The Pragmatic Programmers

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

Ken Kousen

Foreword by Venkat Subramaniam

Edited by Margaret Eldridge

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

by Ken Kousen

Version: P1.0 (January 2023)

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/#about-ebooks. To learn more about this book and access the free resources, go to https://pragprog.com/book/mockito, the book's homepage.

Thanks for your continued support,

The Pragmatic Bookshelf

The team that produced this book includes: Dave Rankin (CEO), Janet Furlow (COO),
Tammy Coron (Managing Editor), Margaret Eldridge (Development Editor),
Andy Hunt and Dave Thomas (Founders)

For customer support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

To Ginger and Xander, the people that matter most to me in the world.

Table of Contents

Foreword to *Mockito Made Clear*

Acknowledgments

Preface

1. Build a Testing Foundation

- Saying Hello to Mockito
- Counting Astronauts by Spaceship
- Creating the Basic Classes
- Adding Unit and Integration (End-to-End) Tests
- Wrapping Up

2. Work with the Mockito API

- Selecting Our System to Test
- Creating Mocks and Stubs
- Setting Expectations
- Using Mocks and Stubs in the Astro Project
- Wrapping Up

3. Use Built-in and Custom Matchers

- Using the Existing Argument Matchers
- Creating Custom Argument Matchers
- Verifying the Order of Methods Called
- Wrapping Up

4. Solve Problems with Mockito

- Deciding Between Mockito and BDDMockito
- Testing void Methods Using Interactions
- Capturing Arguments
- Setting Outputs Based on Inputs with Custom Answers
- Spying to Verify Interactions
- Wrapping Up

5. Use Mockito in Special Cases

- Mocking Final Classes and Methods
- Using Mockito with Android
- Mocking Static Methods
- Mocking Constructors
- Working with the Spring Framework
- Deciding When *Not* to Use Mockito
- Wrapping Up

A1. Running Mockito Tests

- Steps Common to All JUnit Versions
- JUnit 5
- JUnit 4

Early Praise for *Mockito Made Clear*

You can avoid mocking in many cases, but it is essential to understand your mocking framework when required. *Mockito Made Clear* is a valuable source to quickly get the most out of Mockito, from simple mocking to complex interactions and solving difficult-to-do accessibility problems like static and finals.

→ Daniel Hinojosa
Consultant, Programmer, Speaker, and Trainer

Since Mockito's inception in 2007, the library has gained so many features that only few developers keep a full overview. This book summarizes them both neatly and briefly. If you are using Mockito, novice or experienced user, this book is for you!

→ Rafael Winterhalter
Software Consultant

Ken did an amazing job of showing exactly what Mockito is, how to use it, and when not to use it. Definitely a must-read for anyone willing to learn how to write good tests.

→ Marcin Grzejszczakr
Book and Video Course Author and International Speaker

Foreword to *Mockito Made Clear*

Your ability to write automated tests to verify the behavior of code is an essential skill for sustainable agile development. Writing tests is easy when the unit of code has no dependencies. It gets hard when the code under test is coupled to other prices of code or services. As a first step to deal with the dependencies, we should decouple, move dependencies away, or even remove them where possible. A good design keeps the dependencies low and the code as loosely coupled as possible.

Once we distill the dependencies to the bare minimum, we have to ultimately contend with them to write effective tests. That's where this books comes in—kudos for picking it up.

You need two things to be able to write tests for code with dependencies. First you need an awesome library to easily create test doubles—the stubs, mocks, and spies—so you can focus on the design instead of fighting the dependencies. For Java and most languages on the JVM, Mockito has become a clear winner over the years. Second, along with knowing good ways to write tests, you need to know how to use the library really well.

Just like you'd want a library that has been tried and tested, one that's been around for a while, you'll want to learn from someone who has a lot of experience teaching, especially the very topic you're interested in learning. You've not only found a book for the topic that you'll benefit learning but also a wonderful instructor who's one of the most capable to write on this topic. Sit back, relax, get your favorite drink, fire up your IDEs, and let Ken lead you into this journey of creating tests to deal with dependencies.

Dr. Venkat Subramaniam, award-winning author and founder, Agile Developer, Inc.

Acknowledgments

This book took much longer to write than I expected. Although I could say that about every book I've ever worked on, the real reason this book took so long is that I kept discovering interesting capabilities in the Mockito framework I either hadn't known or hadn't understood. Even though I'd been using Mockito for years, both in practical work and teaching it in training classes, I had no idea there was so much to it. That's the best part of writing a book—digging into the parts of the code you'd not normally encountered, and Mockito turns out to have parts like that in abundance.

I definitely want to thank the Mockito team for building and maintaining such a powerful tool. Rather than list all the contributing members here, let me specifically mention Rafael Winterhalter, who made extensive recommendations on a beta version of the book that improved the final product immeasurably. I need to thank the other reviewers who submitted comments as well, including David Chelimsky, Bauke Scholtz, Ludovico Fischer, Marcin Grzejszczak, Marcin Zajaczkowski, and Daniel Hinojosa. I'm very grateful for your contributions and support.

Venkat Subramaniam also reviewed the book, and I appreciate that, but I want to especially thank him for writing the Foreword. I'm always glad to have him involved in any project, and we had several "spirited" discussions about Mockito, both in terms of how to use it and how to write about it. I try not to tell him too often because I don't want to feed his ego too much, but it's an honor to consider him a friend as well as a colleague.

I've been teaching training classes and making presentations involving Mockito on both the O'Reilly Learning Platform and the No Fluff, Just Stuff conference

tour, and I’m grateful to both for the opportunity. There’s no better way to find the gaps in your knowledge than to present information to a group of intelligent professionals, and I learned a lot about both Mockito and testing in general that way. I also want to thank my editor, Margaret Eldridge, for providing just the right combination of feedback, criticism, and support I needed to keep the project moving, even as other professional commitments interfered. In fact, the entire team at Pragmatic Programmers has been a pleasure to work with, and I highly recommend them to anyone who is thinking about writing a technical book.

Finally, and most especially, I’d like to thank my wife, Virginia, and my son, Xander, for their patience with me as I navigated my way through yet another writing project. I promise to take a break after this one, and this time I really mean it.

Ken Kousen

ken.kousen@kousenit.com

January, 2023

Preface

This book is about the Mockito testing framework, but it’s not an exhaustive reference. No books these days try to do that. For any library, the online documentation beats printed books every time because it keeps up with new versions of the software and can be adjusted to show new features and bug fixes. Where books become valuable is as a front end to the online docs, in that they help you understand what you’re getting into and what to expect when you get there. Books are curated content. They try to show you what’s important and why.

For Mockito, a book like this one is especially important, because the online documentation contains significant gaps. The docs were written by developers who understand tools like Mockito that generate mocks, stubs, and spies and who help you use them. That’s good, in that the docs focus on exactly how to make the framework do what you want. The downside is that you need to understand the problems that Mockito solves ahead of time, and unless you’ve already had experience with similar tools, those concepts are probably not obvious.

This book is intended to fill that gap. It’s about the concepts behind Mockito more than showing the details of how to use it. Those details are included, of course, for a set of representative examples, but the goal is understanding rather than the specific details of the syntax. The online docs are good for syntax (the classes and methods in the API) and, to a lesser extent, semantics (how to use the tool to do what you want), but they don’t explain why Mockito is useful in the first place or how it can make your job easier.

This is exactly what the *Pragmatic Answers* series is all about. *Answers* books

cover topics that are too big for a simple blog post but probably not large enough to warrant a 300-page book. Books in this series are short and sweet. The goal is to get you up and running with the tool in a way that you could devour in a weekend if necessary. The end result is to get you started and to help you understand the goals, the strategies, and the basic tactics of the framework. Then you can find any specific details in the online resources and know how to apply them.

Hopefully you'll find this book a helpful addition to your library and it'll help you write more productive tests in your Java systems, to be sure your code is doing what you want.

One note about software versions: the current version of Mockito at the time of this writing is 4.0.0. You may be using a version from the 2 line or the 3 line. That's fine, because these are the only differences:

- Mockito 3 is just Mockito 2 with a required Java version of 1.8 or above.
- Mockito 4 is just Mockito 3 with deprecated items removed.
- Mockito 5 (soon to be released) is just Mockito 4 with the inline mock maker included in the core.

In other words, all the code in this book is written for Mockito 4 but should work on any Mockito version 2.* or above. The Java version will be 11 because even though the current LTS (Long Term Support) version is 17, not many users in the community have moved to that version. All the code in the book will work on all Java versions from 1.8 through 19.

The tests are all based on the JUnit testing framework.[1] Appendix 1, *Running Mockito Tests*, discusses how to set up a project to include Mockito with both JUnit 4 and JUnit 5, using either the Gradle or Maven build tools, and then how to run the associated tests.

Now it's time to start looking at the types of problems Mockito is designed to solve.

Footnotes

[1] https://junit.org

Chapter 1

Build a Testing Foundation

Why do we need a tool like Mockito, and how does it makes our lives as developers easier?

Like many developers, I was a career changer. I came from engineering, where I spent my time with a lot of math and a lot of (shudder) Fortran. Eventually I switched to software, adopted Java in its early years, and retrained for IT. One unexpected benefit of this transition was that the next time I visited my parents, my father told me something in the house was broken, and I was able to reply, “Sorry, that’s a hardware problem.”

I feel that way about most hardware, and, to be honest, about most software too. If the code works, I’m happy. If it doesn’t, then I have to narrow down where it’s failing. Since everything is interconnected, it can be challenging to isolate a particular component to verify it’s working correctly.

Mockito is a tool that helps you isolate particular components of software. You use Mockito to replace dependencies of the components you’re testing so that methods in each dependency return known outputs for known inputs. That way, if an error occurs when testing the component, you know where and why.

One challenge with using Mockito is that it rests on a foundation of testing principles that you need to know to use the tool correctly. In this chapter we’ll use an interesting example to illustrate those foundations so that we can then focus on how to make Mockito do what you want it to do.

If you’re the kind of person who likes definitions, you can read about the differences between mocks, stubs, and spies in *Mockito Terminology (Mocks Aren’t Stubs)*. Otherwise, we’ll jump right into a project that is simple but sufficiently complex that we need tests to be sure it’s working correctly. The tests will give you confidence that the code is doing its job properly and show where unit testing, integration testing, and functional testing help in development. Along the way, we’ll highlight a series of steps to take whenever you want to use Mockito in your own projects, while still stubbing methods that you don’t need to test.

Saying Hello to Mockito

Ever since the original publication of *The C Programming Language*[2] by Kernighan and Richie (1978), every programming language and framework is required, by law, to include a Hello, World! example. We'll look at the Mockito version in this section, explain what Mockito does and what it is for, and introduce part of a larger system that we'll use for further discussions later in the book.

For Java, the classic Hello, World! program is this, of course:

HelloWorld.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

The `HelloWorld` class contains a `main` method, but `main` returns `void`, and `void` methods are tough to test. Mockito is a testing library, so let's modify this to make a class that welcomes the user with a simple `String` returned from a `greet` method:

HelloMockito.java

```
public class HelloMockito {
    private String greeting = "Hello, %s, from Mockito!";

    public String greet(String name) {
        return String.format(greeting, name);
    }
}
```

That's better. It's easy enough to create a JUnit 5 test case for the `greet` method:

HelloMockitoTest.java

```
import org.junit.jupiter.api.Test;
```

```
class HelloMockitoTest {
    private HelloMockito helloMockito = new HelloMockito();

    @Test
    void greetPerson() {
        String greeting = helloMockito.greet("World");
        assertEquals("Hello, World, from Mockito!", greeting);
    }
}
```

Since the `HelloMockito` class doesn't have any dependencies, Mockito isn't needed at all. To see where Mockito is useful, let's add a couple of dependencies, one to look up a person who we can greet by name and one to translate the resulting message into different languages:

HelloMockitoRevised.java

```
public class HelloMockito {
    private String greeting = "Hello, %s, from Mockito!";

    // Dependencies
    private final PersonRepository personRepository;
    private final TranslationService translationService;

    // Constructor to inject the dependencies
    public HelloMockito(PersonRepository personRepository,
                        TranslationService translationService) {
        this.personRepository = personRepository;
        this.translationService = translationService;
    }

    // Method we want to test
    public String greet(int id, String sourceLang, String targetLang) {
        Optional<Person> person = personRepository.findById(id);
        String name = person.map(Person::getFirst).orElse("World");
        return translationService.translate(
            String.format(greeting, name), sourceLang, targetLang);
    }
}
```

Now we're getting somewhere. The argument to the `greet` method is now intended to be an integer `id` for a `Person` object (a simple plain old Java object—

or POJO—that wraps a first name, a last name, and a date of birth). We retrieve that `Person` from a repository class representing some kind of persistent storage. Also, once we form the welcome greeting to be returned, we run that string through a translation service and send back the result.

Here are the two dependencies as Java interfaces. First, the `PersonRepository`, which has a lot more methods than just the `findById` method needed in `HelloMockito`:

PersonRepository.java

```
public interface PersonRepository {
    Person save(Person person);

    Optional<Person> findById(int id);

    List<Person> findAll();

    long count();

    void delete(Person person);
}
```

(If you're familiar with the Spring Data framework, these methods will look familiar. We'll be using this interface, combined with a `PersonService`, in several upcoming examples. We'll also look at how to use Mockito with Spring in *Working with the Spring Framework*.)

The method needed by our `HelloMockito` implementation is `findById`, which takes an integer `id` and returns an `Optional` wrapped around a `Person` if the `id` corresponds to an existing row in the database, or an empty `Optional` otherwise. The `greet` method in `HelloMockito` extracts the person's first name if the person exists or defaults to the word `World` if not.

Next is the `TranslationService`, which is also an interface, because there are many publicly available services to choose from, including writing our own:

TranslationService.java

```
public interface TranslationService {
    default String translate(String text,
                             String sourceLang,
                             String targetLang) {
        return text;
    }
}
```

Our default implementation is to return the input string. If we were to implement this in a real system, it would be a lot more complicated.

The `HelloMockito` class is now a lot harder to test because we have to do something about those two dependencies. Let's get some help from Mockito. Mockito will generate implementations of our dependencies for us, which we can configure to do what we want. We can then *inject* those dependencies into our class under test and update our tests accordingly.

Adding Mockito to the Project

First, to make Mockito available in our project, we'll update the build file to include Mockito. The following sample uses Gradle as the build tool, but the Maven version uses the same coordinates for the JUnit 5 and Mockito dependencies.

