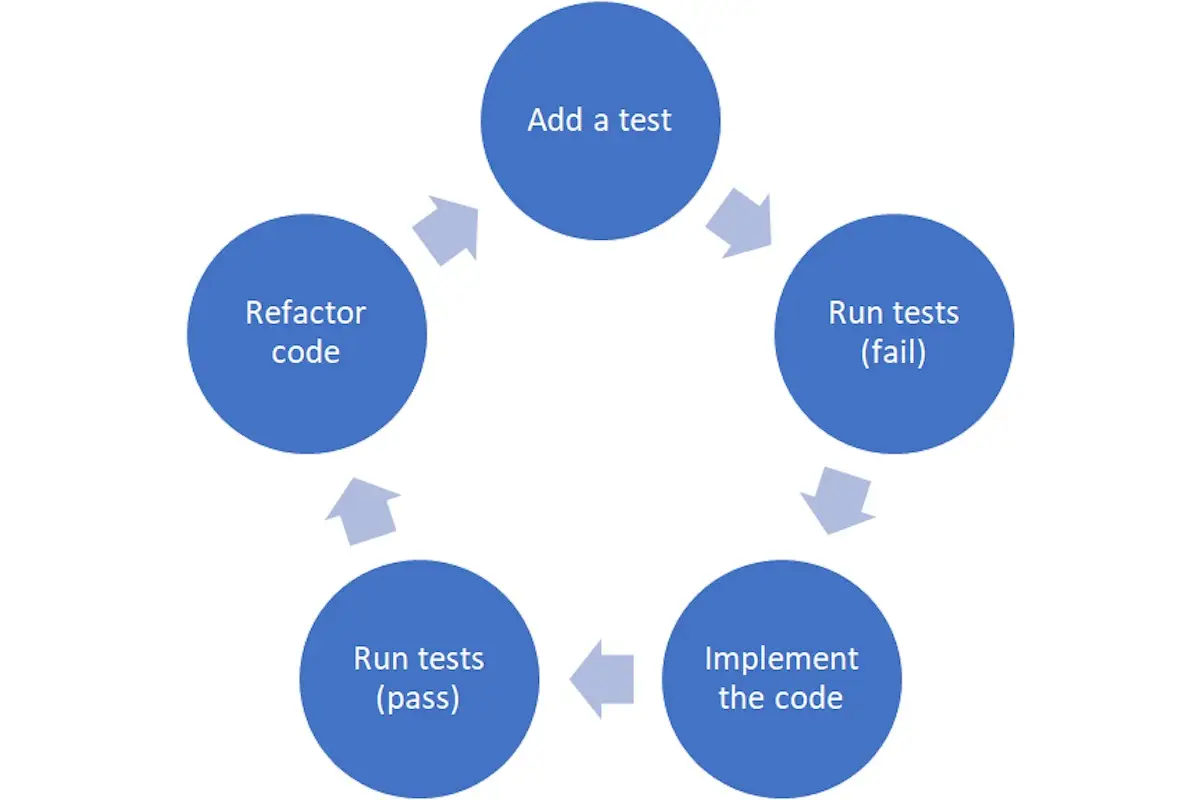
**Test Driven Development**

**Test-driven development (TDD)** is a software development process that interweaves coding, testing, and design. It is **a test-first approach** that aims to improve the quality of your applications. Test-driven development is defined by the following lifecycle

