Java Enterprise Edition

ANUDIP FOUNDATION

TDD with Junit 5

Objective:	**Materials Required:**
• Understand the fundamentals of TDD and JUnit5 • Use TDD to build, test, and package a Java library • Perform TDD by performing Red-Green-Refactor iterations • Learn how to use TDD to build, test, and package a Java library • Learn how to create unit tests using the JUnit5 testing framework	1. Eclipse/IntelliJ/STC 2. Maven
Theory:180mins	**Practical:60mins**
Total Duration: 240 Min	

· TDD with Junit 5
o Types of Tests
o Why Unit Tests Are Important
o What's JUnit?
o JUnit 5 Architecture
o IDEs and Build Tool Support
o Setting up JUnit with Maven
o Lifecycle Methods
o Test Hierarchies
o Assertions
o Disabling Tests
o Assumptions
o Test Interfaces and Default Methods
o Repeating Tests
o Dynamic Tests
o Parameterized Tests
o Argument Sources
o Argument Conversion
o What Is TDD?
o History of TDD
o Why Practice TDD?
o Types of Testing
o Testing Frameworks and Tools
o Testing Concepts
o Insights from Testing
o Mocking Concepts
o Mockito Overview
o Mockito Demo
o Creating Mock Instances
o Stubbing Method Calls

TYPES OF TESTS

Introduction:-

Testing is the process of executing a program with the aim of finding errors. To make our software perform well it should be error-free. If testing is done successfully it will remove all the errors from the software.

Principles of Testing:-

(i) All the test should meet the customer requirements
(ii) To make our software testing should be performed by a third party
(iii) Exhaustive testing is not possible. As we need the optimal amount of testing based on the risk assessment of the application.
(iv) All the test to be conducted should be planned before implementing it
(v) It follows the Pareto rule(80/20 rule) which states that 80% of errors come from 20% of program components.
(vi) Start testing with small parts and extend it to large parts.

Manual vs. automated testing

At a high level, we need to make the distinction between manual and automated tests. Manual testing is done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling. This is very expensive as it requires someone to set up an environment and execute the tests themselves, and it can be prone to human error as the tester might make typos or omit steps in the test script.

Automated tests, on the other hand, are performed by a machine that executes a test script that has been written in advance. These tests can vary a lot in complexity, from checking a single method in a class to making sure that performing a sequence of complex actions in the UI leads to the same results. It's much more robust and reliable than automated tests – but the quality of your automated tests depends on how well your test scripts have been written.

Automated testing is a key component of continuous integration and continuous delivery and it's a great way to scale your QA process as you add new features to your application. But there's still value in doing some manual testing with what is called exploratory testing as we will see in this guide.

Types of Testing:-

1. Unit Testing

It focuses on the smallest unit of software design. In this, we test an individual unit or group of interrelated units. It is often done by the programmer by using sample input and observing its corresponding outputs.
Example:

a) In a program we are checking if loop, method or

function is working fine

b) Misunderstood or incorrect, arithmetic precedence.

c) Incorrect initialization

2. Integration Testing

The objective is to take unit tested components and build a program structure that has been dictated by design. Integration testing is testing in which a group of components is combined to produce output.

Integration testing is of four types: (i) Top-down (ii) Bottom-up (iii) Sandwich (iv) Big-Bang
Example

(a) Black Box testing:- It is used for validation.

In this we ignore internal working mechanism and

focuse on **what is the output?**.

(b) White Box testing:- It is used for verification.
In this we focus on internal mechanism i.e.
how the output is achieved?

3. Regression Testing

Every time a new module is added leads to changes in the program. This type of testing makes sure that the whole component works properly even after adding components to the complete program.
Example

In school record suppose we have module staff, students and finance combining these modules and checking if on

integration these module works fine is regression testing

4. Smoke Testing

This test is done to make sure that software under testing is ready or stable for further testing
It is called a smoke test as the testing an initial pass is done to check if it did not catch the fire or smoke in the initial switch on.
Example:

If project has 2 modules so before going to module

make sure that module 1 works properly

5. Alpha Testing

This is a type of validation testing. It is a type of *acceptance testing* which is done before the product is released to customers. It is typically done by QA people.
Example:

When software testing is performed internally within

the organization

6. Beta Testing

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for a limited number of users for testing in a real-time environment
Example:

When software testing is performed for the limited

number of people

7. System Testing

This software is tested such that it works fine for the different operating systems. It is covered under the black box testing technique. In this, we just focus on the required input and output without focusing on internal working.
In this, we have security testing, recovery testing, stress testing, and performance testing
Example:

This include functional as well as non functional

testing

8. Stress Testing

In this, we give unfavorable conditions to the system and check how they perform in those conditions.
Example:

(a) Test cases that require maximum memory or other
resources are executed

(b) Test cases that may cause thrashing in a virtual
operating system

(c) Test cases that may cause excessive disk requirement

9. Performance Testing

It is designed to test the run-time performance of software within the context of an integrated system. It is used to test the speed and effectiveness of the program. It is also called load testing. In it we check, what is the performance of the system in the given load.
Example:

Checking number of processor cycles.