Assume the project is organized in standard Gradle or Maven structure, so application sources reside under `src/main/java` and test sources live under `src/test/java`. Further, assume the project has a Gradle build file called `build.gradle` in the project root directory. Here's that build file:

build.gradle

```
// Using Gradle version catalogs -- see gradle/libs.versions.toml
plugins {
    id 'java'
}

group 'com.kousenit'
version '1.0'

repositories {
```

```
    mavenCentral()
}

dependencies {
    // JUnit bundle (includes vintage engine)
    testImplementation libs.bundles.junit

    // Mockito bundle (inline and JUnit Jupiter engine)
    testImplementation libs.bundles.mockito

    // AssertJ
    testImplementation libs.assertj
}

tasks.named('test') {
    useJUnitPlatform()
}
```

Note this file relies on the new Gradle Version Catalogs introduced in Gradle 7.4,[3] which means the **gradle** folder includes a file called **libs.versions.toml**, with the following contents:

libs.versions.toml

```
[versions]
assertj = "3.23.1"
junit = "5.9.1"
mockito = "4.9.0"

[libraries]
assertj = { module = "org.assertj:assertj-core",
            version.ref = "assertj" }
junit-jupiter = { module = "org.junit.jupiter:junit-jupiter",
            version.ref = "junit" }
junit-vintage = { module = "org.junit.vintage:junit-vintage-engine",
            version.ref = "junit" }
mockito-inline = { module = "org.mockito:mockito-inline",
            version.ref = "mockito" }
mockito-junit = { module = "org.mockito:mockito-junit-jupiter",
            version.ref = "mockito" }

[bundles]
junit = ["junit-jupiter", "junit-vintage"]
mockito = ["mockito-inline", "mockito-junit"]
```

The TOML file contains the version numbers for each dependency and creates the `libs` reference used in the `build.gradle` build file.

Because we're using JUnit 5 for the tests, we included two dependencies for Mockito:

- The `mockito-core` dependency, and
- The JUnit 5 extension for Mockito, here called `mockito-junit-jupiter`.

You'll find the two dependencies listed as a *bundle* in the TOML file because they're used together.

If this project used Maven instead of Gradle, the same two dependencies would be required, just using the Maven XML-based syntax in `pom.xml` and not including the TOML file.

Finally, we need the method call `useJUnitPlatform` to tell Gradle that we're using JUnit 5 for our tests; otherwise, the tests won't be detected during a gradle build. Now we're ready to bring everything together for a complete test of Hello, World!

A Mockito Test Class with Everything

Let's annotate the resulting test class with `@ExtendWith(MockitoExtension.class)` and update the tests to set the expectations on each mock.

Demonstrating Many Mockito Capabilities in One Example

The following test has it all: the Mockito JUnit 5 extension, the `@Mock` and `@InjectMocks` annotations, setting expectations on the generated stubs using the `when/thenReturn` syntax, and even verifying that the mocked methods were called the right number of times in the right order.

We haven't discussed any of those features yet, so think of this example as a teaser for what's coming in the rest of the book.

Without further ado, our complete Hello, World! example:

HelloMockitoTestFull.java

```
@ExtendWith(MockitoExtension.class)
class HelloMockitoTest {
    @Mock
    private PersonRepository repository;

    @Mock
    private TranslationService translationService;

    @InjectMocks
    private HelloMockito helloMockito;

    @Test
    @DisplayName("Greet Admiral Hopper")
    void greetAPersonThatExists() {
        // set the expectations on the mocks
        when(repository.findById(anyInt()))
            .thenReturn(Optional.of(new Person(1, "Grace", "Hopper",
                LocalDate.of(1906, Month.DECEMBER, 9))));
        when(translationService
            .translate("Hello, Grace, from Mockito!", "en", "en"))
            .thenReturn("Hello, Grace, from Mockito!");

        // test the greet method
        String greeting = helloMockito.greet(1, "en", "en");
        assertEquals("Hello, Grace, from Mockito!", greeting);

        // verify the methods are called once, in the right order
        InOrder inOrder = inOrder(repository, translationService);
        inOrder.verify(repository).findById(anyInt());
        inOrder.verify(translationService)
            .translate(anyString(), eq("en"), eq("en"));
    }

    @Test
    @DisplayName("Greet a person not in the database")
    void greetAPersonThatDoesNotExist() {
        when(repository.findById(anyInt()))
            .thenReturn(Optional.empty());
        when(translationService
            .translate("Hello, World, from Mockito!", "en", "en"))
```

```
            .thenReturn("Hello, World, from Mockito!");

        String greeting = helloMockito.greet(100, "en", "en");
        assertEquals("Hello, World, from Mockito!", greeting);

        // verify the methods are called once, in the right order
        InOrder inOrder = inOrder(repository, translationService);
        inOrder.verify(repository).findById(anyInt());
        inOrder.verify(translationService)
            .translate(anyString(), eq("en"), eq("en"));
    }
}
```

One of the advantages of using a library like Mockito is that we can test a class with dependencies even before we've implemented those dependencies. This is the pattern we will use when a class has dependencies:

```
class ClassUnderTest {
    private DependencyOne dependency1;
    private DependencyTwo dependency2;

    public Result methodToBeTested(Object... args) {
        // use dependencies and arguments and return Result
        return result;
    }
}
```

We can test the `methodToBeTested` by mocking the two dependencies, injecting them into the class under test, and then calling the test method.

Mockito Test Process

Formally, the process we're using consists of:

1. *Creating stubs* to stand in for the dependencies.
2. *Setting expectations* on the stubs to do what you want.
3. *Injecting the stubs* into the class you're planning to test.
4. *Testing the methods in the class under test* by invoking its methods, which in turn call methods on the stubs.
5. *Checking* the methods work as expected.
6. *Verifying* that the methods on the dependencies got invoked the right number of times, in the right order.

You can use these steps every time you want to use Mockito (or, honestly, any similar tool) for replacing dependencies in the class under test, thereby writing true unit tests.

The rest of this book is about how to use Mockito to implement these steps. To begin, let’s consider a program that uses a RESTful web service to download and transform JSON data into Java objects and then post-processes those objects to answer interesting questions. The problem involves astronauts in space. I chose this project because (1) it’s reasonably small, and (2) astronauts in space are, by definition, cool.

Counting Astronauts by Spaceship

How many astronauts are on each ship in space right now? Let's talk through a modern Java solution that doesn't require many classes but uses a familiar architecture you'll find in many applications. We'll use this problem as an introduction to testing in general and Mockito in particular.

The final product will have three components:

1. A class called `AstroGateway` that accesses a RESTful web service to retrieve the astronaut data in JSON form and then converts the JSON structure to Java POJOs (plain old Java objects).
2. A class called `AstroService` that processes the Java POJOs and returns a `Map` of strings to integers, where the map keys are the spacecraft names and the map values are the number of astronauts aboard each.
3. An application class that drives the process, which in our case will be a series of test cases that invoke the right methods and print and verify the results.

A Note on Java Versions

Mockito requires only Java 8. The current Long Term Support (LTS) version of Java is 17, but as of 2022, the community is split between Java 8 and the previous LTS version, Java 11.

The implementation of this "astro" system uses the HTTP client added to Java in version 11, and 11 is the default version of Java we'll use throughout the book. None of the code uses records, sealed classes, pattern matching, switch expressions, or any of the other newer features added since 11.

For those people still on Java 8, first, you have my condolences, and second, I've included an alternative implementation of the gateway that uses the Retrofit library. Hopefully, this way, everybody will feel included.

We'll be testing an existing code implementation that we'll discuss as we go along. If you're curious about the details and want to see them now, take a look at the book's GitHub repository.[4] The relevant code is in the `com.kousenit.astro` package.

As a reminder, to add Mockito to a project, the only dependencies required are `mockito-core` and, when using the Mockito JUnit 5 extension, as here, `mockito-junit-jupiter`.

With the project configured for testing, let's add the classes to do the job.

Creating the Basic Classes

The astronaut data comes from the free RESTful web service at Open Notify[5] called People in Space.[6] It supports only HTTP GET requests, but it's free and doesn't require registration or any kind of key. It returns a JSON response showing all the astronauts in space at any given moment. If you send an HTTP GET request to http://api.open-notify.org/astros.json, you'll get back a response similar to this:

astro_data.json

```
{
  "message": "success",
  "people": [
    {
      "name": "Bob Hines",
      "craft": "ISS"
    },
    {
      "name": "Oleg Artemyev",
      "craft": "ISS"
    },
    ...,
    {
      "name": "Cai Xuzhe",
      "craft": "Tiangong"
    }
  ],
  "number": 10
}
```

As you can see from the sample, the response includes the total `number` of astronauts and a collection called `people` which contains `name` and `craft` combinations for each astronaut. Java POJOs make a convenient structure to hold the parsed JSON information. Only two Java classes are needed:

- `Assignment`, representing the astronaut `name` and `craft` pair, and
- `AstroResponse`, which shows the total `number` of people in space, a success `message`, and the `people` array or list of the assignments.

The Java classes will be implemented as traditional POJOs, meaning they'll have attributes that match the JSON properties, along with getter and setter methods. Here's the `Assignment` class:

Assignment.java

```
public class Assignment {
    private final String name;
    private final String craft;

    // constructors, getters and setters, toString
}
```

Here's the `AstroResponse` class:

AstroResponse.java

```
public class AstroResponse {
    private final int number;
    private final String message;
    private final List<Assignment> people;

    // constructors, getters and setters, toString
}
```

To access the RESTful web service, we'll use the Gateway[7] design pattern. A gateway is a class that encapsulates access to an external resource. In this particular case, we need only a single gateway, called an `AstroGateway`, that connects to the remote service, downloads the astro data, and converts the JSON response into records. Java is happiest when you use interfaces, and doing so here will make testing easier because we'll be able to substitute in a fake gateway when we need one. Therefore, add a `Gateway` interface that contains a single method called `getResponse`:

Gateway.java

```
public interface Gateway<T> {
    T getResponse();
}
```

The `getResponse` method returns an instance of its generic type.

The concrete class implementing the `Gateway` interface will be `AstroGateway`. You'll implement it using either the `HttpClient` API if you're using Java 11+, or the Retrofit 2[8] library if you're still on Java 8.

Abstraction Theater

Introducing an interface (`Gateway`) with only a single implementation (`AstroGateway`) may feel like an unnecessary distraction. In this particular case, there will be two implementations: one that uses the new `HttpClient` API added in Java 11 and one that uses the external Retrofit library, which you can use if Java 11 isn't available. Only the `HttpClient` version will be included here, but both are contained in the GitHub repository for the book.

We'll get to the tests next in *Adding Unit and Integration (End-to-End) Tests*. Here we're defining the basic classes, so let's also define the `AstroService` class, whose job is to convert the records returned by the gateway into a Java `Map`. The class needs only a single method, called `getAstroData`, with the following signature:

```
public Map<String,Long> getAstroData()
```

The `AstroService` class uses the gateway to access the remote web service. The gateway retrieves the remote data and turns it into either a `Success` instance containing an `AstroResponse` or a `Failure` instance containing any thrown exception if not. Assuming it's successful, the service then extracts the data we want and converts it to a map, where the keys are spacecraft names and the values are the number of astronauts aboard each one.

In the end, our application instantiates the `AstroService`, calls its `getAstroData` method, and processes the results. The results (based on the JSON data shown earlier) will be as follows:

```
3 astronauts aboard Tiangong
7 astronauts aboard ISS
```

Now let’s turn to the tests necessary to verify the implementation works as desired.

Adding Unit and Integration (End-to-End) Tests

For our purposes, an *integration test* is one that tests a system including components it interacts with (dependencies), whereas a *unit test* is one that tests a class in isolation with no interactions or dependencies.

Integration tests are often easier to understand because you're treating the system as it's used in practice, with no additional information about its implementation. In this case, the system is driven by the `getAstroData` method on the `AstroService`, which uses an `AstroGateway`. Its integration test looks like this:

AstroServiceTest.java

```
@Test
void testAstroData_usingRealGateway_withHttpClient() {
    // Create an instance of AstroService using the real Gateway
    service = new AstroService(new AstroGateway());

    // Call the method under test
    Map<String, Long> astroData = service.getAstroData();

    // Print the results and check that they are reasonable
    astroData.forEach((craft, number) -> {
        System.out.println(number + " astronauts aboard " + craft);
        assertAll(
            () -> assertThat(number).isPositive(),
            () -> assertThat(craft).isNotBlank()
        );
    });
}
```

We add the `AstroGateway` to the `AstroService` using a constructor argument. If we want to replace the actual `AstroGateway` with a fake instance, we can use that constructor to inject it into our service. The important point is that the gateway is a *dependency* of the service.

To use Mockito, you first need to determine the dependencies of the class you're testing. For `AstroService`, the only dependency is the `Gateway`.

The test uses the service to call the `getAstroData` method and checks the returned map. The test uses the `forEach(BiConsumer)` method added to the `Map` interface in Java 8, which automatically separates the keys from the values, so it's easy enough to print and check the results. Then the test confirms the returned values using assertions from the AssertJ library.[9] The result checks the combination of `AstroService` and `AstroGateway` classes together, which is necessary if we want to believe the system is working properly.

Unit Tests Versus Integration Tests

The concept of a unit test originally meant a conceptual part of software that can work independently. The idea was that a developer could work on that *unit* and confirm it worked with tests that ignored the rest of the system. Later, you could bring different units back together. The tests that confirmed the entire system worked were called *integration* tests. The distinction between unit and integration tests was about testing code in isolation, as opposed to testing code brought together.

Over time, with object-oriented systems, *unit* tests grew to mean testing a single class in isolation, while *integration* tests checked a class connected with other classes. That definition is arguably much more fine-grained than the original concepts. Since virtually all classes depend on something outside themselves, you can argue that practically all tests are actually integration tests.

If you're interested in reading more, Martin Fowler's blog post[10] talks about meaning and use of the terms *unit* and *integration* testing.

In our integration test, we took a practical approach. The purpose of testing is to isolate problems. If we had a failing integration test, we'd use unit tests to isolate the failure to either the gateway or the service.

Failed Integration Tests Lead to Unit Tests

> The key question is always, "How do you know you're right?" If a test fails, you know there's a problem, but how easy is it to track down? Successful integration tests are necessary—otherwise, you don't know the system works.
>
> Failing integration tests are what lead you to unit tests, because if a unit test fails, you'll know quickly where and why.

Let's say our integration test failed and we now want to build a test for the `AstroService`.

A Stub for the Gateway

A true unit test for the `AstroService` needs a stand-in for the gateway—some kind of fake object that will return exactly what we want when the service calls its `getResponse` method. We don't want any problems with the gateway to affect the success or failure of our tests of the service. This stand-in for the gateway is called a *fake* object. From a Mockito point of view, the fake object is either a *mock* or a *stub*, depending on how it's used.

Before getting to that distinction, we can always provide our own fake object for the dependency. In this case, here's a class called `FakeGateway` that hard-codes a response:

FakeGateway.java

```
public class FakeGateway implements Gateway<AstroResponse> {
    @Override
    public AstroResponse getResponse() {
        return new AstroResponse(7, "Success",
            List.of(new Assignment("Kathryn Janeway", "USS Voyager"),
                new Assignment("Seven of Nine", "USS Voyager"),
                new Assignment("Will Robinson", "Jupiter 2"),
                new Assignment("Lennier", "Babylon 5"),
                new Assignment("James Holden", "Rocinante"),
                new Assignment("Naomi Negata", "Rocinante"),
                new Assignment("Ellen Ripley", "Nostromo")));
    }
}
```

Here's a unit test of the service that uses the `FakeGateway`:

AstroServiceTest.java

```
@Test
void testAstroData_usingOwnMockGateway() {
    // Create the service using the mock Gateway
    service = new AstroService(new MockGateway());

    // Call the method under test
    Map<String, Long> astroData = service.getAstroData();

    // Check the results from the method under test
    astroData.forEach((craft, number) -> {
        System.out.println(number + " astronauts aboard " + craft);
        assertAll(
            () -> assertThat(number).isPositive(),
            () -> assertThat(craft).isNotBlank()
        );
    });
}
```

Given the response returned by the mock gateway, the service now prints:

```
1 astronauts aboard Babylon 5
1 astronauts aboard Nostromo
1 astronauts aboard Jupiter 2
2 astronauts aboard USS Voyager
2 astronauts aboard Rocinante
```

While it was easy enough in this case to write our own mock gateway, two points are worth mentioning:

1. One job of Mockito is to automate the creation of fake objects like our `FakeGateway`. Mockito will generate a class that implements the `Gateway` interface for us. We can then tell it what to return when the `getResponse` method is called, and the rest happens automatically.
2. Our `FakeGateway` is either a mock or a stub, depending on how it's being used.