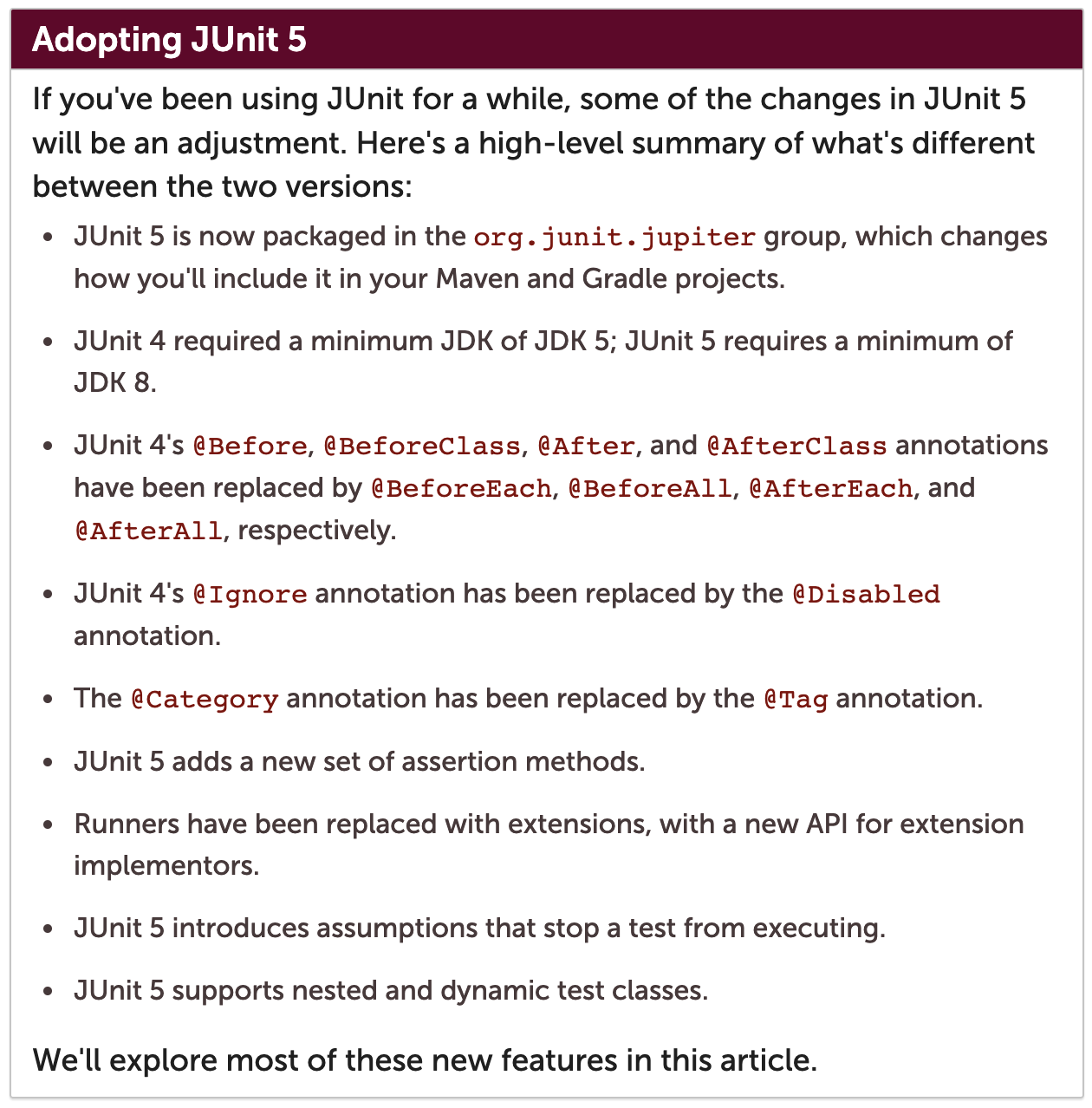


1. Add a test.
2. Run all of your tests and observe the new test failing.
3. Implement the code.
4. Run all of your tests and observe the new test succeeding.
5. Refactor the code.

There's a twofold purpose to writing tests before writing your code.

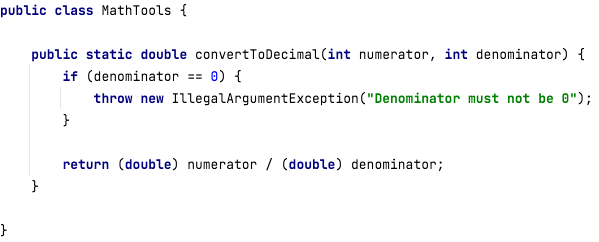
First, **it forces us to think about the business problem we are trying to solve**. For example, how should successful scenarios behave? What conditions should fail? How should they fail?

Second, **testing first gives us more confidence in our tests**. Whenever we write tests after writing code, we always have to break them to ensure that they are actually catching errors. Writing tests first avoids this extra step.



**Unit testing with JUnit 5**

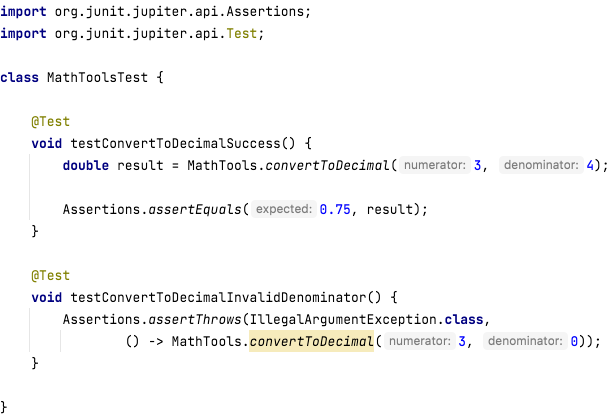
Listing 1 shows a **MathTools** class whose method **converts a numerator and denominator** to a **double**.



We have two **primary scenarios for testing the MathTools class** and its method:

* A ***valid test***, in which we pass **non-zero integers** for the numerator and denominator.
* A ***failure scenario***, in which we pass a **zero value for the denominator**.

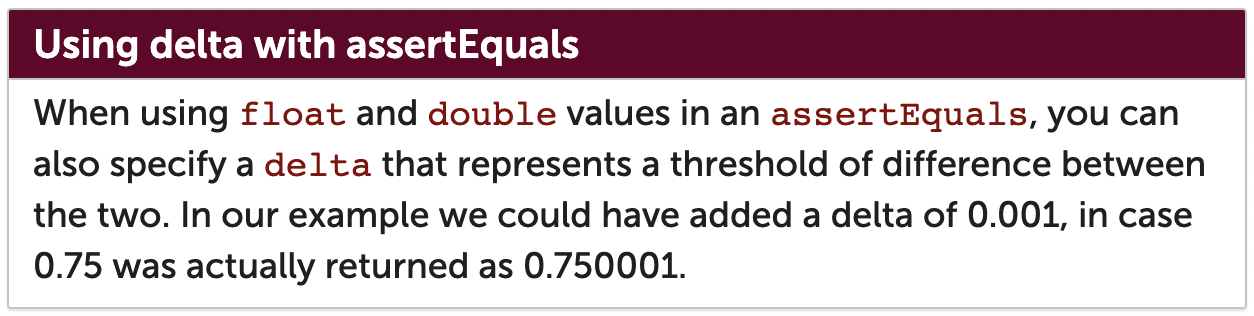
Listing 2 shows a **JUnit 5 test class** to test these two scenarios.



The **org.junit.jupiter.api.Test** **annotation** **denotes a test method**. Note that the **@Test annotation** now **comes from the JUnit 5 Jupiter API** **package** **instead of JUnit 4's org.junit package**.

The **org.junit.jupiter.api.Assertions** **class** **provides a set of static methods for comparing actual and expected results**.

The **assertThrows** **method** **executes the function** and **validates that the expected type of exception is thrown**.



