has seven lines of code to bring the SUT in the desired state and check the expected behavior.

- **Check one single thing**: Check one thing at a time. If you need to test multiple things, write an additional test similar to the one you've just previously run, but change the **check**:

```
fun
whenIncrementingScore_aboveHighScore_shouldAlsoIncrementHighScor
e() {
  val score = Score(0)

  score.increment()

  if (score.highest == 1) {
    print("Success")
  } else {
    throw AssertionError("Invalid high score")
  }
}
```

As you can see, this test is very similar to the previous one; however, the **check** is different. In the first test, you checked that the score of the quiz game incremented correctly. Now, you check that the *highest score* also increments along with the score.

- **Readable**: Anyone in the team should be able to read and understand what is going on in your test and/or what is the purpose of the test. Consequently, you should pay attention to the naming of each variable or method used and the logic sequence of the test. If you don't, then the tests will become difficult to maintain and keep up-to-date.

What should you test?

You should test code that is related to the logic of your app. This may include code that you have to write to:

- Show the UI and navigate between the screens of your app.
- Make network requests to an API.
- Persist data.
- Interact with device sensors.
- Model your domain.

Having tests for the logic of your application should be your main goal, however, bear also in mind the following:

Code that breaks often

If you have a legacy project without tests, and it breaks often whenever you modify its code, it's useful to have tests for them, so that the next time you make a modification you will be sure that it won't keep breaking.

Code that will change

If you know that some code will be refactored in the near future, tests will be useful here, too, because if you wrote tests for this feature, you can support on them to refactor the code and be sure you don't break anything.

What should you not test?

External dependencies

You should assume that all dependencies (libraries and frameworks, including those from the Android SDK) have been tested. Thus, you shouldn't test functionality of external libraries because the goal is to test things that you control, not a third party tool created by someone else.

> **Note**: In the real world, sometimes some of those dependencies are *not* tested. So, as a rule of thumb, when you have to choose between two or more libraries that have the same functionality, you should go with the one that has tests. This assures you that the features of the library work as expected and that the library developers won't break the features when adding new features and releasing new versions.

Autogenerated code

You shouldn't write tests for autogenerated code. Following the previous principle, it's supposed to be that the library or tool that generates code is tested properly.

When should you not test?

Throwaway/prototype code

Usually, when writing a Minimal Viable Product (MVP), you should focus on just writing the features so the client can get a feeling of what the final product could be.

However, all the stakeholders need to understand that all the code (or almost everything) you wrote will be thrown away. In this case, it doesn't make sense to write any kind of tests.

Code you don't have time to test

This is a controversial topic. Often, developers get stuck in a rut wherein they are fighting fires instead of proactively writing quality code, and they are not given the time to address code quality.

If you are working on a cash-strapped startup, where requirements are changing rapidly, that extra time to test could cause this fledgling company to miss key deadlines, not iterate fast enough, fail to raise it's next round of funding and go out of business.

On a new greenfield project, writing tests can double the amount of time to get features out in the short term. But, as the project gets larger, the tests end up saving time. Writing tests has its benefits; however, it'll take time to write and maintain tests. You and your team will need to make sure that you understand the trade-offs when determining which path you want to take.

A Note on Technical Debt

When you take out a financial loan, you get the benefit of an immediate infusion of cash. But a lender charges you **interest** on the loan in addition to the **principal**, all of which you will need to pay back. If you take on too much debt, you can end up in a situation where it is impossible to pay back the loan. In this case, you might have to declare **bankruptcy**.

Technical debt has many parallels to financial debt. With **technical debt** you make trade offs in your code, such as not writing unit tests, not refactoring, having less stringently quality standards, etc. to get features out quicker. This is analogous to getting a financial cash infusion. But as the code base grows, the lack of tests increase the number of regressions, bugs, time it takes to refactor and QA time. This is analogous to **interest** on a financial loan. In order to pay off that debt you start to add unit tests to your code. That is analogous to paying down the **principal** on a loan. Finally, if too many shortcuts are taken for too long, the project may reach a point where it is more advantageous to scrap the entire project and start with a clean slate. That is the same as declaring **bankruptcy** to get relief from too much financial debt.