Mockito Terminology (Mocks Aren't Stubs)

So an object can be a mock or a stub, depending on use? Yes, and since that

statement is potentially confusing, let's define those terms.

In January of 2007, Martin Fowler wrote a blog post entitled "Mocks Aren't Stubs"[11] that is highly influential.

Since that article shaped how I think about mocks and stubs, I've based the following definitions on his thinking.

stub
: A *stub* is an object that stands in for dependencies and returns known outputs for known inputs.

mock
: A *mock* is an object that does what a stub does, plus it keeps track of interactions performed. By tracking interactions, a mock *verifies* that stubbed methods are called by the class under test the right number of times, with the correct parameters, in the right order.

For completeness let's add the definition of a spy.

spy
: A *spy* is an object that wraps the actual dependent object, which allows you to track calls to that object by the class under test. A spy is similar to a mock, in that it records which methods were called, but a spy invokes those methods on the actual underlying object instead of on a stub.

In a nutshell, that's how we'll be using the terms *mock*, *stub*, and *spy* throughout this book.

Why Not Write Your Own Stubs?

In simple cases like our example, it's easy enough to create the needed stub yourself, plug an instance into the class under test, and go from there. The `FakeGateway` is a stub. Stubs provide known outputs given known inputs. If you also want to verify afterward that the service called the `getResponse` method on the gateway exactly once, your same object would then be considered a mock. In each case *the class is the same*. The only difference is how you're using the

object in the test.

Writing your own stubs causes problems:

- If the interface you are trying to stub has many methods in it, and you plan to call only one or two, you still have to implement all the methods rather than just the ones you care about.
- If you're going to write the stub yourself, you're going to have to maintain it as well, so if the required methods change, you'll have to update the stub classes.
- If a class you plan to stub doesn't implement a convenient interface, you'll have to create a subclass instead. That may work, but what if the class is marked `final` and can't be extended at all?
- What if the methods you need to stub are `static`? You can't override `static` methods.
- How do you keep track that the stubbed methods are invoked the right number of times, in the right order, with the right parameters?

These problems are where Mockito helps. Mockito generates mocks and stubs for you. Mockito can generate classes that implement an interface. It can override classes (even, with a bit of help, `final` classes). You even have the capability to stub `static` methods. Mockito will implement all the methods in an interface so that they return default values, which you can then override.

Now that we've seen an integration test and a test with our own stubbed object, we're ready to use Mockito to automate the process. After we tackle the basics of the Mockito API, we'll return to the Astro project to see how to write true unit tests for the service.

Wrapping Up

So far, we've introduced that Mockito generates fake classes that stand in for actual dependencies so you can isolate the class you're testing. We've talked about using stubs when we need to simply provide known outputs for known inputs. When we want to check that the class under test called the methods on the mock the right number of times in the right order, we know to use a mock. In short, we've covered why you'd use the Mockito framework.

Now we're ready to get into the process of using Mockito in tests. In the next chapter, we'll create mocks, inject them into class under test, and get them to respond the way we want.

Footnotes

[2] https://en.wikipedia.org/wiki/The_C_Programming_Language

[3] https://docs.gradle.org/current/userguide/platforms.xhtml

[4] https://github.com/kousen/mockitobook

[5] https://open-notify.org

[6] http://open-notify.org/Open-Notify-API/People-In-Space/

[7] https://martinfowler.com/articles/gateway-pattern.xhtml

[8] https://square.github.io/retrofit/

[9] https://assertj.github.io/doc/

[10] https://martinfowler.com/articles/2021-test-shapes.xhtml

[11] https://martinfowler.com/articles/mocksArentStubs.xhtml

Chapter 2

Work with the Mockito API

I've always been a good test taker—part of what I call the Game of School, which is very different from getting an education. In the Game of School, your primary task is to figure out what teachers want to hear and how they want to hear it. That knowledge comes in handy when you want to pass a test without studying much. Just like that teacher, Mockito expects certain things, and right now they're a mystery.

The game begins with the Mockito API. You'll need to know what it does and how it expects to be invoked. We'll start with creating mocks, either using the `mock` method or using the annotations `@Mock` and `@InjectMocks`. Then we'll move on to setting expectations on our mocks, and finally to verifying that the class under test interacted with the mocks in the way we wanted.

Mockito is a large API, and you have several choices in how you can use it. But don't worry. Once you've finished this chapter, you'll be able to play the Game of Mockito well enough to win without studying—you won't have to spend hours digging into the JavaDocs either.

Selecting Our System to Test

We need a class to test that includes a dependency, so we'll use the `PersonRepository` interface, reproduced here:

PersonRepository.java

```
public interface PersonRepository {
    Person save(Person person);

    Optional<Person> findById(int id);

    List<Person> findAll();

    long count();

    void delete(Person person);
}
```

In a three-layered architecture typical of modern Java systems, you start with a presentation layer of controllers and views, then access a service layer that contains business logic and transaction boundaries, and finally go through a persistence layer to retrieve data from the database. The `PersonRepository` is part of that last layer, whose job is to convert table rows into Java classes and back again.

The class to test is called `PersonService` and uses a `PersonRepository` as an attribute:

PersonServiceAbbreviated.java

```
public class PersonService {
    private final PersonRepository repository;

    public PersonService(PersonRespository repository) {
        this.repository = repository;
    }

    // methods to be tested...
}
```

To act as our database, we'll use a Java `List<Person>` as an attribute of the `PersonServiceTest` class:

PersonServiceTest.java

```
private final List<Person> people = Arrays.asList(
    new Person(1, "Grace", "Hopper", LocalDate.of(1906, Month.DECEMBER, 9)),
    new Person(2, "Ada", "Lovelace", LocalDate.of(1815, Month.DECEMBER, 10)),
    new Person(3, "Adele", "Goldberg", LocalDate.of(1945, Month.JULY, 7)),
    new Person(14, "Anita", "Borg", LocalDate.of(1949, Month.JANUARY, 17)),
    new Person(5, "Barbara", "Liskov", LocalDate.of(1939, Month.NOVEMBER, 7)));
```

We'll start by adding a method to the service that retrieves all the last names of the people in the database. In keeping with the principles of Test Driven Development (TDD), we'll start with the test for the `getLastNames` method.

PersonServiceTest.java

```
@Test
void getLastNames_usingMockMethod() {
    // create a stub for the PersonRepository
    PersonRepository mockRepo = new PersonService(mockRepo);

    // Set the expectations on the mock...
    when(mockRepo.findAll()).thenReturn(people);

    // Inject the mock into the service
    PersonService personService = new PersonService(mockRepo);

    // Get the last names (this is the method to test)
    List<String> lastNames = personService.getLastNames();

    // Check that the last names are correct (using AssertJ)
    assertThat(lastNames)
        .contains("Borg", "Goldberg", "Hopper", "Liskov", "Lovelace");

    // Verify that the service called findAll on the mockRepo exactly once
    verify(mockRepo).findAll();
}
```

As the test shows, to invoke the method to be tested you need an instance of `PersonService`, and to instantiate that class you need a `PersonRepository`. Now let's

see how to create the stub representing the `PersonRepository`, first by hand and then automated using Mockito.

Creating Mocks and Stubs

Before we use Mockito, note that we could write our own stub implementation, called, for example, `InMemoryPersonRepository`. Here's an abbreviated version of that class (the complete version can be found in the GitHub repository):

InMemoryPersonRepository.java

```
public class InMemoryPersonRepository implements PersonRepository {
    private final List<Person> people =
        Collections.synchronizedList(new ArrayList<>());

    @Override
    public final Person save(Person person) {
        synchronized (people) {
            people.add(person);
        }
        return person;
    }

    @Override
    public final void delete(Person person) {
        synchronized (people) {
            people.remove(person);
        }
    }

    // ... other methods from PersonRepository ...
}
```

You may have noticed the issues with this approach:

1. We have to implement all the methods in the interface, even though our test requires only one of them.
2. We have to maintain this class, so if our repository methods change, the stub will have to be updated as well.
3. If we want to test failure modes, validation, or exceptional cases, we'll need either additional classes or some logic that lets us choose those alternatives.

Now let's use the Mockito API instead. The first step is to create a mock

repository. Mockito has two ways of doing that: using the static `mock` method on the `Mockito` class or using annotations. We'll take each in turn.

First, we'll use the static `Mockito.mock(Class)` method to create our stub.

Using the mock Method

The `mock` method in the `Mockito` class has the following signature:

```
public static <T> T mock(Class<T> classToMock)
```

You tell Mockito which class or interface to mock, and Mockito returns the mock object. Here's the line of code to do that in our test:

```
PersonRepository mockRepo = mock(PersonRepository.class);
```

That's easy enough. Mockito will generate a class that implements this interface, where all the methods will return their default values. If you need convincing, the following test demonstrates the default implementations:

PersonServiceTest.java

```
@Test
void defaultImplementations() {
    PersonRepository mockRepo = mock(PersonRepository.class);
    assertAll(
        () -> assertNull(mockRepo.save(any(Person.class))),
        () -> assertTrue(mockRepo.findById(anyInt()).isEmpty()),
        () -> assertTrue(mockRepo.findAll().isEmpty()),
        () -> assertEquals(0, mockRepo.count())
    );
}
```

Smart Nulls

Mockito does better than just return nulls for mocked reference types. For example, if the return type of a method is a list, Mockito will return an empty list. Most of the time you don't need to be concerned with this, but it's good to know that Mockito is trying to be helpful.

The static methods `any(Class)` and `anyInt` come from the `ArgumentMatchers` class, which we'll cover in detail in Chapter 3, *Use Built-in and Custom Matchers*. Here, they simply represent invoking a method with any instance of the `Person` class or any integer, respectively.

The mock returns a `null` for the `save` method, empty `Optional` instances for the two finder methods, and 0 for `count`. The `delete` method already returns `void`, so the mock doesn't change that and therefore doesn't need to be tested.

Returning to our mock, for our test of the `getLastNames` method on the service to pass, we need to change the default behavior of the `findAll` method in the `PersonRepository`. One way to do that is to use the combination of the static `when` method along with the chained `thenReturn` method. We need to tell the mock that when the service invokes the `findAll` method, it should return our prepared list of people:

```
when(mockRepo.findAll()).thenReturn(people);
```

We don't need to set the expectations on any other methods in the interface. That's one of the big advantages of using Mockito.

Now we can run our test, and it should pass. The last line of the test is a verification step, repeated here:

```
verify(mockRepo).findAll();
```

This step is optional but does check that the protocol was followed—meaning that our class under test invoked the `findAll` method on the mock exactly once.

Hopefully, you'll find the process intuitive. It does include two steps that can get tedious, because they'll appear in every test we write:

1. We need to create the mock every time, and
2. We need to inject the mock into the class under test every time.

While doing so manually, as shown, is fine, Mockito provides a simpler alternative using annotations.

Using Annotations to Create the Mocks

Mockito provides two annotations for mocks: `@Mock` and `@InjectMocks`. You add the `@Mock` annotation to any dependencies in your class that you want Mockito to mock, and add `@InjectMocks` to the class under test. Note this is done inside the test class; no changes are needed for the class under test. The combination of one or more attributes with `@Mock` and a single `@InjectMocks` on the class under test tells Mockito to create the necessary mocks and do its best to inject them into the class under test.

An example shows how to use the `@Mock` and `@InjectMocks` annotations on the attributes of the `PersonServiceTest`:

PersonServiceTest.java

```
@ExtendWith(MockitoExtension.class)
public class PersonServiceTest {

    @Mock
    private PersonRepository repository;

    @InjectMocks
    private PersonService service;
```

We have only one dependency, so we need only a single `@Mock` annotation on the attribute here. That attribute will be instantiated, the expectations on it will be set, and then it will be injected in the class under test, here annotated with `@InjectMocks`.

Whenever you use annotations, you need to tell Mockito to process them. JUnit 5 uses the Mockito JUnit 5 Extension, which is the argument to the `@ExtendWith` annotation on the test class. This extension initializes the mocks, and, as a side note, “handles strict stubbings.” That means in your tests you can mock only methods that your method under test actually calls.

You no longer have to instantiate the `PersonService` inside the test. Mockito does all that for you. You still have to set the expectations on the mock list, and you can verify the methods at the end if you want, but at least the creation of the

mock and its injection into the class under test is automated for you.

The test for `getLastNames` is now simplified:

PersonServiceTest.java

```
@Test
public void getLastNames_usingAnnotations() {
    when(repository.findAll()).thenReturn(people);

    assertThat(service.getLastNames())
        .contains("Borg", "Goldberg", "Hopper", "Liskov", "Lovelace");

    verify(repository).findAll();
}
```

This test uses the `repository` and `service` attributes rather than local variables of those types.

The process of instantiating the classes and wiring them together automatically is powerful but has some restrictions, which we'll talk about next.

Injecting the Mocks into the Class Under Test

One benefit of the annotation approach is that Mockito tries to inject the mocks into the class under test for you. How does it do that, exactly? Have a look at the details in the JavaDocs for the `@InjectMocks` annotation,[12] which says that Mockito attempts the following sequence of steps:

1. Calls the largest constructor on the test class if it takes the right types of arguments.
2. Calls setter methods of the proper type (and the proper name if there is more than one for a given type).
3. Sets the fields directly.

Suffice it to say Mockito will make a good effort to plug the mocks into the class under test, but, as the docs say, "Mockito is not a dependency injection framework; don't expect this shorthand utility to inject a complex graph of objects, be it mocks/spies or real objects."

Despite the disclaimer, that Mockito will try to do the work for you is a strong argument in favor of the annotation approach.

This talk of injection may remind you of the Spring framework,[13] which truly is a dependency injection framework. Spring is very friendly with Mockito. Spring not only includes the Mockito library as a testing dependency, it also provides its own annotation, called `@InjectBean`, to help Mockito. We'll look at Spring and Mockito in *Working with the Spring Framework*.

Regardless of whether you use the `when` method or the annotation approach to create the mocks, the next step is to set expectations on them. The tests so far demonstrated that, but only in its simplest case. Let's look at the process in more detail.