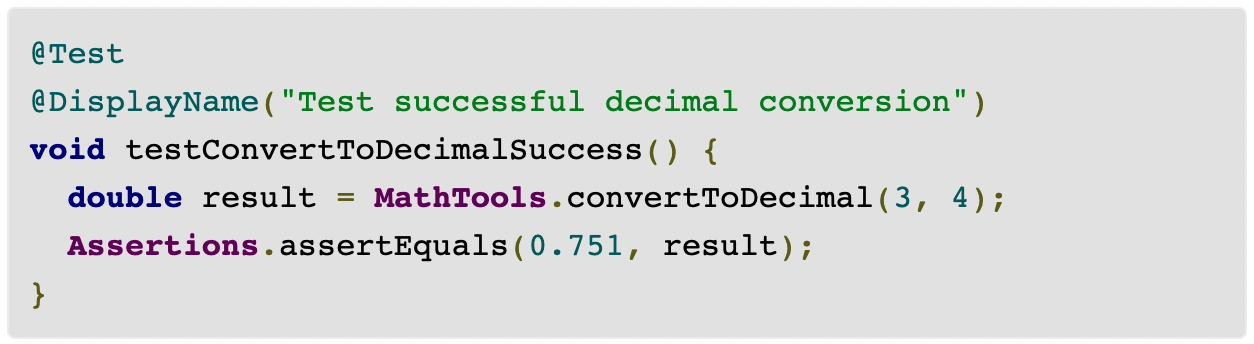
In addition to validating a value or behavior, the **assert methods can also accept a textual description of the error**, which can **help us diagnose failures**. For example:

**Assertions**.assertEquals(0.75, result, "The MathTools::convertToDecimal value did not return the correct value of 0.75 for 3/4");

**Assertions**.assertEquals(0.75, result, () -> "The MathTools::convertToDecimal value did not return the correct value of 0.75 for 3/4");

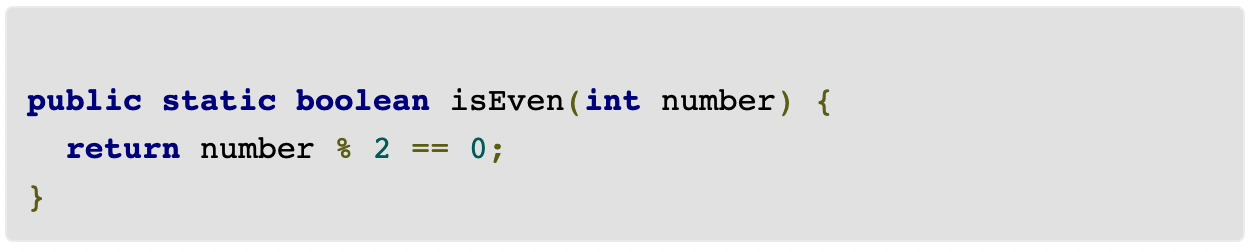
The difference between the two variations is that **the first one always creates the message**, **even if it is not displayed**, whereas **the second one only constructs the message if the assertion fails**. In this case, the construction of the message is trivial, so it doesn't really matter. Still, **there is no need to construct an error message for a test that passes**, so it's usually a **best practice to use the second style**.

Finally, if you're using [an **IDE like IntelliJ**](https://www.javaworld.com/article/3114167/choosing-your-java-ide.html) to run your tests, **each test method will be displayed by its method name**. This is fine if your method names are readable, but you can also **add a @DisplayName annotation to your test methods to better identify the tests**:

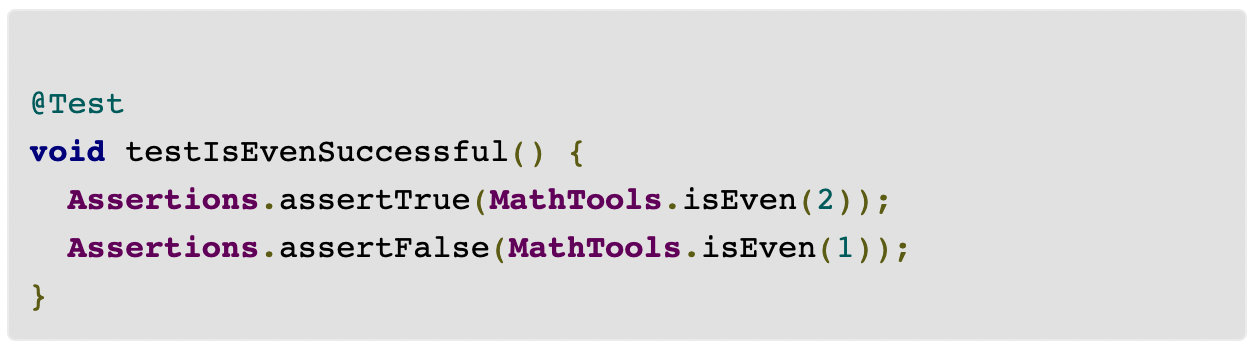


## **Parameterized tests in JUnit 5**

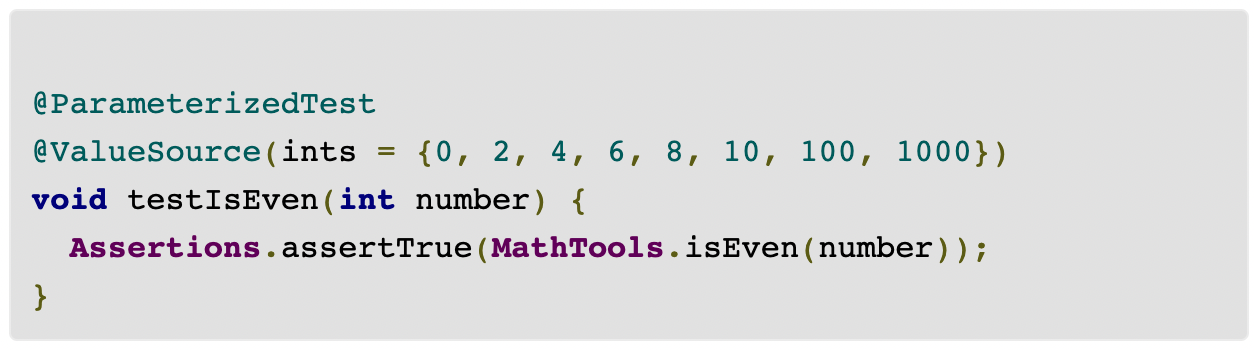
To start, let's add another method to the MathTools class, named isEven:



We could test this code the same way we did in the previous section, by **passing different numbers** to the **isEven** method and **validating the response**:



The methodology works, but if we want **to test a large number of values**, it will soon **become cumbersome to enter the values manually**. In this case, **we can use a parameterized test to specify the values that we want to test**:

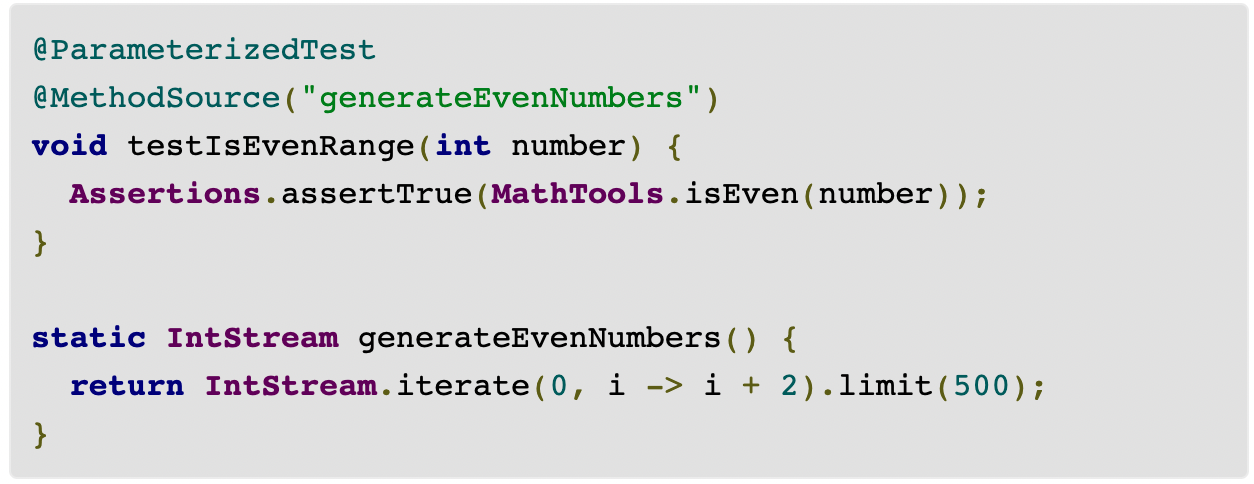


For this test, we use the **@ParameterizedTest** annotation instead of the @Test annotation. We also have to provide a **source** for the parameters.

### **Using sources in parameterized testing**

There are **different types of sources**, but the **simplest is the @ValueSource**, which **lets us specify a list of Integers or Strings**. The **parameter is passed as an argument to the test method** and then can be used in the test. In this case, we're passing in eight even integers and validating that the MathTools::isEven method properly identifies them as even.

This is **better, but we still have to enter all of the values** we want to test. What would happen if we wanted to **test all the even numbers between 0 and 1,000?** Rather than manually entering all 500 values, we could **replace our @ValueSource with a @MethodSource**, **which generates the list of numbers for us**. Here's an example:



When using a **@MethodSource**, we **define a static method** that **returns a stream or collection**. **Each value will be sent to our test method as a method argument**. In this example, we create an IntStream, which is a stream of integers. The IntStream starts at 0, increments by twos, and limits the total number of items in the stream to 500. This means that the isEven method will be called 500 times, with all even numbers between 0 and 998.

Parameterized tests include support for the following types of sources: **ValueSource, MethodSource, EnumSource, CsvSource, CsvFileSource, ArgumentSource, NullSource, EmptySource, etc …**