Code spikes

At some point, you may find yourself working with a new library or, perhaps, you may realize that you aren't sure how to implement something. This makes it very difficult to write a test first because you don't know enough about how you are going to implement the functionality to write a meaningful failing test. In these instances, a **code spike** can help you figure things out.

A code spike is a throwaway piece of untested code that explores possible solutions to a problem. This code should not be considered **shippable**. Once you have a solution, you will want to delete your spike and then build up your implementation using **TDD**.

What is test coverage?

You can measure how many lines of code of your app have been executed when you run your tests. An app with a high test coverage percentage "suggests" that it works as expected and has a lower chance of containing bugs.

You may have asked yourself how many tests should you write. As mentioned before, you should at least write those that cover the most common scenarios.

In general, you can think of this metric as follows:

$$Test\ Coverage\ \% = \frac{Lines\ of\ code\ called\ by\ the\ test\ suite}{Total\ lines\ of\ code} * 100$$

Criterion

To measure, there are several coverage criterion that you may choose. The most common are:

- **Function/method coverage**: How many functions have been called?
- **Statement coverage**: How many statements of each function have been executed?
- **Branch coverage**: Has each branch in an `if` or a `when` statement been executed?
- **Condition coverage**: Has each subcondition in an `if` statement been evaluated to `true` and also to `false`?

For example, suppose that the following code is part of a feature of your app:

```
fun getFullname(firstName: String?, lastName: String?): String {
  var fullname = "Unknown"
  if (firstName != null && lastName != null) {
    fullname = "$firstName $lastName"
  }
  return fullname
}
```

Having at least one test that calls this function would satisfy the function/method coverage criteria.

If you have a test that calls `getFullname("Michael", "Smith")` you would satisfy the statement coverage criteria, because every statement would be executed.

If you also have a test calling `getFullname(null, "Smith")`, now it complies with branch coverage criteria, because the line inside the `if` is not executed and the previous test that called `getFullname("Michael", "Smith")` executes the line inside the `if` statement.

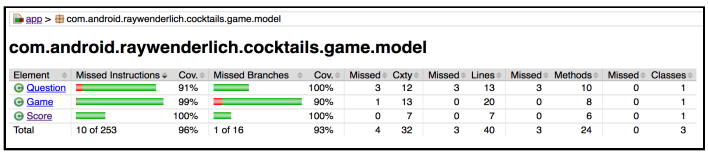
To satisfy the condition coverage criteria, you need tests that call `getFullname(null, "Smith")` and `getFullname("Michael", null)` so that each subcondition, `firstName != null` and `lastName != null` would evaluate to `true` and `false`.

Tools

There are tools that can assist you to measure the test coverage metric.

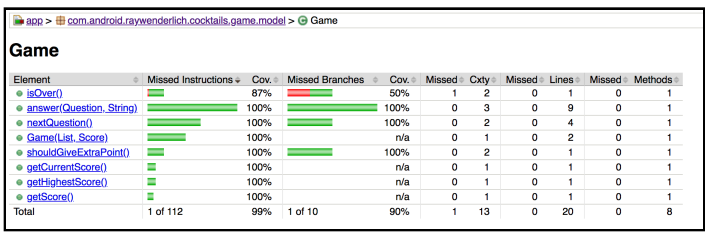
JaCoCo (Java Code Coverage Library) is one of them. Don't worry, it handles Kotlin as well!