Setting Expectations

The Mockito-based example that mocked the `PersonRepository` dependency of `PersonService` used the methods `when` and `thenReturn` to set the expectations on the `findAll` method. Here's the signature for `when`:

```
public static <T> OngoingStubbing<T> when(T methodCall)
```

The argument to `when` is the declaration of an invocation of the method you want to call on the stub. The return type connects to the various `then` methods, like `thenReturn`, `thenThrow`, or `thenAnswer`, which are normally chained to the output. There are a couple of overloads of the `thenReturn` method:

```
OngoingStubbing<T> thenReturn(T value)
OngoingStubbing<T> thenReturn(T value, T... values)
```

For example, add another method to the `PersonService` called `findByIds`, which takes a vararg list of integer IDs and returns a `List<Person>`:

PersonService.java

```
public List<Person> findByIds(int... ids) {
    return Arrays.stream(ids)
        .mapToObj(repository::findById)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .collect(Collectors.toList());
}
```

To test this method on the *service*, you need to tell the dependent *repository* what to do when `findById(int)` is called. You can give those instructions in a verbose fashion, using the `thenReturn` version that takes a single argument, like this:

PersonServiceTest.java

```
@Test
void findByIds_explicitWhens() {
    when(repository.findById(0))
        .thenReturn(Optional.of(people.get(0)));
```

```
        when(repository.findById(1))
            .thenReturn(Optional.of(people.get(1)));
        when(repository.findById(2))
            .thenReturn(Optional.of(people.get(2)));
        when(repository.findById(3))
            .thenReturn(Optional.of(people.get(3)));
        when(repository.findById(4))
            .thenReturn(Optional.of(people.get(4)));
        when(repository.findById(5))
            .thenReturn(Optional.empty());

        List<Person> personList = service.findByIds(0, 1, 2, 3, 4, 5);
        assertThat(personList).containsExactlyElementsOf(people);
    }
```

That's rather a mouthful. Fortunately, we can simplify the test like this:

PersonServiceTest.java

```
    @Test
    void findByIds_thenReturnWithMultipleArgs() {
        when(repository.findById(anyInt())).thenReturn(
            Optional.of(people.get(0)),
            Optional.of(people.get(1)),
            Optional.of(people.get(2)),
            Optional.of(people.get(3)),
            Optional.of(people.get(4)),
            Optional.empty());

        List<Person> personList = service.findByIds(0, 1, 2, 3, 4, 5);
        assertThat(personList).containsExactlyElementsOf(people);
    }
```

This version uses the `anyInt` *argument matcher* as the parameter for `findById`. The `anyInt` method is a static method in the `ArgumentMatchers` (plural) class, which represents, naturally enough, any integer. We'll look at argument matchers in detail in Chapter 3, *Use Built-in and Custom Matchers*.

The interesting part is the `thenReturn` method, which this time takes a series of arguments. The first time `findById` is called, the first person is returned (wrapped in an `Optional`). The second invocation returns the second person in that collection, and so on. If the method is called more than five times, an empty

`Optional` is returned on that call and all subsequent calls.

Though it wasn't needed here, you can mix and match calls to `thenReturn`, `thenThrows`, and `thenAnswer`. The first call could return an instance, the second one could throw an exception, the third could return an instance again, and so on.

For example, the following code presumes that the class under test invokes the `findById` method on the `PersonRepository` multiple times:

PersonServiceTest.java

```
@Test
void testMultipleCalls() {
    when(repository.findById(anyInt()))
        .thenReturn(Optional.of(people.get(0)))
        .thenThrow(new IllegalArgumentException("Person with id not found"))
        .thenReturn(Optional.of(get(1)))
        .thenReturn(Optional.empty));

    // .. rest of test ...
}
```

The first call to `findById` returns an `Optional` containing the first person, the second call throws an `IllegalArgumentException`, the third call (assuming the exception was caught and the code continues) returns the second person, and the fourth and all subsequent calls return an empty `Optional`.

After returning, you can take advantage of one more capability of Mockito: verifying that the methods on the mocks were called the right number of times, in the right order.

Verifying the Multiplicity

The static `verify` method in Mockito is overloaded to take an optional second argument. The full signature of the method is this:

```
public static <T> T verify(T mock,
                           org.mockito.verification.VerificationMode mode)
```

The first argument is the mocked object. The second argument is an interface

that you rarely need to implement because it's already generated by the many factory methods. Those are static methods in the Mockito class, listed here:

- times(int)
- never()
- atLeastOnce()
- atLeast(int)
- atMostOnce()
- atMost(int)

The default, naturally enough, is times(1), so if you call verify with just one argument, the check is that the mocked method was invoked exactly once.

Note that if you wrote your own mocks, you wouldn't be able to do this. Java doesn't provide an easy way to keep track of how many times methods were called. By doing so, Mockito allows you to verify the *protocol*, which is the interaction between the class under test and its dependencies.

No Further Interactions

Use the never method to verify that a particular method is never invoked. More generally, Mockito has a method called verifyNoMoreInteractions(mock), which checks that no methods other than the ones explicitly stated were invoked at all.

That method sounds powerful, but the JavaDocs contain a caution:

A word of warning: Some users who did a lot of classic, expect-run-verify mocking tend to use verifyNoMoreInteractions() very often, even in every test method. verifyNoMoreInteractions is not recommended to use in every test method. verifyNoMoreInteractions() is a handy assertion from the interaction testing toolkit. Use it only when it's relevant (for example, regression testing of legacy code). Abusing it leads to over-specified, less maintainable tests.

The when/thenReturn/thenThrow/thenAnswer pattern works well, but you can't use it

in methods that return `void` (that is, nothing); these require special handling. Fortunately, while the recommended syntax looks different, using it is just as easy.

Mocking Methods That Return void

You need to know one quirk of the Mockito syntax. When you try to set expectations on a method that returns `void`, you can't use the `when`/`then` syntax.

In the `PersonRepository`, the `delete` method returns `void`. Say we were testing failure modes and we decided that if the `PersonService` calls `delete` with a `null` argument (which hopefully will never happen), the repository should throw a `RuntimeException`.

The way we would normally set an expectation like that would be to write (*Note: this does not compile.*):

```
when(repository.delete(null)).thenThrow(RuntimeException.class);
```

This code doesn't work, with the error message "no instance(s) of type variable(s) T exist so that void conforms to T". That's a rather confusing message, but fortunately there's a simple fix. Whenever you need to mock a method that returns `void`, use one of the various `do` methods first:

```
doThrow(RuntimeException.class).when(repository).delete(null);
```

As an example, here's the test for `deleteAll`, using a `List<Person>` that contains only a `null`:

PersonServiceTest.java

```
@Test
public void deleteAllWithNulls() {
    // Set up findAll to return a list containing nulls of type Person
    when(repository.findAll()).thenReturn(
        Arrays.asList((Person) null));

    // This won't compile:
    // when(repository.delete(null)).thenThrow(RuntimeException.class);
```

```
        // But this will:
        doThrow(RuntimeException.class).when(repository).delete(null);

        assertThrows(RuntimeException.class, () -> service.deleteAll());

        verify(repository).delete(null);
    }
```

Sometimes using the `do` methods feels like you're writing the expression backward, but you get used to it. For the record, here is the complete list of `do` methods:

- `doReturn`
- `doThrow`
- `doAnswer`
- `doNothing`
- `doCallRealMethod`

You can use these anywhere, but most people use them only for methods that return `void`.

Using Mocks and Stubs in the Astro Project

In *Counting Astronauts by Spaceship*, we introduced a project that processed the number of astronauts in space and returned how many were aboard each spacecraft. The examples there showed an integration test for the `AstroService` class and how to create a unit test using our own `FakeGateway`. Now we can use Mockito instead to mock the `Gateway<T>` interface.

As a reminder, `AstroService` retrieves the data from a `Gateway<T>`, reproduced here:

Gateway.java

```
public interface Gateway<T> {
    T getResponse();
}
```

The class that implements this interface is called `AstroGateway`, and it returns an `AstroResponse` that could possibly be null. If we're going to isolate the `AstroService`, we need to ask Mockito to return either that `AstroResponse` or throw a `RuntimeException` if it's null.

Let's use JUnit 5 along with the Mockito annotations. The test class for the `AstroService` now starts with this:

AstroServiceTest.java

```
@ExtendWith(MockitoExtension.class)
class AstroServiceTest {

    @Mock
    private Gateway<AstroResponse> gateway;

    @InjectMocks
    private AstroService service;
```

We'll rely on Mockito to create the mock of the gateway and then inject it into the service because we've provided a constructor to the `AstroService` class that takes a `Gateway` as an argument.

One test then looks like this:

AstroServiceTest.java

```
@Test
void testAstroData_usingInjectedMockGateway() {
    // Mock Gateway created and injected into AstroService using
    //    @Mock and @InjectMock annotations
    //
    // Set the expectations on the mock
    when(gateway.getResponse())
        .thenReturn(mockAstroResponse);

    // Call the method under test
    Map<String, Long> astroData = service.getAstroData();

    // Check the results from the method under test
    assertThat(astroData)
        .containsEntry("Babylon 5", 2L)
        .containsEntry("Nostromo", 1L)
        .containsEntry("USS Cerritos", 4L);
    astroData.forEach((craft, number) -> {
        System.out.println(number + " astronauts aboard " + craft);
        assertAll(
            () -> assertThat(number).isPositive(),
            () -> assertThat(craft).isNotBlank()
        );
    });

    // Verify the stubbed method was called
    verify(gateway).getResponse();
}
```

The annotations take care of two of our steps: creating the mock and injecting into the service. We still need to set the expectations, which we can do with the `when/thenReturn` pair. After calling the method under test `getAstroData` in `AstroService`, we can then verify that the results are correct. At the end, we can optionally check that the service did indeed invoke the `getResponse` method on the gateway exactly once.

Testing for a failure is similar and uses the `thenThrow` method:

AstroServiceTest.java

```
// Check network failure
@Test
void testAstroData_usingFailedGateway() {
    when(gateway.getResponse()).thenThrow(
        new RuntimeException(new IOException("Network problems")));

    assertThatExceptionOfType(RuntimeException.class)
        .isThrownBy(() -> service.getAstroData())
        .withCauseInstanceOf(IOException.class)
        .withMessageContaining("Network problems");
}
```

Since the `getResponse` method on the gateway returns a `Failure` that wraps the exception, we don't have to use the `doThrow/when` construct, which is required when the method we're mocking returns `void`.

Another advantage of using Mockito is you can test for the case of network failure without disabling the network. I bet the astronauts would be happy about that.

Congratulations. You've successfully used Mockito to mock the `Gateway`, and that concludes round one of the Mockito API game.

Wrapping Up

You can either create your mocks manually and insert them into your class under test or you can use Mockito's annotation support to inject them automatically. Mockito doesn't provide a full injection framework, like Spring or Guice, but it does make a good faith effort to plug in your mocks via either constructor injection, setter injection, or even field injection.

You now have the mechanisms for telling your mocks (or stubs, in this case) what to return when methods are called. Chained methods made the process much simpler, and you can use multiple arguments to respond to each invocation with different results. You also now know about the special handling required to mock methods that return `void`.

In the next chapter, we'll talk about generalizing the arguments to mocked methods so you don't have to give explicit values every time. Mockito has a class that contains factory methods for that purpose—we'll also use Java lambdas to implement custom argument matchers.

Footnotes

[12] https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/InjectMocks.xhtml

[13] https://spring.io

Chapter 3

Use Built-in and Custom Matchers

I remember long ago, when newspapers (kids, ask your parents) had comics in them. One day I read a *Peanuts* comic where Charlie Brown was trying to teach his younger sister Sally how to count. He showed her a picture of a harbor and asked her, “How many ships do you see?”

She replied, “All of them.”

She would have made an excellent software developer. Her answer, while useless, would have passed all the basic unit tests.

The Mockito API lets you specify individual outputs for specified inputs, but it also has a lot of generalized methods called *argument matchers*. With those, you can say, “for any int” or “for any String”, rather than just “3” or “abc”. You can even say “for any Ship” if you want to, and if you write your own custom matcher, you can say “for any Ship in this Picture”.

In this chapter, we’ll look at the many methods in the `ArgumentMatchers` (plural) class, which are static factory methods to produce objects that implement the `ArgumentMatcher` (singular) interface. We’ll also see how to implement our own argument matcher using lambda expressions.

Using the Existing Argument Matchers

The `ProjectService` class from Chapter 2, *Work with the Mockito API*, contains a method called `findByIds(int...)`. That method in turn calls the `findById(int)` method on the `PersonRepository`, which we needed to mock to isolate our `PersonService` for testing. Here's a test of the service method for the case where the requested ID doesn't exist in the database:

PersonServiceTest.java

```
@Test
public void findByIdThatDoesNotExist() {
    when(repository.findById(anyInt()))
        .thenReturn(Optional.empty());

    List<Person> personList = service.findByIds(999);
    assertTrue(personList.isEmpty());

    verify(repository).findById(anyInt());
}
```

The `anyInt` method was used both when setting the expectations and when verifying them. The method `anyInt` is a static method in the `ArgumentMatchers` class, which represents, as you might expect, any integer. Like so many of the methods you use in Mockito on a regular basis, the method is static, so you'll use a static import to access it. That's why the syntax used in most tests is simply `anyInt()` rather than `ArgumentMatchers.anyInt()`.

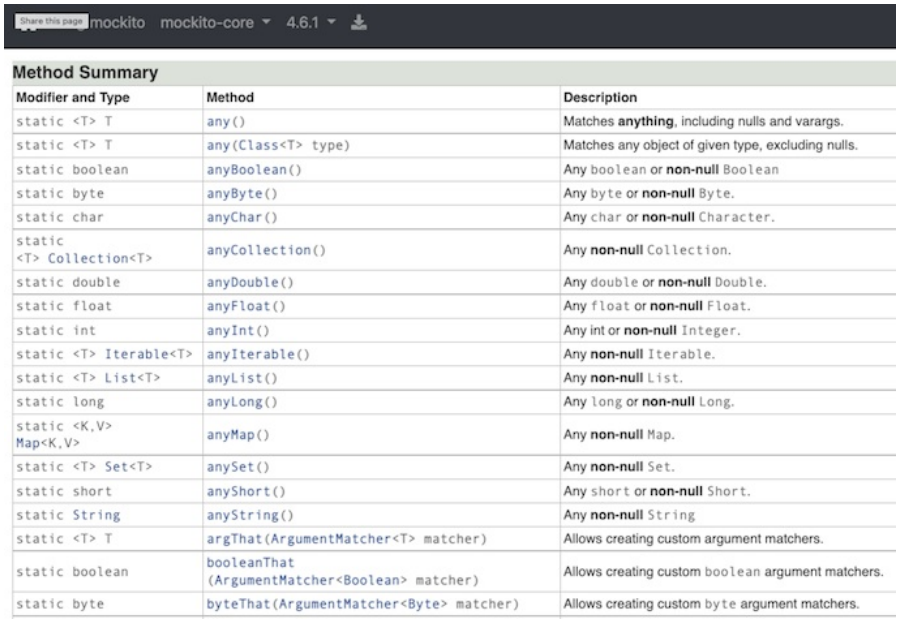
Where Is the Static Import of ArgumentMatchers?

The `ArgumentMatchers` class contains a wide range of useful static methods that return argument matchers. Each of the returned values implements the `ArgumentMatcher` interface. Generally, you'll find it easier to use static methods before implementing your own

custom matchers.

If you look at tests, however, you'll notice the only static import statement is for the `Mockito` class itself. It turns out that the `Mockito` class extends the `ArgumentMatchers` (plural) class, so the static import of `Mockito` covers both. It's an unusual usage of inheritance, but it does make writing the tests simpler.