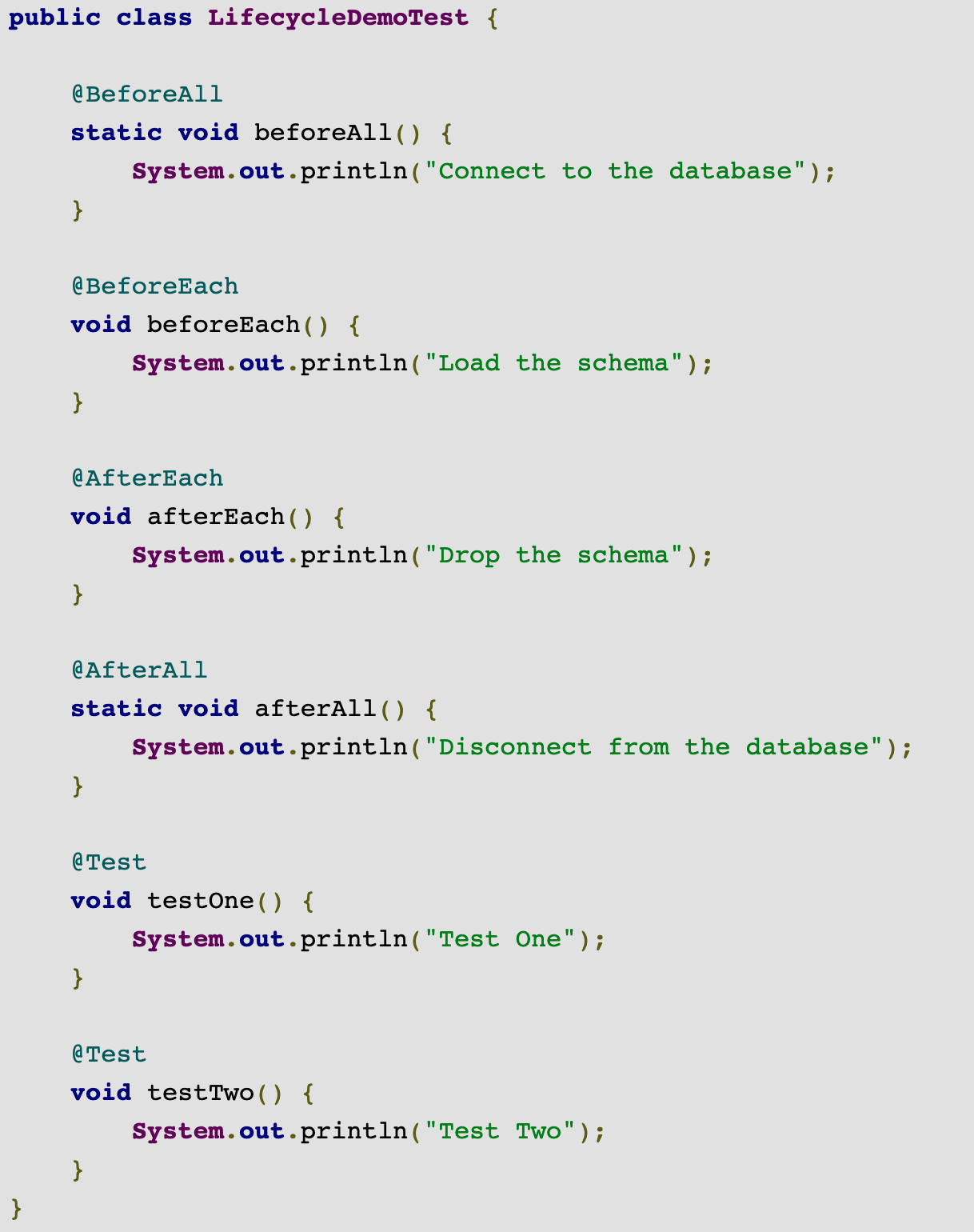
## **JUnit 5's test lifecycle**

For many tests, **there are things that we might want to do before and after each of our test runs** and **before and after all of our tests run**. For example, if we were testing database queries, we might want to set up a connection to a database and import a schema before all the tests run, insert test data before each individual test runs, clean up the database after each test runs, and then delete the schema and close the database connection after all the tests run.

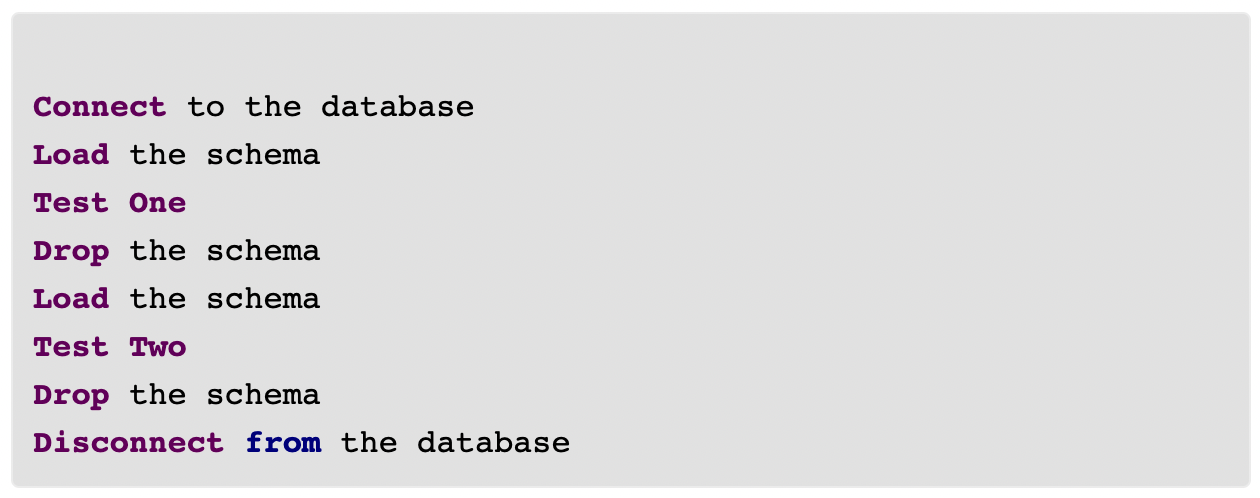
**JUnit 5 provides the following annotations** that we can **add to methods in the test class** to do this:

1. **@BeforeAll**:
   * A **static method** in your test class that is **called before all of its tests run**.
2. **@AfterAll**:
   * A **static method** in your test class that is **called after all of its tests run**.
3. **@BeforeEach**:
   * A **method** that is **called before each individual test runs**.
4. **@AfterEach**:
   * A **method** that is **called after each individual test runs**.

Listing 5 shows a very **simple example that logs the invocations** of the **various lifecycle methods**.