This library generates a report for you to check which lines were covered by your tests (green) and which ones were not (red or yellow).


app > com.android.raywenderlich.cocktails.game.model

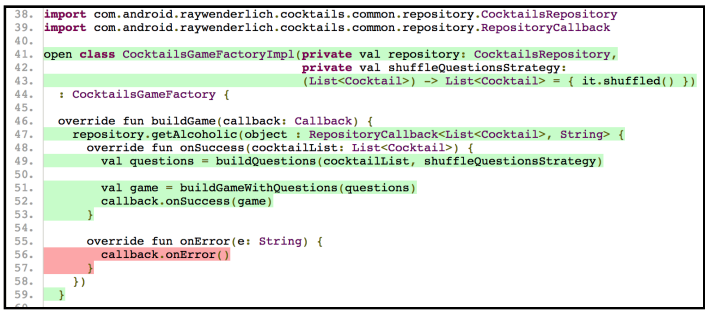
com.android.raywenderlich.cocktails.game.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Question		91%		100%	3	12	3	13	3	10	0	1
Game		99%		90%	1	13	0	20	0	8	0	1
Score		100%		100%	0	7	0	7	0	6	0	1
Total	10 of 253	96%	1 of 16	93%	4	32	3	40	3	24	0	3

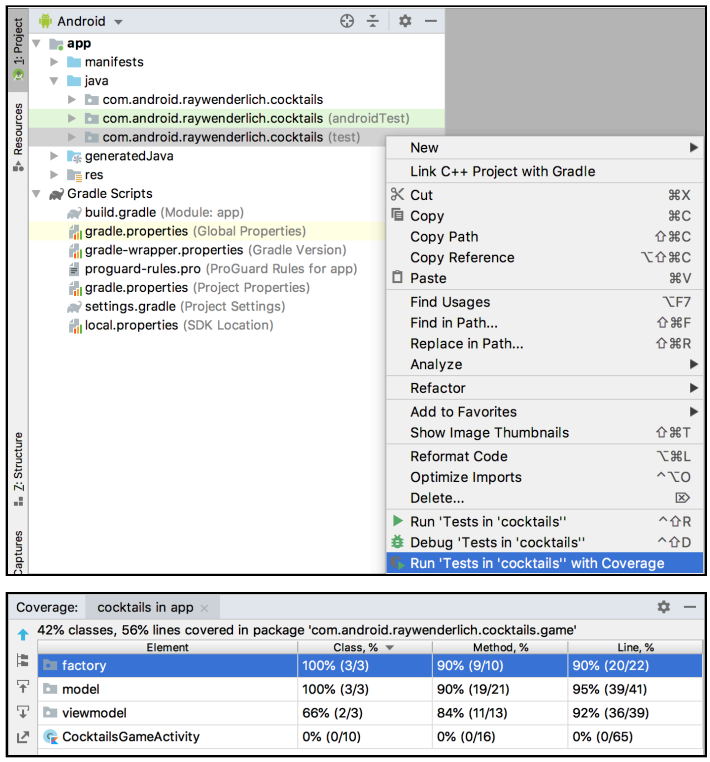

app > com.android.raywenderlich.cocktails.game.model > Game

Game

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isOver()		87%		50%	1	2	0	1	0	1
answer(Question, String)		100%		100%	0	3	0	9	0	1
nextQuestion()		100%		100%	0	2	0	4	0	1
Game(List, Score)		100%		n/a	0	1	0	2	0	1
shouldGiveExtraPoint()		100%		100%	0	2	0	1	0	1
getCurrentScore()		100%		n/a	0	1	0	1	0	1
getHighestScore()		100%		n/a	0	1	0	1	0	1
getScore()		100%		n/a	0	1	0	1	0	1
Total	1 of 112	99%	1 of 10	90%	1	13	0	20	0	8


```
38. import com.android.raywenderlich.cocktails.common.repository.CocktailsRepository
39. import com.android.raywenderlich.cocktails.common.repository.RepositoryCallback
40.
41. open class CocktailsGameFactoryImpl(private val repository: CocktailsRepository,
42.                                     private val shuffleQuestionsStrategy:
43.                                     (List<Cocktail>) -> List<Cocktail> = { it.shuffled() })
44.   : CocktailsGameFactory {
45.
46.   override fun buildGame(callback: Callback) {
47.     repository.getAlcoholic(object : RepositoryCallback<List<Cocktail>, String> {
48.       override fun onSuccess(cocktailList: List<Cocktail>) {
49.         val questions = buildQuestions(cocktailList, shuffleQuestionsStrategy)
50.
51.         val game = buildGameWithQuestions(questions)
52.         callback.onSuccess(game)
53.       }
54.
55.       override fun onError(e: String) {
56.         callback.onError()
57.       }
58.     })
59.   }
```