Here's a snapshot of part of the `ArgumentMatchers` methods in the JavaDocs:



mockito mockito-core 4.6.1

Method Summary

Modifier and Type	Method	Description
static <T> T	any()	Matches **anything**, including nulls and varargs.
static <T> T	any(Class<T> type)	Matches any object of given type, excluding nulls.
static boolean	anyBoolean()	Any boolean or **non-null** Boolean
static byte	anyByte()	Any byte or **non-null** Byte.
static char	anyChar()	Any char or **non-null** Character.
static <T> Collection<T>	anyCollection()	Any **non-null** Collection.
static double	anyDouble()	Any double or **non-null** Double.
static float	anyFloat()	Any float or **non-null** Float.
static int	anyInt()	Any int or **non-null** Integer.
static <T> Iterable<T>	anyIterable()	Any **non-null** Iterable.
static <T> List<T>	anyList()	Any **non-null** List.
static long	anyLong()	Any long or **non-null** Long.
static <K,V> Map<K,V>	anyMap()	Any **non-null** Map.
static <T> Set<T>	anySet()	Any **non-null** Set.
static short	anyShort()	Any short or **non-null** Short.
static String	anyString()	Any **non-null** String
static <T> T	argThat(ArgumentMatcher<T> matcher)	Allows creating custom argument matchers.
static boolean	booleanThat (ArgumentMatcher<Boolean> matcher)	Allows creating custom boolean argument matchers.
static byte	byteThat(ArgumentMatcher<Byte> matcher)	Allows creating custom byte argument matchers.

Most of the static methods in the `ArgumentMatchers` class are intuitive, and though there are over forty of them, they fall into a handful of categories:

- Methods that match any instance of primitive types: `anyByte`, `anyShort`, `anyInt`, `anyLong`, `anyFloat`, `anyDouble`, `anyChar`, and `anyBoolean`. These methods work exactly the way you would expect.
- Methods that match collections, like `anyCollection`, `anyList`, `anySet`, and `anyMap`. These methods match any non-null collection of the specified type.
- Methods that work on strings, like `anyString`, `startsWith`, `endsWith`, and the two overloads of `matches`, one that takes a regular expression as a string and the other a `Pattern`.

- General-purpose matchers for nulls, like `isNull` and `isNotNull` (and its companion, `notNull`, which is just an alias), and `nullable(Class)`, which matches either null or a given type.
- For types, the matcher `isA(Class)` matches any argument that is assignable to a reference of the specified class. That leads to `any(Class)`, which matches any instance of the specified type, excluding nulls, and the ultimate matcher, `any`, which matches anything, including nulls and varargs.

The class includes one set of overloaded matchers called `eq`, which may seem puzzling. In the next section, we'll look at those methods and the reason they exist, uncovering an easy trap to fall into when verifying mocks.

Using the eq Matchers

The `ArgumentMatchers` class has one category of methods you might not expect: a whole series of `eq` methods, overloaded for all eight primitive types, as well as `eq(T)`, which matches any instance of class `T`. Why would you need those methods? To avoid a subtle trap.

If you're stubbing a method that takes more than one argument and you use an argument matcher for one of the arguments, *you have to use argument matchers for all of them*.

The docs are very clear about this requirement and include the following example. Say you want to verify that `someMethod` is called on a mock, and `someMethod` takes three arguments: an integer, a string, and another string. If you know what you want the last string to be, you might think you can just provide it, as in the following (*Note: this example does not work.*):

```
verify(mock).someMethod(anyInt(), anyString(), "third argument");
```

This call will fail because the first two arguments use argument matchers, but the last doesn't. That's where the `eq` matchers come in. If you use one argument matcher, you have to use argument matchers for all the arguments. Using `eq`, you can simply rewrite the call as:

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
```

Now everything works. Missing an argument matcher is an easy trap to fall into because the compiler doesn't identify it as an error. The code won't fail until you run the actual tests. Fortunately, it's an easy fix.

Going beyond the basic `ArgumentMatchers`, Mockito includes another class, called `AdditionalMatchers`, which includes methods like `and`, `or`, and `not`, for combining values, as well as a comparison relationships, like `lt`, `leq`, `gt`, `geq`, and various forms of array equality checks. See the JavaDocs for `AdditionalMatchers`[14] for details.

Now let's look at the last category of methods in `ArgumentMatchers`—methods that allow you to create custom matchers.

Creating Custom Argument Matchers

To create your own matcher, the `ArgumentMatchers` class has a method called `argThat`, which takes an `ArgumentMatcher` (singular) as an argument. `ArgumentMatcher` is an interface containing a single abstract method, with this signature:

```
boolean matches(T argument)
```

This method should return `true` if the argument matches whatever condition you implement, and `false` otherwise.

In the dark days before lambda expressions were added in Java 8, implementing your own matcher wasn't exactly difficult, but it was verbose. The example given in the documentation is hardly inspiring (comments added):

ListCustomMatcher.java

```
// custom matcher that implements the ArgumentMatcher interface
class ListOfTwoElements implements ArgumentMatcher<List> {
    public boolean matches(List list) {
        return list.size() == 2;
    }

    public String toString() {
        //printed in verification errors
        return "[list of 2 elements]";
    }
}

    // create the mock
    List mock = mock(List.class);

    // set the expectation using argThat and the custom matcher
    when(mock.addAll(argThat(new ListOfTwoElements()))).thenReturn(true);

// somewhere in the actual test, test a method that invokes addAll
// with a two-element list:
    mock.addAll(Arrays.asList("one","two"));
```

```
    // verify that the test called addAll with the custom matcher
        verify(mock).addAll(argThat(new ListOfTwoElements()));
```

That's a fair amount of work just to see if the mock had one of its methods invoked with a list of size two. Fortunately, with lambda expressions you can write this test easily. Simply provide a lambda expression that takes a single argument of the proper type and returns a Boolean. You can eliminate the class in the example and write the following to set the expectation:

```
verify(mock).addAll(argThat(list -> list.size() == 2));
```

Easy peasy, lemon squeezy. By removing the friction of implementing your own class and instantiating it both when setting the expectations and when verifying them, the whole process becomes much easier.

A working example of using lambdas to implement custom matchers is shown in *Testing void Methods Using Interactions*. Here, though, is an example from the `PersonService` tests, which leads to the next subtle (but easy to fix) trap. We want to check for IDs that do not exist. Since we know all the IDs in the sample data set are less than 14, we could use a custom matcher like this:

PersonServiceTest.java

```
@Test
@Disabled("Do not use argThat with integers")
public void findByIdsThatDoNotExist_argThat() {
    when(repository.findById(argThat(id -> id > 14)))
        .thenReturn(Optional.empty());

    List<Person> personList = service.findByIds(15, 42, 78, 999);
    assertTrue(personList.isEmpty());

    verify(repository, times(4)).findById(anyInt());
}
```

This leads to a problem, however. The test compiles, but we've fallen into another trap, which involves `argThat` with primitive types. Fortunately this, too, is easy to fix.

Using Argument Matchers for Primitive Types

Invoking the `findById` method on `PersonRepository` returns an empty `Optional` whenever the requested ID was greater than 14. That was expressed using a custom matcher implemented using a lambda expression:

```
when(repository.findById(argThat(id -> id > 14))
        .thenReturn(Optional.empty());
```

That code looks simple enough, and the lambda expression is correct. Unfortunately, when you run the test, you get a `NullPointerException`.

Why would the return value from the `argThat` method be null? Even weirder, if you look at the JavaDocs for the `argThat`[15] method, even though the signature claims it returns `T` (the class), the docs say it's supposed to return null, so again, why is this a problem?

The answer lies in this warning for `argThat`:

> *NullPointerException auto-unboxing caveat*. In rare cases when matching primitive parameter types you *must* use relevant intThat(), floatThat(), etc. method. This way you will avoid NullPointerException during auto-unboxing. Due to how Java works we don't really have a clean way of detecting this scenario and protecting the user from this problem. Hopefully, the JavaDoc describes the problem and solution well. If you have an idea how to fix the problem, let us know via the mailing list or the issue tracker.

In other words, the problem is related to unboxing wrapper classes into primitive types, and the docs claim this isn't easily fixable. There is, however, an easy fix. When dealing with primitives, instead of calling the `argThat` method, call one of its primitive variations: `byteThat`, `shortThat`, `charThat`, `intThat`, `longThat`, `floatThat`, `doubleThat`, and `booleanThat`.

So the right way to add the custom matcher in this case is simply to use `intThat`:

PersonServiceTest.java

```
@Test
```

```
public void findByIdsThatDoNotExist_intThat() {
    // Custom matcher as lambda argument to intThat:
    when(repository.findById(intThat(id -> id > 14)))
        .thenReturn(Optional.empty());

    List<Person> personList = service.findByIds(15, 42, 78, 999);
    assertTrue(personList.isEmpty());

    verify(repository, times(4)).findById(anyInt());
}
```

Now everything works. That list of methods based on the primitive types is the last category of methods in the `ArgumentMatchers` class.

Custom Matchers for Primitives

When using a custom argument matcher that is based on primitive types, use the methods designed for that purpose: `byteThat`, `shortThat`, `intThat`, `longThat`, `floatThat`, `doubleThat`, instead of `argThat`. That approach will avoid potential null pointer exceptions due to unboxing.

We've now covered all the important methods in the `ArgumentMatchers` class, including how to implement your own custom argument matcher. We can now verify that methods are called with the proper arguments. But to verify the protocol, we also want to check that the methods on the mocked objects are called in the proper order. Fortunately, Mockito makes that easy as well, as you'll see next.

Verifying the Order of Methods Called

One last check we can make is to verify that a particular method on one stub was invoked before another method on a different stub. We'll need an additional class in Mockito to do that, called `org.mockito.InOrder`, and we'll create an instance of it using another static method from the `Mockito` class, also called `inOrder` (but with a lowercase first letter).

Here's the signature of the `inOrder` method:

```
public static InOrder inOrder(Object... mocks)
```

The `inOrder` method takes a vararg list of mocks, so to use it you create the mocks first. It's interesting that you can check that different methods were called on different mocks in the order specified. So if the object we're testing had multiple dependencies and we mock them, we can verify that methods are called in the proper order even across different mock objects.

The example in the docs makes this clearer:

```
InOrder inOrder = inOrder(firstMock, secondMock);

inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");
```

This code checks that the `add` method was called on the first mock with the string argument "was called first", and then the `add` method was called on the second mock with argument "was called second". You have to explicitly list the calls in order, but they work.

In Chapter 1, *Build a Testing Foundation*, we looked at a class called `HelloMockito`, which contained two dependencies: one for the `PersonRepository` we've been using throughout the current chapter and one on a `TranslationService` that translated the greeting message into whatever language we wanted. We can now check that the appropriate methods in those dependencies are not only invoked the proper number of times (once each), but also in the proper order, as

shown in this excerpt from the `greetAPersonThatExists` test:

HelloMockitoTestFull.java

```
@Test
@DisplayName("Greet Admiral Hopper")
void greetAPersonThatExists() {
    // set the expectations on the mocks
    when(repository.findById(anyInt()))
        .thenReturn(Optional.of(new Person(1, "Grace", "Hopper",
            LocalDate.of(1906, Month.DECEMBER, 9))));
    when(translationService
        .translate("Hello, Grace, from Mockito!", "en", "en"))
        .thenReturn("Hello, Grace, from Mockito!");

    // test the greet method
    String greeting = helloMockito.greet(1, "en", "en");
    assertEquals("Hello, Grace, from Mockito!", greeting);

    // verify the methods are called once, in the right order
    InOrder inOrder = inOrder(repository, translationService);
    inOrder.verify(repository).findById(anyInt());
    inOrder.verify(translationService)
        .translate(anyString(), eq("en"), eq("en"));
}
```

In that test, the mocks were created using the `@Mock` annotation and injected into the class under test using the `@InjectMocks` annotation. We set the expectations on the mock, using the `anyInt` argument matcher, tested the `greet` method, and then used the `InOrder` class to not only `verify` the methods were called but confirmed that `findById` was called on the mock repository first, followed by `translate` on the mock translation service. Note also the use of the `eq` matcher for the last two `String` arguments in the `translate` method.

In other words, now you know all the features of Mockito shown in that test, from annotations to setting expectations to verifying method order. Hopefully it all makes sense now.

Wrapping Up

In this chapter, we covered the topic of argument matchers. We use argument matchers to simplify both setting expectations on mock objects and verifying that the expected methods were called the right number of times in the right order.

The `Mockito` class extends the `ArgumentMatchers` class, which contains a large number of factory methods, like `anyInt`, to provide most of the argument matchers we need. It also includes the method `argThat`, which takes a custom `ArgumentMatcher` that we can provide as a lambda expression.

Along the way we identified two potential issues that you need to keep in mind. One is that if you use an argument matcher for one argument of a mocked method, you need to use matchers for all of them—thus the `eq` methods in the `ArgumentMatchers` class. Also, if you are customizing a matcher that involves primitives, you should use their associated methods, like `intThat` or `longThat`, instead of `argThat`, to avoid nullability problems associated with unboxing.

Finally, we looked at the `inOrder` method, which allows you to verify that mocked methods were called in the proper order, even across different mocked dependencies. So our mocks now satisfy the full protocol: the mocked methods not only return the proper values, as stubs would, but also can check that the methods are called the proper number of times, in the expected order.

You can do a lot with Mockito using just the capabilities we've covered so far. In the next chapter, however, we'll move beyond the basics and talk about mocking methods that return `void`, capturing the arguments used when mocked methods are called, and using spies to verify the protocol of legacy methods.

Footnotes

[14] https://www.javadoc.io/doc/org.mockito/mockito-core/2.2.7/org/mockito/AdditionalMatchers.xhtml

[15] https://www.javadoc.io/doc/org.mockito/mockito-

core/2.7.13/org/mockito/hamcrest/MockitoHamcrest.xhtml

Chapter 4

Solve Problems with Mockito

I've always preferred chess to poker, partly because I have a terrible poker face. It's hard for me to keep my emotions out of my expressions, and while that's a good idea in chess, it's vital in poker, where the other players are constantly trying to judge whether you have a good hand or not and adjusting their bets accordingly. Ideally, when the other players look at me, I should be returning *nothing* to make the game difficult for them.

Nothingness can also cause difficulties when testing. In Java, all too often developers write methods that return nothing. Those methods are hard to test, because, after all, nothing is coming back. Sometimes you also have to mock a method that returns nothing, like telling it to throw an exception for certain inputs. In *Mocking Methods That Return void*, we discussed how to *mock* methods that return nothing but not how to *test* methods that return nothing. That's one of the hard problems that Mockito solves easily. In this chapter, we'll tackle a variety of problems that Mockito can help you solve:

- Testing methods that return `void`.
- *Capturing* the actual arguments of mocked methods.
- Using *custom answers* to return values based on invoked arguments.
- *Spying* on real objects.

We'll also look at how Mockito supports the concept of Behavior Driven Development (BDD).[16] Then we'll look at testing those `void` methods. After that, we'll see how to capture the actual arguments used when a method on a mock is invoked. Then we'll see how to make the mock return a value based on

its input argument. Finally, we'll spy on some real objects.

I should mention that I view my lack of a poker face to be more of a feature than a bug. Since I know I can't hide how good or bad my hand is, I don't gamble much, and when I do, it's for very small stakes. Still, it would be nice to be able to bluff once in a while.

Deciding Between Mockito and BDDMockito

Mockito contains two separate APIs with identical functionality. Normally that's considered a "code smell," but there's a reason for both, and you'll encounter both APIs in practice.

So far when using Mockito, we've started with static methods from the `Mockito` class. You'll find a completely parallel set of methods in a subclass of `Mockito` called `org.mockito.BDDMockito`. The developers added the `BDDMockito` class to support BDD.

We've been using methods like the `when/thenReturn` pair to set expectations and the `verify` method to check that methods on the stubs were called. The `BDDMockito` class has a different API to accomplish the same tasks. BDD emphasizes writing tests using *given*, *when*, and *then* to represents the setup of the test, invoking the test method, and checking its results. You can probably see the problem immediately: the `Mockito` class already has a method called `when` that's not designed to fit the BDD model.

Since the developers of Mockito wanted to support the BDD approach, they added the `BDDMockito` class. In that class, setting the expectations begins with the method called `given`, followed by variations on `when` to set the return value. This is the signature for the `given` method in `BDDMockito`:

```
public static <T> BDDMyOngoingStubbing<T> given(T methodCall)
```

In the regular `Mockito` class, the `when` method is followed in a chain by one of `thenReturn`, `thenThrow`, or other methods. In `BDDMockito`, the `given` method is followed by `willReturn`, `willThrow`, or others in the same pattern.

For example, here's a test of the `getMaxId` method of the `PersonService`, which requires us to mock the `findAll` method of the `PersonRepository`:

PersonServiceTest.java

```
@Test
public void findMaxId() {
    when(repository.findAll()).thenReturn(people);
    assertThat(service.getHighestId()).isEqualTo(14);
    verify(repository).findAll();
}
```

If this is written using the corresponding methods in the BDDMockito class, the test looks like this:

PersonServiceTest.java

```
@Test
public void findMaxId_BDD() {
    given(repository.findAll()).willReturn(people);
    assertThat(service.getHighestId()).isEqualTo(14);
    then(repository).should().findAll();
}
```

Setting the expectation simply replaces when with given and thenReturn with willReturn. The verification step is slightly more complicated, because in addition to replacing verify with then, BDDMockito requires you to add a should method call before specifying the mocked method that was invoked.

Why is the should method required? The verify method from Mockito is overloaded to specify the multiplicity:

```
verify(repository, times(1)).findAll()
```

Calling the findAll method only once is the default, so you can leave it out, as in the test. The then method in BDDMockito doesn't have an overloaded version, however. To specify the multiplicity, you put the times argument inside should:

```
then(repository).should(times(1)).findAll();
```

It seems odd to be forced to include should even when the multiplicity is exactly 1 (the default), but it does read better to a native English speaker.

The rest of the BDDMockito class is simple search-and-replace variations on the Mockito methods. You can also still use argument matchers, including lambda

expressions as custom matchers, in the usual way.

For methods that return `void` (discussed in *Testing void Methods Using Interactions*, Mockito requires you to start with one of the `do` methods, like `doReturn` or `doThrow`. In `BDDMockito`, the similar methods begin with `will`, as in `willReturn` or `willThrow`, as shown in the example from the docs:

```
//given
willThrow(new RuntimeException("boo")).given(mock).foo();

//when
Result result = systemUnderTest.perform();

//then
assertEquals(failure, result);
```

Note the usage of the added "given/when/then" comments: the *given* section is for setting the expectations, the *when* section invokes the method under test, and the *then* section checks the return value from the test method. No verification is done in this example.

Which API you decide to use, BDD or traditional, is purely a matter of style. But be aware that you'll encounter both in practice.

Next up, let's look at testing (as opposed to mocking) methods that return `void`, which provides a nice demonstration of why mocks can be so helpful.

Testing void Methods Using Interactions

As developers adopt a more test-driven style, they learn to avoid writing methods that return nothing. It's hard to test methods that return `void`. After all, if nothing is coming back, how do you know those methods worked at all? If you're lucky, the methods change the state of the object (that is, its contained attributes) and you can check them, but what if that doesn't happen either? Mockito can help by letting you check the interactions between the class you're testing and its mocked dependencies.

Let's introduce a new example involving a popular architecture these days: a publisher/subscriber system, where the publisher sends signals to subscribers that receive them. Publishers and subscribers are often used in asynchronous systems, but here we'll use them as an example of classes that don't return values.

(As an aside, this example was adapted from a similar one used to demonstrate the Spock[17] testing framework.)

Here's a simple `Publisher` class:

Publisher.java

```
package com.kousenit.pubsub;

import java.util.ArrayList;
import java.util.List;

// Adapted from a similar example in the Spock framework
public class Publisher {
    private final List<Subscriber> subscribers = new ArrayList<>();

    public void addSubscriber(Subscriber sub) {
        subscribers.add(sub);
    }

    // Want to test this method
    public void send(String message) {
```

```
            for (Subscriber sub : subscribers) {
                try {
                    sub.receive(message);
                } catch (Exception ignored) {
                    // evil, but what can you do?
                }
            }
        }
    }
```

A `Publisher` maintains a list of `Subscriber` instances. When the `send` method is called on the `Publisher`, it calls the `receive` method on each subscriber, with a string message argument. The challenge will be to test the `send` method, which returns nothing. Also note that invoking `send` doesn't change anything in the publisher, whose only attribute is the list of subscribers. To make matters worse, if any of the subscribers throws an exception, the publisher catches and ignores it, which is guaranteed to come up on a code review but won't help us now. The motivation is that even if one or more of the subscribers throws an exception, we still want the publisher to reach the rest of them. Still, it would no doubt be better to log those exceptions somewhere, but that might give us something for us to test, and we're trying to see what to do when nothing is available.

The `Subscriber` here will simply be an interface:

Subscriber.java

```
package com.kousenit.pubsub;

public interface Subscriber {
    void receive(String message);
}
```

The `receive` method returns `void` as well.

How to you test the `send` method in the `Publisher`? Since the job of the `send` method is to invoke a method on each subscriber, you can use Mockito to mock the subscribers and verify that each of them had their `receive` method called with the proper argument.

To do this, we'll create an instance of `Publisher` and add two mock subscribers to it. In this case, the annotation approach (using `@Mock` and `@InjectMocks`) isn't helpful, because the publisher doesn't have a constructor or setter method for adding subscribers, and its only attribute is a collection of them. This is a case where the simple `mock` method works instead of the annotations, even though that means we'll have to instantiate the classes and inject the dependencies ourselves.

Here's the start of the `PublisherTest` class:

PublisherTest.java

```
class PublisherTest {
    private final Publisher pub = new Publisher();
    private final Subscriber sub1 = mock(Subscriber.class);
    private final Subscriber sub2 = mock(Subscriber.class);

    @BeforeEach
    void setUp() {
        pub.addSubscriber(sub1);
        pub.addSubscriber(sub2);
    }
```

The test creates a publisher and two mock subscribers and adds the subscribers to the publisher in the `setUp` method.

The first test checks the `send` method by verifying the subscribers:

PublisherTest.java

```
@Test
void publisherSendsMessageToAllSubscribers() {
    pub.send("Hello");

    verify(sub1).receive("Hello");
    verify(sub2).receive("Hello");
}
```

Even though nothing is coming back from `send`, and the `receive` method in the subscribers doesn't return anything, we know the publisher is reaching all the

subscribers. We just tested a method that returned `void`, that invoked methods that also returned `void`. Pretty slick.

If we want, we can even check that the subscribers received the message in the order they were added:

PublisherTest.java

```
@Test
void testSendInOrder() {
    pub.send("Hello");

    InOrder inorder = inOrder(sub1, sub2);
    inorder.verify(sub1).receive("Hello");
    inorder.verify(sub2).receive("Hello");
}
```

That may or may not be necessary, depending on whether the ordering of subscribers is important, but it's easy enough to do.

If the message string has a pattern, you can use argument matchers to check it. In this test the existing `anyString` argument matcher is used, which may be sufficient, or you can use a custom matcher as shown below that.

PublisherTest.java

```
@Test
void publisherSendsMessageWithAPattern() {
    pub.send("Message 1");
    pub.send("Message 2");

    // Check for any string
    verify(sub1, times(2)).receive(anyString());
    verify(sub2, times(2)).receive(anyString());

    // Alternatively, check for a specific pattern
    verify(sub1, times(2)).receive(
        argThat(s -> s.matches("Message \\d")));
    verify(sub1, times(2)).receive(
        argThat(s -> s.matches("Message \\d")));
}
```

You wouldn’t do both in practice (checking for any string and for strings with a specific pattern), but the example shows how easy it is to create your own custom matcher using lambda expressions.

What about a misbehaving subscriber that throws an exception? Since the publisher is catching and ignoring all exceptions, all subscribers will still receive all the messages.

PublisherTest.java

```
@Test
void handleMisbehavingSubscribers() {
    // sub1 throws an exception
    doThrow(RuntimeException.class).when(sub1).receive(anyString());

    pub.send("message 1");
    pub.send("message 2");

    // both subscribers still received the messages
    verify(sub1, times(2)).receive(anyString());
    verify(sub2, times(2)).receive(anyString());
}
```

Methods That Return void

Recall that for methods that return `void`, you have to use the `do` methods (`doThrow`, `doNothing`, and so on) when setting expectations. That’s why `doThrow` was used here when telling the `receive` method (which returns `void`) to throw a `RuntimeException`.

The bottom line is that Mockito made it easy to verify that the `send` method is doing its job, even though neither it nor the `receive` method in `Subscriber` is returning anything.

In the next two sections, we’ll see how to deal with two more challenges when using Mockito: capturing the actual arguments used when a mocked method is called and how to use a custom `Answer` to get a method to return a value based on its supplied arguments.

Capturing Arguments

Returning to the `PersonService` and its dependency of type `PersonRepository`, let's add a method to the service that creates a `Person` from individual arguments.

PersonService.java

```
public Person createPerson(int id, String first,
                           String last, LocalDate dob) {
    Person person = new Person(id, first, last, dob);
    return repository.save(person);
}
```

In this variation we instantiate a `Person` as a local variable before invoking the `save` method on the repository. In some cases, the input arguments can be modified before instantiating the `Person`. Here, the date of birth argument is a `String` that needs to be parsed to call the `Person` constructor. We would like to *capture* the instantiated `Person` instance used as the argument to `save`.

To do this, add an `@Captor` annotation to an instance of `Person` in the test class, and then, in the verification step, use the `capture` method to capture it and the `getValue` method to get the actual value.

The resulting test looks like this:

PersonServiceTest.java

```
@Captor
private ArgumentCaptor<Person> personArg;

@Test
public void createPersonUsingDateString() {
    Person hopper = people.get(0);
    when(repository.save(hopper)).thenReturn(hopper);
    Person actual = service.createPerson(1, "Grace", "Hopper", "1906-12-09");

    verify(repository).save(personArg.capture());
    assertThat(personArg.getValue()).isEqualTo(hopper);
    assertThat(actual).isEqualTo(hopper);
```

```
}
```

The implementation of `createPerson` instantiates a `Person` from those arguments as a local variable. Since that variable is used as the argument to a mocked method (`save`), we can capture it, as shown. Then we can extract the `Person` instance from it using `getValue`, and we're done.

Argument captors allow you to find out exactly what arguments were supplied to a mocked method, even when that argument is created as a local variable in the implementation.

Setting Outputs Based on Inputs with Custom Answers

Mockito is a mature library, so it probably provides everything you want to do. But sometimes those capabilities aren't obvious, either from the API or from the documentation. One such capability involves using the `Answer` interface. Custom answers let you tell Mockito that the output of a mocked method should be based in some way upon its inputs.

Here's a simple example. In the `PersonService` interface, say there's a `savePeople` method that takes a vararg list of `Person` instances as arguments and returns a list of their generated primary keys. It implements this by passing each `Person` to the corresponding `save` method in `PersonRepository`:

PersonService.java

```
public List<Integer> savePeople(Person... person) {
    return Arrays.stream(person)
        .map(repository::save) // save() generates ids
        .map(Person::getId)
        .collect(Collectors.toList());
}
```

The `PersonService` therefore depends on the `PersonRepository` to do its job. If you want to test the `savePeople` method in `PersonService`, you need to mock the `save(Person)` method in `PersonRepository`:

```
public interface PersonRepository {
    // ... other methods ...
    Person save(Person person);
}
```

Here's one attempt to write a test for `savePeople`:

PersonServiceTest.java

```
@Test
public void saveAllPeople() {
```

```
        when(repository.save(any(Person.class)))
            .thenReturn(people.get(0),
                people.get(1),
                people.get(2),
                people.get(3),
                people.get(4));

        // test the service (which uses the mock)
        assertEquals(List.of(1, 2, 3, 14, 5),
            service.savePeople(people.toArray(Person[]::new)));

        // verify the interaction between the service and the mock
        verify(repository, times(people.size())).save(any(Person.class));
        verify(repository, never()).delete(any(Person.class));
    }
```

This example assumes you set up a list of `Person` ahead of time. The expectation on the `save` method is that the first time it's called, it should return the first element of that collection, the second time it returns the second person, and so on. The `thenReturn` method explicitly lists each return value, one by one. When calling `savePeople`, the `toArray` method converts the list into an array, which matches the vararg argument.

The thing is, there should be a way to set the expectations on `save` without stating each return value one by one. After all, all we want is for the method to return its argument. Can't we just say that?

In fact, that's easy to do, though the syntax isn't necessarily intuitive. Instead of using `thenReturn`, use the `thenAnswer` method, which takes as its argument an implementation of the `Answer<T>` interface.