The output from running this test prints the following:



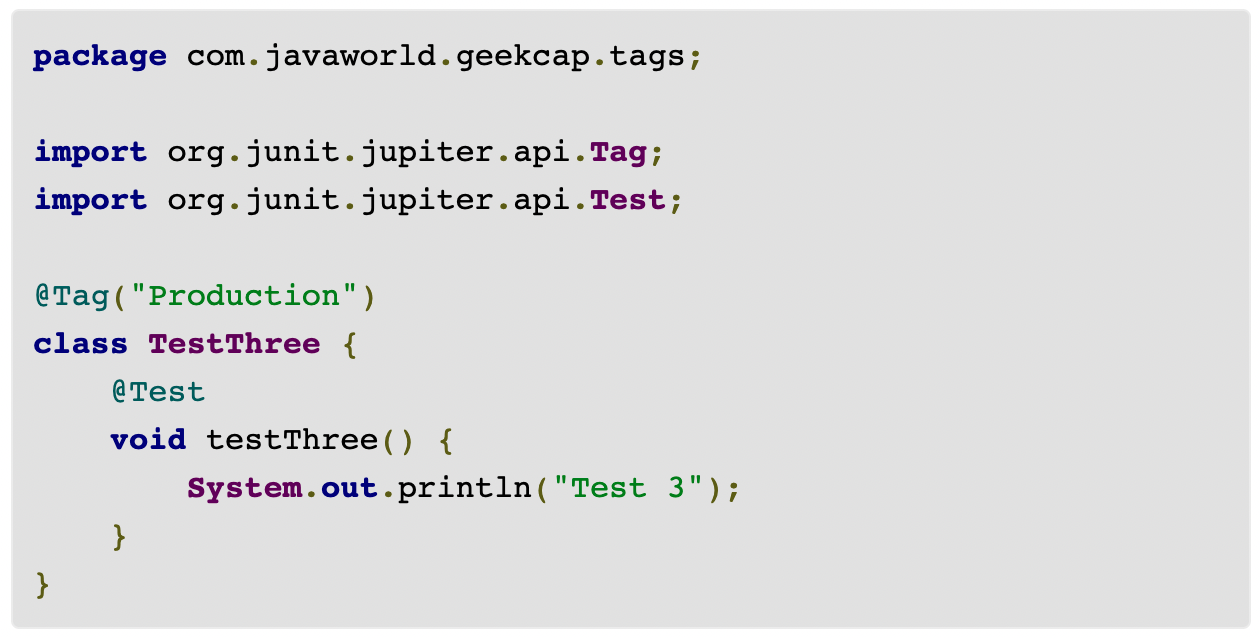
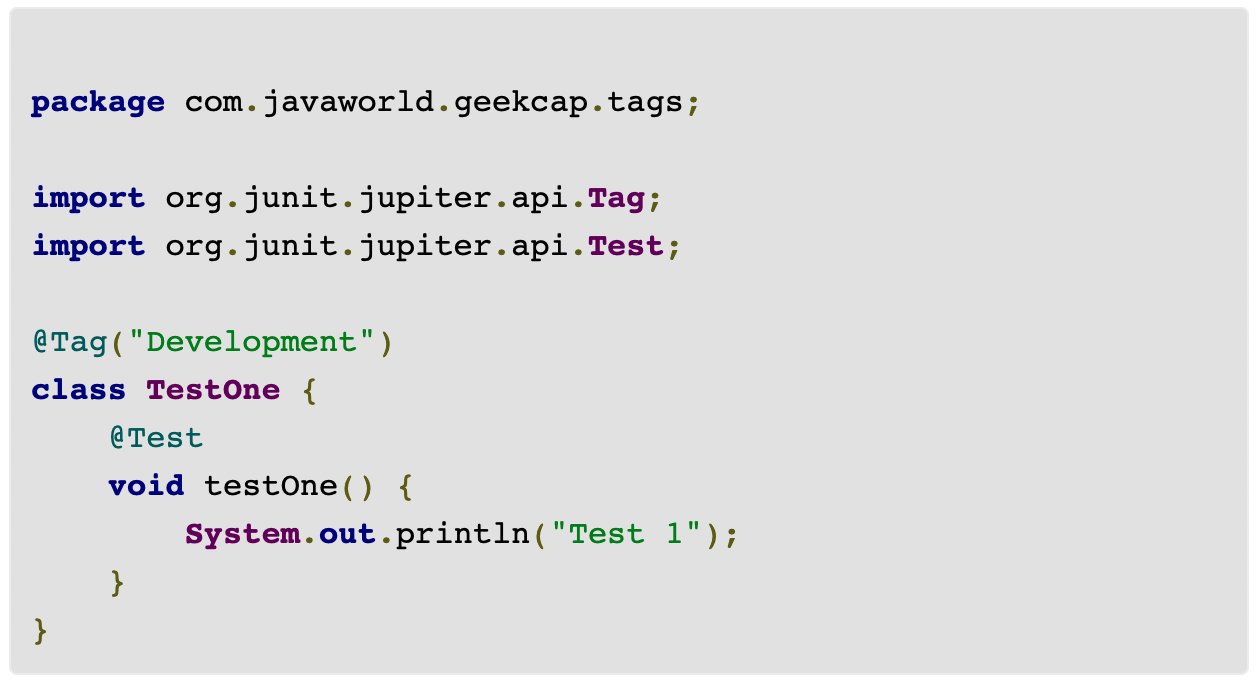
As you can see from this output, the **beforeAll** method is **called first** and can do something like connect to a database or create a large data structure into memory. Next, the **beforeEach** method does things to **prepare the data for each test**, such as populating a test database with an expected set of data. Then the first test runs, followed by the **afterEach** method. **This process of: beforeEach, test, and afterEach continues until all the tests have completed**. Finally, the **afterAll** method **cleans up the test environment**, such as by disconnecting from a database.