Android Studio also comes with a **built-in feature** to run tests with Coverage.



100% coverage?

In real-world apps, reaching a test coverage of 100%, no matter which criterion you use, is almost impossible to achieve. It often doesn't add value to test all methods, of all the classes, all of the time.

For example, suppose you have the following class:

```
data class Pet(var name: String)
```

You shouldn't write the following test:

```
fun whenCreatingPetWithName_shouldTheNameSetFromTheConstructor()
{
  val aName = "Rocky"
  val aPet = Pet(aName)

  if (aPet.name == aName) {
    print("Success\n")
```

```
    } else {
      throw AssertionError("Invalid pet name")
    }
  }
```

In this case, you are testing a feature (getting and setting a property) of a Kotlin data class that is auto-generated for you!

Test coverage gives you an exact metric of how much of your code has *not* been tested. If you have a low measure, then you can be confident that the code isn't well tested. The inverse however is not true. Having a high measure *is not sufficient* to conclude that your code has been thoroughly tested.

If you try to reach 100% test coverage, you'll find yourself writing meaningless, low-quality tests for the sake of satisfying this goal.

Neither you nor any team member should be obsessed with a test coverage of 100%. Instead, make sure you test the most common scenarios and **use this metric to find untested code** that **should** be tested.

If you feel that writing a particular test is taking too long, you might want to take a step back and evaluate if that test is adding enough value to justify the effort. Also, if a simple fix is causing a lot of changes to your tests, you may need to look at refactoring your tests or implementation to make them less brittle.

At the end of the day, your goal is to create software that provides value to its users. If you are doing TDD well, as your project gets larger, the total amount of effort spent on tests, implementation and QA should be the same or less than if you were creating the same product, with the same level of quality without doing TDD. That said, a project that is doing a good job at TDD may still take more development effort than a project that is not because the project with TDD will have a higher level of quality. The key is finding the right balance for your project.

Key points

- A test is a procedure used to evaluate if a method, an entire class, a module or even a whole application behaves correctly.
- This book focuses on writing tests before implementing the features.
- You should write tests to have confidence when refactoring.
- Tests also act as complementary documentation of the application features.

- The tests you write should be short, simple to read and easy to follow.
- You should only write tests related to the logic of your application.
- You can use test coverage tools to find untested code that should be tested.

Where to go from here?

Congratulations! Now you should understand what a test is, why it matters and the coverage metric.

In the next chapter, you'll find out what Test Driven Development (TDD) is and what the benefits are of writing tests before writing the feature. In the following chapters, you'll also start writing apps with their corresponding tests.

Where to Go From Here?

We hope you enjoyed this sample of *Android Test-Driven Development by Tutorials*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

Chapter 1: Introduction: If you're new to testing and want an overview of what this book covers, start with chapter 1 for a gentle introduction into testing.

Chapter 2: What Is a Test?: Do you know why you should test or what to test? In this chapter, "What is a Test?", you'll learn both of these and also dive into code coverage.

Chapter 3: What Is TDD? Now that you know what a test is, start writing your first tests! Not only will you start writing tests, but you'll also do so in a test-driven development way.

Chapter 4: The Testing Pyramid: If you're wondering how UI tests, integration tests and unit tests all fit into one application, learn about the testing pyramid and how you can structure various types of tests for your app.

Chapter 5: Unit Tests: Get your drink ready as you explore a cocktail app and write your unit tests in Android practicing test-driven development.