```
OngoingStubbing<T> thenAnswer(Answer<?> answer)
```

The `Answer` interface has a single abstract method called `answer`:

```
T answer(InvocationOnMock invocation) throws Throwable
```

This code doesn't look terribly helpful at first glance, but the key is the `InvocationOnMock` argument. Here are the methods in that class:

```
public interface InvocationOnMock extends Serializable {
    Object getMock();
    Method getMethod();
    Object[] getArguments();
    <T> T getArgument(int index); // return the argument at a given index
    <T> T getArgument(int index, Class<T> clazz);
    Object callRealMethod() throws Throwable;
}
```

The method we want is the fourth one: the overload of `getArgument` that takes an integer index. That allows you to rewrite the expectations line:

```
when(repository.save(any(Person.class)))
            .thenAnswer(invocation -> invocation.getArgument(0));
```

When `save` is invoked with any `Person`, reply with its first argument. That's all that's necessary. The only problem (if there is one) is that the names of those methods don't necessarily tell you that using them, you can tell a mocked method to return an output based entirely on its input arguments.

This is the resulting test:

PersonServiceTest.java

```
@Test
public void useAnswer() {
    // Lambda expression implementation of Answer<Person>
    when(repository.save(any(Person.class)))
        .thenAnswer(invocation -> invocation.getArgument(0));

    List<Integer> ids = service.savePeople(people.toArray(Person[]::new));

    List<Integer> actuals = people.stream()
        .map(Person::getId)
        .collect(Collectors.toList());
    assertEquals(ids, actuals);
}
```

Use an `Answer` if you ever need the output of mocked methods to be based on their input arguments.

Spying to Verify Interactions

Finally, let's talk about spies. A *spy* is an object that wraps an actual dependent instance, which allows you to track the interactions between the class under test and the spies. A spy sits in between the class you are testing and its dependencies. This gives you two capabilities:

1. You can intercept method calls to the dependencies for later verification.
2. You can mock some methods in the dependencies rather than all of them. This is called a *partial mock*.

We'll start with a `PersonService` that uses an actual implementation of the `PersonRepository` interface, called `InMemoryPersonRepository`. The in-memory version internally uses a (synchronized) `ArrayList` to store the people. (The complete implementation is contained in the GitHub repository for the book.)

You can certainly test that the repository is working correctly, and you don't need to use Mockito to do it:

PersonServiceTest.java

```
@Test
void testInMemoryPersonRepository() {
    PersonRepository personRepo = new InMemoryPersonRepository();
    PersonService personService = new PersonService(personRepo);

    personService.savePeople(people.toArray(Person[]::new));
    assertThat(personRepo.findAll()).isEqualTo(people);
}
```

What you can't do is check the expected behavior: verify that the service invokes the `save` method on the repository the expected number of times. You can, however, wrap the repository in a spy and use that to keep count:

PersonServiceTest.java

```
@Test
void spyOnRepository() {
```

```
        // Spy on the in-memory repository
        PersonRepository personRepo = spy(new InMemoryPersonRepository());
        PersonService personService = new PersonService(personRepo);

        personService.savePeople(people.toArray(Person[]::new));
        assertThat(personRepo.findAll()).isEqualTo(people);

        // Verify the method calls on the spy
        verify(personRepo, times(people.size())).save(any(Person.class));
    }
```

That's useful, and a good reason to use spies.

Spies also have the capability to mock some methods rather than all of them. But that's risky because the internal state of the real object can become inconsistent. For example, if you mock the `save` method but not the `count` method in `PersonRepository`, the internal count won't increment when you save a new `Person`. Even worse, the documentation points out that Mockito "does not delegate calls to the passed real instance, instead it actually creates a copy of it." So if you call regular (that is, unstubbed) methods on the spy but not on the real instance itself, you won't see their effects.

Partial mocks are therefore usually more trouble than they're worth. The Mockito documentation discusses a couple of situations where they can be helpful, but the Mockito team discourages their use in general.

Partial Mocks for Subsections

Normally you don't want to use partial mocks because their state is not maintained between the mocked methods and the non-mocked methods. One place where partial mocks can be useful, therefore, is when the mocked methods don't interact with the others.

For example, if part of your system involves authentication or authorization, you can use a partial mock to make sure those methods either confirm or deny a user has been processed. As long as the rest of the system only interacts with the security system through the mocked methods, it should work properly.

Wrapping Up

We covered a lot of ground in this chapter. First, we discussed the methods in the `BDDMockito` class, which are mostly a “global search-and-replace” on the analogous methods in the `Mockito` class. Then we talked about how to set expectations on dependency methods that return `void`, which is easy enough but does take some getting used to.

We also looked at two special situations. Sometimes when a method under test calls a mocked method, you want to know exactly what arguments were supplied. Mockito provides an argument *captor* for that purpose. The other situation is when you’re mocking a method and you want the output to be based on the input in some way. You can use custom answers, implemented as lambda expressions, to handle that.

Finally, we introduced spies, which watch interactions between the class under test and its dependencies. Spies are controversial when used as partial mocks, but they’re perfect as a way of monitoring interactions.

In the next chapter, we’ll look at some more recent additions to Mockito, like the ability to mock final methods and classes or static methods. We’ll also see how the Spring Framework supports Mockito and consider when *not* to mock objects.

Footnotes

[16] https://en.wikipedia.org/wiki/Behavior-driven_development

[17] https://spockframework.org/

Chapter 5

Use Mockito in Special Cases

I spent several years teaching introductory Java courses to developers coming from procedural languages, and I found certain concepts particularly confusing for students. One is the concept of a `static` method, because the English word *static* has nothing to do with class-level methods that can be invoked without an object. The English word *static* is similar to Java's keyword `final`, meaning a method or class that can't be overridden.

While Mockito has always had the ability to let you mock classes, mocking `final` classes and methods or mocking `static` methods requires special care. We'll look at both in this chapter.

Another special situation we'll explore is how the Spring Framework—one of the most popular Java frameworks in the open source world—supports Mockito. Specifically, Spring Boot introduced an annotation called `@MockBean`, which combines mocking a class with Spring's excellent dependency injection capabilities.

Finally, the Mockito team is careful to point out that not every class should be mocked. We'll review their strong recommendations about when *not* to use Mockito.

Mocking Final Classes and Methods

When you mock an interface, Mockito generates an implementation of that interface—not terribly surprising. But Mockito can mock classes too, which requires extending the class and overriding some or all of its methods.

What happens, though, when you ask Mockito to mock a method or a class marked `final`? Final methods can't be overridden, and final classes can't be extended, at least not without some help via reflection. What does Mockito do then?

Mock finals in Mockito 5

In this section, we discuss the additional configuration needed to mock final methods and classes. The ability to mock final members was added in Mockito 2.1.0 and simplified later. Mockito 5, however, integrates this capability into Mockito core. When used with Java 17, Mockito 5 uses the inline capability by default and the old-style mock-maker is no longer supported or needed.

By default, nothing. Without some additional configuration, Mockito can't mock final methods or final classes. You can change this behavior in one of two ways:

1. Add a directory to your project called `mockito-extensions` (usually under `src/main/resources` or `src/test/resources`). Add a file to that directory called `org.mockito.plugins.MockMaker` and, inside that file, add the single line `mock-maker-inline`. Seriously.
2. As a much simpler alternative, in your Gradle or Maven build file, replace the artifact "mockito-core" with "mockito-inline".

The second option is certainly easier, though both options do the same thing.

To show how this works, recall the `InMemoryPersonRepository` class discussed in *Creating Mocks and Stubs*. Because that class used a shared, in-memory database, the underlying data structure had to be protected from multi-threading

issues. That meant the `save` and `delete` methods needed to be synchronized, as shown in the class:

InMemoryPersonRepository.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Optional;

public class InMemoryPersonRepository implements PersonRepository {
    private final List<Person> people =
        Collections.synchronizedList(new ArrayList<>());

    @Override
    public final Person save(Person person) {
        synchronized (people) {
            people.add(person);
        }
        return person;
    }

    @Override
    public final void delete(Person person) {
        synchronized (people) {
            people.remove(person);
        }
    }

    // ... other methods from PersonRepository ...
}
```

To avoid anyone accidentally removing the synchronized block, the `save` and `delete` methods were marked `final`. Say you want to test the `PersonService` with a mock repository, but instead of doing what we've done so far, which is to mock the `PersonRepository` interface itself, you decide to mock the in-memory implementation class. Normally that wouldn't be a problem, but what about the two `final` methods?

If you configure the mock-maker using the steps above, nothing changes. The resulting test looks like this:

PersonServiceTest.java

```
@Test
void testMockOfFinalMethod() {
    // Can mock a class containing final methods
    PersonRepository personRepo = mock(InMemoryPersonRepository.class);

    // Set the expectations on (final) save method in the mock
    when(personRepo.save(any(Person.class)))
        .thenAnswer(invocation -> invocation.getArgument(0));

    // Inject the mock
    PersonService personService = new PersonService(personRepo);

    // Test the service
    List<Integer> ids = personService.savePeople(
        people.toArray(Person[]::new));
    assertThat(ids).containsExactly(1, 2, 3, 14, 5);

    // Verify the save method in the mock was invoked five times
    verify(personRepo, times(5)).save(any(Person.class));
}
```

The test shows that you can mock the class, set expectations on the `final` method (`save` in this example), and verify that the mocked method was invoked the proper number of times. You could also mock classes marked `final` the same way.

Admittedly, there's nothing gained by mocking the class in this case rather than simply mocking the associated interface, but in many situations you'll have only the class rather than an interface. Assuming you own the class (meaning you can modify it), an alternative is to extract an interface from the class and mock that, but if that possibility isn't available, you can mock the class directly.

You can mock classes and even enums in the standard library the same way, though be careful. First, try to avoid mocking code you don't own (see *Deciding When Not to Use Mockito*). Also, optimizations in the library may take advantage of `final` methods by inlining the associated code, so the mocked methods themselves may never be called.

A couple of additional restrictions apply as well. Let's mock a `String`, for

example. We get the following error message:

> org.mockito.exceptions.base.MockitoException: Cannot mock/spy class java.lang.String Mockito cannot mock/spy because : Cannot mock wrapper types, String.class or Class.class

The message couldn't be much clearer. Don't try to mock `String`, `Class`, or any of the wrapper classes (`Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`, or `Boolean`).

The Mockito API includes an annotation called `@DoNotMock` which you can add to your own classes if you want to prevent Mockito from mocking them.

Speaking of `final`, in this book we've been dealing with Java, which allows you to override any method or extend any class as long as it's not marked `final`. The default in Kotlin, however, is the opposite—you can't override anything unless it's marked `open`. Kotlin is the recommended language for Android development, so how do Android developers handle mocking? As it turns out, Mockito has a subproject designed for that.

Using Mockito with Android

One of the challenges of testing Android applications is that while regular unit tests run on the local machine, any test that involves the Android API is an *instrumented* test and requires either an actual device or an emulator to run. As discussed on the Android developer website,[18] if you can mock classes from the Android library, you can improve the performance of your tests considerably.

While normally you don't want to mock classes you don't own, the Mockito Android library was specifically designed to help you perform unit tests on Android applications without requiring emulators or external devices. Android testing is a topic beyond the scope of this book, but be aware that Mockito knows about it and is able to help. See the online resources for details.

The same inline mock maker that Mockito uses to mock `final` methods can also be used to mock `static` methods, which we cover next.

Mocking Static Methods

Until Java 8, you couldn't have static methods in interfaces at all. As for classes, static methods are associated with the class as a whole, rather than an instance, and they can't be overridden. So what does it even mean to mock a static method?

It's easy enough to imagine a system that has a dependency on a static method. For example, consider a class, called `BioService`, which takes a vararg list of strings in the constructor and grabs a summary of their associated pages at Wikipedia. Wikipedia maintains a MediaWiki Action API,[19] which you can use to access Wikipedia programmatically.

(All the URLs end with `.php`, which tells you how Wikipedia is implemented.)

The book's GitHub repository contains a package called `com.kousenit.wikipedia` that uses this API. The `BioService` class is very simple:

BioService.java

```
package com.kousenit.wikipedia;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class BioService {
    private final List<String> pageNames;

    public BioService(String... pageNames) {
        this.pageNames = Arrays.stream(pageNames)
            .collect(Collectors.toList());
    }

    public List<String> getBios() {
        return pageNames.stream()
            .map(WikiUtil::getWikipediaExtract)
            .collect(Collectors.toList());
    }
```

```
}
```

In this case, the dependency is on a class called `WikiUtil`, which has a static method called `getWikipediaExtract`. That method takes a string name and returns the page with that name as a string.

For the `BioService`, you give the constructor a list of names, and the `getBios` method gets the Wikipedia "extract" for each one. The `WikiUtil` class is fairly complicated because it has to formulate the HTTP request from many parameters and parse the results.

Here's an integration test (without Mockito) for this class:

BioServiceTest.java

```
@Test
// Integration test
void checkBios() {
    BioService service = new BioService("Anita Borg", "Ada Lovelace",
        "Grace Hopper", "Barbara Liskov");

    assertThat(service.getBios()).hasSize(4);
}
```

This test checks only that the right number of bio pages are returned, but you can print them if you like.

The question is how do you create a unit test for the `BioService` class when its only dependency is the call to the static method in `WikiUtil`?

To mock static methods, Mockito provides a method called, naturally enough, `mockStatic`, which takes a `Class<T>` reference as an argument. The method returns an interface called `MockedStatic`. The docs say that the `mockStatic` method creates a thread-local controller for all static methods in the supplied class or interface and that you should call the `close` method on it when the test completes. In practice, that advice means you call the `mockStatic` method inside a try-with-resources block.

Here's a unit test for the `BioService` that mocks the static methods in `WikiUtil`:

BioServiceTest.java

```
@Test
void testBioServiceWithMocks() {
    BioService service = new BioService("Anita Borg", "Ada Lovelace",
        "Grace Hopper", "Barbara Liskov");

    // Use mockStatic in a try-with-resources block
    try (MockedStatic<WikiUtil> mocked = mockStatic(WikiUtil.class)) {

        // Same when/then methods as a regular mock
        mocked.when(() -> WikiUtil.getWikipediaExtract(anyString()))
            .thenAnswer(invocation -> "Bio for " +
                invocation.getArgument(0));

        assertThat(service.getBios()).hasSize(4);

        // Verify using a MockedStatic.Verification argument
        mocked.verify(() -> WikiUtil.getWikipediaExtract(
            anyString()), times(4));
    }
}
```

You may notice these points of interest:

1. The `MockedStatic` instance is created in a try-with-resources block.
2. The expectations on the static method use a lambda expression and a `thenAnswer` call because the mocked reply is based on the input argument.
3. You can call the `getBios` method in the class under test and check the number of results.
4. Optionally, you can verify that the static method was called the right number of times. This version of `verify` takes a `MockedStatic.Verification` argument, whose single abstract method, `apply`, takes no arguments and returns `void`. This call to `verify` therefore uses a compatible expression lambda, followed by the required multiplicity.

The test works, but it's a bit involved due to the additional complexity. You can follow this model whenever you need to mock static methods.

Intrinsics Not Mocked

Be careful with mocking library methods. In addition to violating the principle “Don’t mock what you don’t own,” some static methods are “intrinsified” by the JVM, meaning they have their own low-level implementations for efficiency. For example, if you try to mock `Math.max`, your version is likely to be ignored because the intrinsic implementation replaces the method call.

Mocking Constructors

So far, we've been lucky because every time we wanted to mock a dependency of a class under test, we had an easy way to insert the mock into the class. Most of the time, we've used constructors for this purpose, like when we injected a mock of the `PersonRepository` into the `PersonService` to test the service.

Of course, this wasn't luck at all; the applications were designed with constructors like that in mind. But what if they weren't? What if we had a class that instantiated a dependency as a local variable? How would we mock that?

Back in Chapter 1, *Build a Testing Foundation*, we looked at a class called `HelloMockito`, which had two dependencies: a `PersonRepository` and a `TranslationService`. Here's a reminder:

HelloMockitoRevised.java

```
public class HelloMockito {
    private String greeting = "Hello, %s, from Mockito!";

    // Dependencies
    private final PersonRepository personRepository;
    private final TranslationService translationService;

    // Constructor to inject the dependencies
    public HelloMockito(PersonRepository personRepository,
                        TranslationService translationService) {
        this.personRepository = personRepository;
        this.translationService = translationService;
    }

    // Method we want to test
    public String greet(int id, String sourceLang, String targetLang) {
        Optional<Person> person = personRepository.findById(id);
        String name = person.map(Person::getFirst).orElse("World");
        return translationService.translate(
            String.format(greeting, name), sourceLang, targetLang);
    }
}
```

What if, instead of supplying the `TranslationService` implementation from outside, we included a constructor for `HelloMockito` that instantiated the service directly?

```
public HelloMockito(PersonRepository personRepository) {
    this(personRepository, new DefaultTranslationService());
}
```

The `DefaultTranslationService` is a class that implements the `TranslationService` interface. It can be simple or complicated, as needed. For example, a trivial implementation could be the following:

DefaultTranslationService.java

```
public class DefaultTranslationService implements TranslationService {
    public String translate(String text,
                            String sourceLanguage,
                            String targetLanguage) {
        return text;
    }
}
```

This just returns the message without modification.