## 

## **New in JUnit 5: Tags**

**Tags** are **used to identify** and **filter specific tests** that you want **to run in different scenarios**. For example, you can tag a test class or a test method as an integration test and another as development. The names and uses of the tags are all up to you.

We'll create **three new test classes** and **tag two of them as development** and **one as production**, presumably to differentiate between tests you want to run when building for different environments.



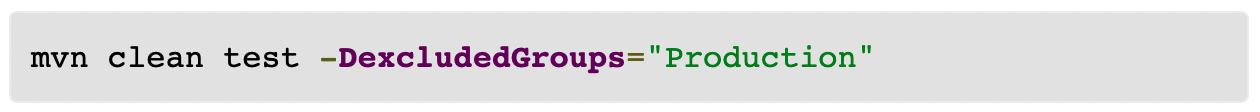
**Tags** are **implemented through annotations**, and you can **annotate either an entire test class** or **individual methods in a test class**; furthermore, **a class or a method can have multiple tags**. In this example, **TestOne** and **TestTwo** are annotated with the "**Development**" tag, and **TestThree** is annotated with the "**Production**" tag. We can filter test runs in different ways based on tags. The simplest of these is to specify a test in your Maven command line; for example, the following only executes tests tagged as "Development":



The **groups** property **allows us to specify a comma-separated list of tag names** for the tests that you want JUnit 5 to run.



In addition to the **groups** property, JUnit 5 allows you to use an **excludedGroups** property to **execute all tests that do not have the specified tag**. For example, in a development environment, we do not want to execute the production tests, so we could execute the following:



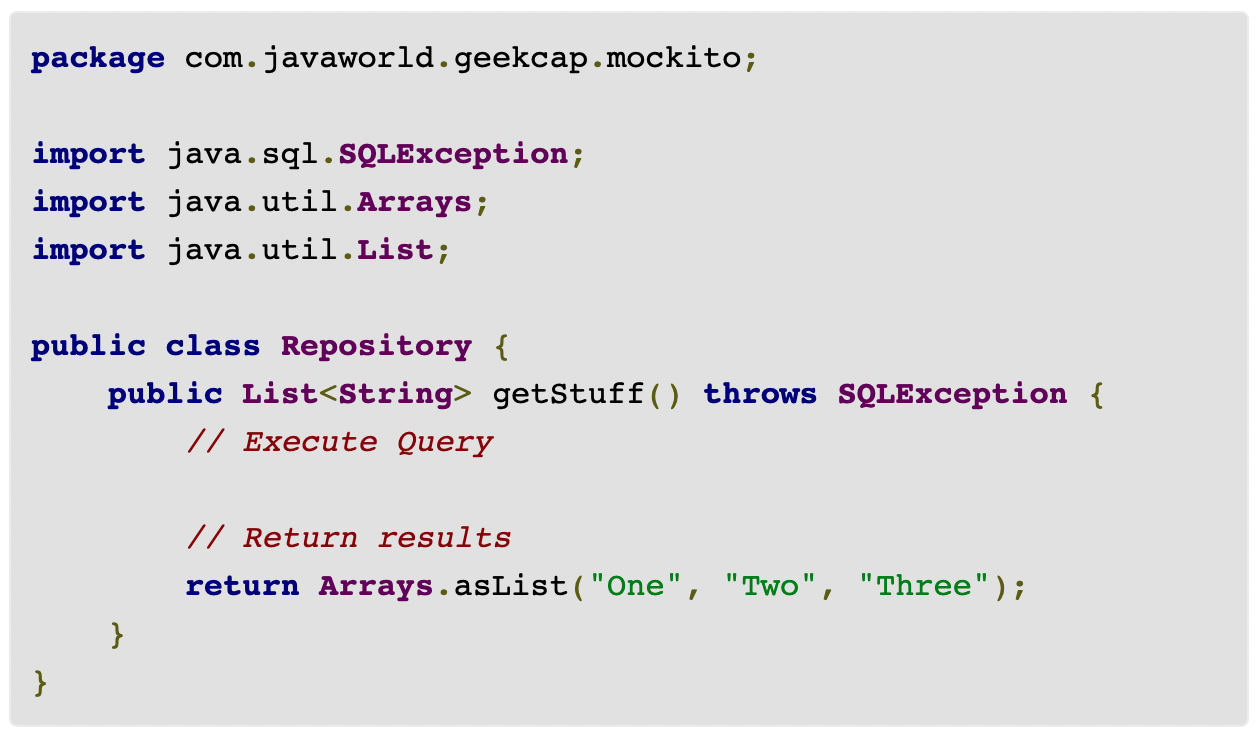
Finally, you can **add these same groups and excludedGroups** fields to the **surefire plug-in in your Maven POM file**. You can also **control these fields using Maven profiles**.

**Introduction to Mock objects using Mockito**

**Thus far we have only reviewed testing simple methods that do not rely on external dependencies, but this is far from normal for large applications**. For example, a business service probably relies on either a database or web service call to retrieve the data that it operates on. So how would we test a method in such a class? And how would we simulate problematic conditions, such as a database connection error or timeout?

The **strategy of mock objects** is to **analyze the class under test** and **create mock versions** of all of **its dependencies**, **creating the scenarios** **that we want to test**. You can do this manually—which is a lot of work—or you could leverage a tool like **Mockito**, which **simplifies the creation and injection of mock objects** into your classes. **Mockito provides a simple API** to **create mock implementations of your dependent classes**, **inject the mocks into your classes**, and **control the behavior of the mocks**.