Chapter 6: Architecting for Testing: The architecture of your project can make or break your testing experience. Learn the pros and cons of each architecture and how they affect your tests.

Chapter 7: Introduction to Mockito: Level up your testing knowledge leveraging Mockito as you learn the basics of how to use mocks and spies in your tests.

Chapter 8: Integration: Often, you're going to have to test the interaction between objects in your application. In this chapter, you'll practice writing integration tests by working on a Wishlist app.

Chapter 9: Testing the Persistence Layer: Get started learning how to test the persistence layer in your app. In this chapter, you'll learn how to handle statefulness in your tests and strategies for creating randomized test data.

Chapter 10: Testing the Network Layer: While HTPP requests can be unpredictable, your tests don't have to be. In this chapter, learn how to write predictable network tests working on the Punchline app. Learn some random jokes on the way as well!

Chapter 11: User Interface: UI tests allow you to test your app end-to-end without having to manually click-test your app. Learn the fundamentals of UI tests using the Espresso library.

Chapter 12: Common Legacy App Problems: Automated tests can help catch bugs in all applications, including legacy applications. Learn how to work around technical debt and apply testing techniques.

Chapter 13: High-Level Testing with Espresso: Get started with the Coding Companion Finder app as you work through a legacy application and write UI tests using Espresso.

Chapter 14: Hands-On Focused Refactoring: Refactoring a legacy application doesn't have to be scary once you have some tests for it. Learn how to take small steps and move slowly as you keep your test suite green as you make changes to your app.

Chapter 15: Refactoring Your Tests: Your tests are code too. In this chapter, you'll learn how to refactor both your app code and test code to make your tests reliable and maintainable.

Chapter 16: Strategies for Handling Test Data: Learn multiple ways to handle test data in your Android tests by learning by tutorials. In this chapter, you'll explore a variety of ways of using test data in your tests.

Chapter 17: Continue Integration & Other Related Tools: Explore what it means to have continuous integration for your tests and the tools you can use to achieve it. You'll learn the pros and cons of different CI strategies in this chapter.

Chapter 18: Testing Around Other Components: Set boundaries in your test and use strategies to help to interact with other libraries and parts of the Android framework.

Chapter 19: Other Related Techniques: Learn about other techniques that can complement your TDD process to make your team more collaborative and therefore effective.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/android-test-driven-development-by-tutorials

We hope you enjoy the book!

— The *Android Test-Driven Development by Tutorials* Team

Learn Android Test-Driven Development!

Writing apps is hard. Writing testable apps is even harder, but it doesn't have to be. Reading and understanding all the official Google documentation on testing can be time-consuming — and confusing.

This is where Android Test-Driven Development comes to the rescue! In this book, you'll learn about Android Test-Driven Development the quick and easy way: by following fun and easy-to-read tutorials.

Who This Book Is For

This book is for intermediate Android developers who already know the basics of Android and Kotlin development but want to learn Android Test-Driven Development.

Topics Covered in Android Test-Driven Development:

- **Getting Started with Testing:** Learn the core concepts involved in testing including what is a test, why should you test, what should you test and what you should not test.
- **Test-Driven Development:** Discover the Red-Green-Refactor steps and how to apply them.
- **The Testing Pyramid:** Learn about the different types of tests and how to organize them.
- **Unit Tests:** Learn how to start writing unit tests with TDD using JUnit and Mockito.
- **Integration Tests:** Writing tests with different subsystems is a must in today's complex application world. Learn how to test with different subsystems including the persistence and network layers.
- **Architecting for Testing:** Explore how to architect your app for testing and why it matters.
- **TDD on Legacy Projects:** Take your TDD to the next level by learning how to apply it to existing legacy projects.
- **And much more,** including Espresso tests, UI tests, code coverage and refactoring.

One thing you can count on: after reading this book, you'll be prepared to take advantage of Android Test-Driven Development in your own apps!

About the Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.