Since version 3.5, Mockito has had the ability to mock constructors. As with mocking static methods, you need to use the `mockito-inline` dependency to take advantage of it, and you should wrap the mocked object inside a try-with-resources block. So we could write a test for `HelloMockito` that uses the default translation service this way:

HelloMockitoMockedConstructorTest.java

```
@Test
void greetWithMockedConstructor() {
    // Mock for repo (needed for HelloMockito constructor)
    PersonRepository mockRepo = mock(PersonRepository.class);
    when(mockRepo.findById(anyInt()))
        .thenReturn(Optional.of(
            new Person(1, "Grace", "Hopper", LocalDate.now())));

    // Mock for translator (instantiated inside HelloMockito constructor)
    try (MockedConstruction<DefaultTranslationService> ignored =
```

```
            mockConstruction(DefaultTranslationService.class,
                (mock, context) ->
                    when(mock.translate(
                        anyString(), anyString(), anyString()))
                        .thenAnswer(
                            invocation -> invocation.getArgument(0) +
                            " (translated)"))) {

        // Instantiate HelloMockito with mocked repo and
        // locally instantiated translator
        HelloMockito hello = new HelloMockito(mockRepo);
        String greeting = hello.greet(1, "en", "en");
        assertThat(greeting).isEqualTo(
            "Hello, Grace, from Mockito! (translated)");
    }
}
```

Let's go through the several moving parts one by one. Remember, the big picture is to test the `greet` method inside the `HelloMockito` class. First, we need to look up an individual in the `PersonRepository`, so the test uses the `mock` method with the `when/then` construct to add Grace Hopper as person with `id` of 1.

Then comes the fun part. In a try-with-resources block, the static `mockConstruction` method of `Mockito` is invoked and has this signature:

```
public static <T> MockedConstruction<T> mockConstruction(Class<T> classToMock,
    MockedConstruction.MockInitializer<T> mockInitializer)
```

The first argument is the class we're mocking, which is `DefaultTranslationService`. The second argument is an instance of the `MockInitializer<T>` interface, which is a functional interface (meaning it contains only a single, abstract method). Since it's a functional interface, we can implement it using a lambda expression. The single abstract method has this signature:

```
void prepare(T mock, MockedConstruction.Context context)
```

The good news is we don't care about the details. We simply add a lambda of two arguments that uses the Mockito API to arrange what we want, which is the following in our test:

```
(mock, context) -> when(mock.translate(anyString(), anyString(), anyString()))
```

```
        .thenAnswer(invocation -> invocation.getArgument(0) + " (translated)")
```

In other words, we're specifying that if we call the `translate` method on the mock with three instances of `anyString`, it should reply with the first string argument followed by "(translated)", just to let us know we did something.

Then, in the `try` block, we instantiate the `HelloMockito` class using that single-argument constructor, which in turn instantiates the `DefaultTranslationService`. We then call the `greet` method and check that the mocked constructor in `DefaultTranslationService` did what we told it to do. When we exit the `try` block, Java will automatically invoke the `close` method on the `MockedConstruction` instance because it extends `AutoCloseable`.

It's also odd that in our `try` block, we're not using the `MockedConstruction` reference directly, which is why we called the variable `ignored`. We had to assign the result of the `mockConstruction` method to something in order to use try-with-resources; but remember, we're not the ones actually instantiating the `DefaultTranslationService`, the `HelloMockito` class is.

The process feels involved, but it does the job, which is to mock the instantiation of a class as a local variable. We're basically intercepting that process and providing our own implementation.

One quirk here is that we might have preferred to mock the construction of `TranslationService`, but that's an interface, and the argument to `mockConstruction` has to be a non-abstract Java class. That's why we mocked the `DefaultTranslationService` instead.

Any instantiation of `DefaultTranslationService` inside the `try` block will return the mocked instance, and any instantiation before or after the `try` block will give us the original.

By the way, in this case a simplification is available. Instead of using `mockConstruction`, we can use the `mockConstructionWithAnswer` method, which takes a class and a default `Answer`. So we could reduce our implementation:

HelloMockitoMockedConstructorTest.java

```
@Test
void greetWithMockedConstructorWithAnswer() {
    // ... mock the PersonRepository as before ...

    try (MockedConstruction<DefaultTranslationService> ignored =
             mockConstructionWithAnswer(DefaultTranslationService.class,
                 invocation -> invocation.getArgument(0) +
                     " (translated)")) {

        HelloMockito hello = new HelloMockito(mockRepo);
        String greeting = hello.greet(1, "en", "en");
        assertThat(greeting).isEqualTo(
            "Hello, Grace, from Mockito! (translated)");
    }
}
```

Not a huge simplification, but a useful one nevertheless. In fact, we could add multiple answers, and each would be returned on separate invocations of the `greet` method for different mocks, but that's definitely getting beyond the scope of this discussion.

Mocking constructors shouldn't come up very often, but when it does, we now have a way of doing so.

Working with the Spring Framework

Earlier, in *Injecting the Mocks into the Class Under Test*, we talked about how when using the `@InjectMocks` annotation, Mockito will do its best to plug the mocks into the class under test. The documentation warns you, however, that Mockito is not a true dependency injection framework.

As it turns out, almost all Java developers regularly work with a true dependency injection framework, called the Spring Framework. Spring supported Mockito from the beginning, and whenever you create a Spring app, Spring includes the Mockito library. Since a full discussion of the Spring Framework is well beyond the scope of this book, we'll assume you're familiar with it and are reading this section because Spring apps already include Mockito as a test dependency and you want to use them together.

The Spring Boot project provides an annotation called `@MockBean` that does two things:

1. It creates a mock object for you, replacing the Mockito annotation `@Mock`.
2. It replaces any instance of the mocked class currently in the `ApplicationContext` with your mock so that it's available for autowiring.

So in Spring, rather than using `@Mock` and `@InjectMocks`, you use `@MockBean` and rely on Spring to instantiate the mocked beans and autowire them into your class under test.

Say you have a standard Spring app that involves a RESTful web service. It includes a service class, to do business logic and mark transaction boundaries, and one or more repository classes that convert objects to database tables and back again. The goal is to test the logic in the service class without needing a real database, so to do that you need to mock the repositories and inject them into the service. That's where the `@MockBean` annotation comes in.

Returning once again to our `PersonService` that depends on a `PersonRepository`, if this was part of a Spring app, the repository would likely be autowired into the

service:

PersonServiceSpring.java

```
@Service // Can only autowire into a managed bean
public class PersonService {
    private final PersonRepository repository;

    @Autowired // Not required if only one constructor, but doesn't hurt
    public PersonService(PersonRepository repository) {
        this.repository = repository;
    }

    // ... methods that use the repository ...
}
```

When writing a test for the `PersonService` outside of Spring, we used `@Mock` on the repository and `@InjectMocks` on the service. That won't work with Spring's autowiring, however. Instead, you use the `@MockBean` annotation to make the repository do what you want.

PersonServiceSpringTest.java

```
@SpringBootTest
class PersonServiceTest {
    @MockBean // Instantiate mock and put into the application context
    private PersonRepository repository;

    @Test
    public void findMaxId() {
        // Set expectations on mock
        when(repository.findAll()).thenReturn(people);

        // Test service method
        assertEquals(14, service.getHighestId().intValue());

        // Verify method call on mock
        verify(repository).findAll();
    }
}
```

The `@MockBean` annotation on the `PersonRepository` not only creates a mock

object, it replaces any existing `PersonRepository` in the application context with the mock. That makes it available for autowiring, and Spring does the rest.

Spring also has a similar annotation called `@SpyBean`. That annotation wraps a Mockito spy around an existing instance of the field. As with all spies, they're useful when you want to verify that methods on the dependencies got called the right number of times in the right order, but you still want to use real instances of the dependencies themselves.

Next we'll consider another controversial notion—when should you avoid using Mockito at all?

Deciding When *Not* to Use Mockito

You likely now have a good grasp of cases for which Mockito is useful. But when is it not the best idea to use Mockito? Let's explore that question.

If you look in the book's GitHub repository, you'll find a test for the `getAstroResponse` method in the `AstroGateway`:

AstroGatewayTest.java

```
@Test
void testDeserializeToRecords() {
    AstroResponse result = gateway.getResponse();
    result.getPeople().forEach(System.out::println);
    assertAll(
        () -> assertTrue(result.getNumber() >= 0),
        () -> assertEquals(result.getPeople().size(), result.getNumber())
    );
}
```

The test instantiates the class and invokes its `getResponse` method. Then the data is checked to confirm that the `number` property of the `AstroResponse` record is non-negative (there can't be a negative number of astronauts) and that the size of the `people` collection matches that number.

The implementation uses external libraries to do its job:

- An HTTP client to call the RESTful web service.
- A JSON parser to convert the response into Java records.

Consequently, the test is arguably an integration test. That's fine, but it means that if something goes wrong with either of those two features, the test will fail. The network could go down, or the parser might not understand how to work with records, or there could be bugs in either library. If the test passes, you're good, but if not, where is the problem?

One way to isolate potential problems is to convert to a unit test. The idea would be to mock both the networking library and the parsing library so that the

`AstroGateway` could be evaluated as a solitary object. This plan runs into problems very quickly. For one, there's no obvious way to get mocks into the `AstroGateway` class. It's not designed to allow you to substitute other networking or JSON libraries into it, via setter methods or constructor arguments or anything else. How are you going to get a mock `HttpClient` and `HttpRequest` into the class, and how do you create a mock `HttpResponse` that will provide the right behavior?

Another problem—what happens if the libraries change? The tests may still work, because the libraries are mocked, but as soon as you put everything into practice the test suddenly breaks due to new internal changes inside the libraries.

Finally, with mocking the libraries, a single mock might lead to an entire series of mocks if the library itself is complicated. For example, do you need to mock the static `newBuilder` methods on `HttpClient` and `HttpRequest`? What about the other methods chained to them? The result would lead to overly specified tests and very complicated test cases.

For all these reasons, the recommendation[20] of the Mockito team is this:

Do not mock a type you don't own.

That means, for our `AstroGateway`, an integration test is appropriate. Mocking everything is an anti-pattern. When evaluating the `AstroService`, mocking the `AstroGateway` makes sense. When testing the `AstroGateway`, stick with an integration test.

WireMock

The general principle in this section—do not mock classes you don't own—is solid. That said, if your specific use case is to replace a network call with a fake one that returns given values, the WireMock[21] project is a good choice for this purpose.

Wrapping Up

In this chapter, we covered a few of the newer capabilities of Mockito, namely, how to mock `final` classes and methods and `static` methods. We looked at the `@MockBean` annotation provided by the Spring Framework that creates a mock and replaces any existing instance in the application context.

Finally, we used our Astro project to uncover reasons for the recommendation from the Mockito team to not mock types you don't own.

Hopefully you now feel ready to use Mockito as it was intended—to make your unit tests more robust, reliable, and helpful.

Footnotes

[18] https://developer.android.com/training/testing/local-tests#mockable-library

[19] https://www.mediawiki.org/wiki/API:Main_page

[20] https://github.com/mockito/mockito/wiki/How-to-write-good-tests

[21] https://wiremock.org/

Appendix 1

Running Mockito Tests

This appendix describes the steps needed to run JUnit tests with Mockito. We'll cover both JUnit 5 and the older JUnit 4. The steps used for JUnit 4 will also work for JUnit 3, though the tests themselves use a different API. Here we assume you're using either the Maven or Gradle build tool. Steps for running with other tools like Ant or Bazel can be found in the JUnit documentation.

Steps Common to All JUnit Versions

The Maven coordinates for Mockito are the same, whether you're using Gradle or Maven as a build tool. Following is the *core* dependency for Mockito in a Maven `pom.xml` file:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
</dependency>
```

And for Gradle, here's the short version, using the Groovy DSL:

```
testImplementation 'org.mockito:mockito-core:4.8.1'
```

The only difference with the Kotlin DSL is that you use double quotes instead of single quotes for the dependency, and the dependency needs to be wrapped in parentheses:

```
testImplementation("org.mockito:mockito-core:4.8.1")
```

If you plan to mock final methods or classes, constructors, or static methods, you need the *inline mock maker* dependency, which simply replaces `mockito-core` with `mockito-inline`. The rest is the same. Though Mockito 5 has not yet been released (an official release is expected very soon), word is that the inline mock maker will be wrapped into the core and become the default.

If all you plan to use in Mockito is the static `mock` method to create your mocks, this dependency is all you need. If, however, you plan to use annotations like `@Mock` and `@InjectMocks`, you need an additional dependency if you use JUnit 5, as shown in the next section.

JUnit 5

JUnit 5 uses *extension* classes to add capabilities to JUnit. This is the dependency in Maven:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
</dependency>
```

And for Gradle it's this:

```
testImplementation 'org.mockito:mockito-junit-jupiter:4.8.1'
```

Again, add parentheses and double-quotes if you're using the Kotlin DSL for Gradle. In either case, you can now add the following to the test class:

```
@ExtendWith(MockitoExtension.class)
```

The extension will work with your annotations to use Mockito for the tests.

Remember, if you plan to use JUnit 5 with Gradle, you must add the following to your build file:

```
tasks.named('test') {
    useJUnitPlatform()
}
```

The syntax shown only executes the `useJUnitPlatform` method if you run a test task. To make that eager, you can replace that line with simply `test`.

If you plan to use both JUnit 5 and JUnit 4 tests in the same project, be sure to include the *vintage* engine with your JUnit dependencies:

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.9.1</version>
```

```
        <scope>test</scope>
    </dependency>
```

Or, for Gradle, use this:

```
    testImplementation 'org.junit.vintage:junit-vintage-engine:5.9.1'
```

With both the regular JUnit 5 dependency and the vintage engine, you can choose which version of JUnit to use on each test class.

JUnit 4

JUnit 4 has three ways to tell Mockito to work with annotations. First, a JUnit runner is available. Add the following `@RunWith` annotation to your class:

```
@RunWith(MockitoJUnitRunner.class)
```

Alternatively, you can use the following JUnit Rule:

```
@Rule
public MockitoRule rule = MockitoJUnit.rule()
                                      .strictness(Strictness.STRICT_STUBS);
```

If you don’t specify a *strictness*, as here, the default for JUnit 4 is lenient. The default for JUnit 5 is strict.

The third way to tell Mockito to use the annotations is to invoke the `openMocks` method, usually called in a set up method tagged with `@Before` (which runs before each test):

```
@Before
public void setUp() {
    MockitoAnnotations.openMocks(this);
}
```

Once the dependencies have been added to the build files and the annotation support is configured, you can run the tests the way you would run any JUnit tests, either in your IDE or at the command line. Both Maven and Gradle include `test` tasks you can execute.

Pragmatic Programmer ebook.

And if you'd like to share your own expertise with the world, why not propose a
of our best authors started off as our readers, just like you. With a 50% royalty, w
name you trust, there's nothing to lose. Visit https://pragprog.com/become-an-aut
started.

We thank you for your continued support, and we hope to hear from you again so

The Pragmatic Bookshelf