The example in Listing 9 shows the source code for a simple **repository**.



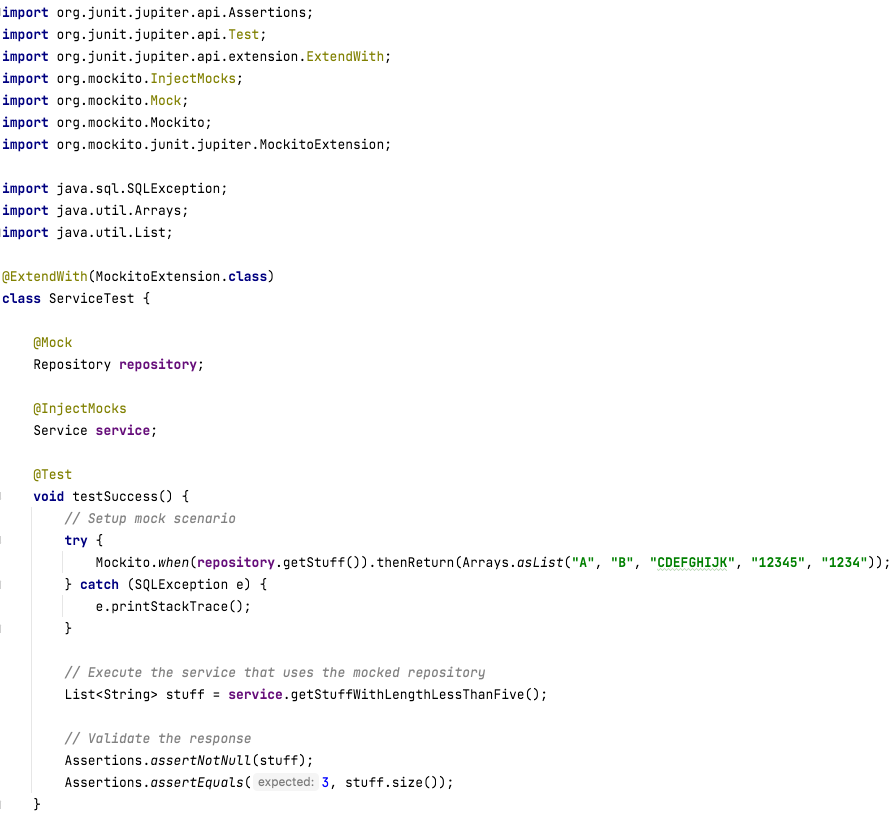
Listing 10 shows the source code for a **service** that **uses the above repository**.



The **Repository** in Listing 9 has a single method, **getStuff**, that would **presumably connect to a database, execute a query, and return the results**. In this example, it simply returns a list of three Strings. The Service in Listing 10 receives the Repository through its constructor and defines a single method, **getStuffWithLengthLessThanFive**, which returns all Strings with a length less than 5. If the **repository throws a SQLException** then it **returns an empty list**.

**Unit testing with JUnit 5 and Mockito**

Now let's look at how we can test our service using JUnit 5 and Mockito. Listing 11 shows the source code for a **ServiceTest** class.





The first thing to notice about this test class is that it is **annotated with** **@ExtendWith(MockitoExtension.class)**. The **@ExtendWith annotation is used to load a JUnit 5 extension**. **JUnit defines an extension API**, which **allows a third-party vendor like Mockito** to **hook into the lifecycle of running test classes and add additional functionality**. The **MockitoExtension** looks at the test class, finds member variables annotated with the **@Mock annotation**, and **creates a mock implementation of those variables**. It then finds member variables annotated with the **@InjectMocks annotation** and **attempts to inject its mocks into those classes**, **using either construction injection or setter injection**.

In this example, **MockitoExtension finds the @Mock annotation on the Repository member variable, so it creates a mock implementation of it and assigns it to the repository variable**. When it sees the **@InjectMocks annotation on the Service member variable, it creates an instance of the Service class, passing the mock Repository to its constructor**. This **allows us to control the behavior of the mock Repository class using Mockito's APIs**.

In the **testSuccess** method, we **use the Mockito API** to **return a specific result set** when its getStuff method is called. The API works as follows: the **Mockito::when defines the condition**, which in this case is the invocation of the repository.getStuff() method. The **when() method returns a org.mockito.stubbing.OngoingStubbing instance**, which **defines a set of methods that determine what to do when the specified method is called**. In this case, we invoke the **thenReturn()** **method** to tell the stub to **return a specific List of Strings**.

At this point, we have a **Service instance** **with a mock Repository**. When the **Repository's getStuff method is called**, it **returns a list** of five **known** **strings**. We invoke the **Service's** **getStuffWithLengthLessThanFive() method**, which will invoke the **Repository's getStuff() method**, and **return a filtered list of Strings whose length is less than five**. We can then **assert that the returned list is not null and that the size of it is three**. **This process allows us to test the logic in the specific Service method, with a known response from the Repository**.

The **testException** **method** **configures Mockito** so that **when the Repository's getStuff() method is called**, it **throws an SQLException**. If this happens, the Service should not throw an exception; rather, it should return an empty list.

**Mockito** **is a powerful tool** and we've only scratched the surface of what it can do. If you've ever wondered how we can **test abhorrent conditions**—such as network, database, timeout, or other I/O error conditions—Mockito is the tool for us, and it works very elegantly with JUnit 5. If you do run into **situations that Mockito does not support**, **such as mocking static member variables or private constructors**, then there is another **powerful but complex tool called** [**PowerMock**](https://github.com/powermock/powermock).