10. Object-Oriented Testing

This testing is a combination of various testing techniques that help to verify and validate object-oriented software. This testing is done in the following manner:

- Testing of Requirements,
- Design and Analysis of Testing,
- Testing of Code,
- Integration testing,
- System testing,
- User Testing.

Why Unit testing is important

As we write a lot about Agile, CI, and TDD, we had to mention unit testing. This time, we will talk about what unit testing is, why it is part of Agile methodology, and the main benefits of using it.

In computer programming, unit testing is a software testing method by which individual units of source code are tested to determine whether they are fit for use. A unit is the smallest possible testable software component. Usually, it performs a single cohesive function. A unit is small, so it is easier to design, execute, record, and analyze test results for than larger chunks of code are. Defects revealed by a unit test are easy to locate and relatively easy to repair.

The goal of unit testing is to segregate each part of the program and test that the individual parts are working correctly. It isolates the smallest piece of testable software from the remainder of the code and determines whether it behaves exactly as you expect. Unit testing has proven its value in that a large percentage of defects are identified during its use. It allows automation of the testing process, reduces difficulties of discovering errors contained in more complex pieces of the application, and enhances test coverage because attention is given to each unit.

For example, if you have two units and decide it would be more cost-effective to glue them together and initially test them as an integrated unit, an error could occur in a variety of places. Is the error in unit 1 or in unit 2? Is the error in both units? Is the error in the interface between the units? Is the error because of a defect in the test?

As you can see, finding the error in the integrated module is much more complicated than first isolating the units, testing each, then integrating them and testing the whole.

At Apiumhub, we work using Agile methodology, and we work a lot with unit tests. Unit testing is a signature of Extreme Programming (XP), another agile software development methodology we use quite often, which led quickly to test-driven development. We strongly believe that being Agile is doing CI and TDD. Using test-driven development, developers create unit tests as they develop their code so that each unit test tests a tiny piece of software code usually before the code is written.

Unit testing provides numerous benefits including finding software bugs early, facilitating change, simplifying integration, providing a source of documentation, and many others, which we will look right now with more detail.

1. Makes the Process Agile

One of the main benefits of unit testing is that it makes the coding process more Agile. When you add more and more features to a software, you sometimes need to change old design and code. However, changing already-tested code is both risky and costly. If we have unit tests in place, then we can proceed for refactoring confidently.

Unit testing really goes hand-in-hand with agile programming of all flavors because it builds in tests that allow you to make changes more easily. In other words, unit tests facilitate safe refactoring.

2. Quality of Code

Unit testing improves the quality of the code. It identifies every defect that may have come up before code is sent further for integration testing. Writing tests before actual coding makes you think harder about the problem. It exposes the edge cases and makes you write better code.

3. Finds Software Bugs Early

Issues are found at an early stage. Since unit testing is carried out by developers who test individual code before integration, issues can be found very early and can be resolved then and there without impacting the other pieces of the code. This includes both bugs in the programmer's implementation and flaws or missing parts of the specification for the unit.

4. Facilitates Changes and Simplifies Integration

Unit testing allows the programmer to refactor code or upgrade system libraries at a later date and make sure the module still works correctly. Unit tests detect changes that may break a design contract. They help with maintaining and changing the code.

Unit testing reduces defects in the newly developed features or reduces bugs when changing the existing functionality.

Unit testing verifies the accuracy of the each unit. Afterward, the units are integrated into an application by testing parts of the application via unit testing. Later testing of the application during the integration process is easier due to the verification of the individual units.

5. Provides Documentation

Unit testing provides documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's interface (API).

6. Debugging Process

Unit testing helps simplify the debugging process. If a test fails, then only the latest changes made in the code need to be debugged.

7. Design

Writing the test first forces you to think through your design and what it must accomplish before you write the code. This not only keeps you focused; it makes you create better designs. Testing a piece of code forces you to define what that code is responsible for. If you can do this easily, that means the code's responsibility is well-defined and therefore that it has high cohesion.

8. Reduce Costs

Since the bugs are found early, unit testing helps reduce the cost of bug fixes. Just imagine the cost of a bug found during the later stages of development, like during system testing or during acceptance testing. Of course, bugs detected earlier are easier to fix because bugs detected later are usually the result of many changes, and you don't really know which one caused the bug.

What is JUnit 5?

Unlike previous versions of JUnit, JUnit 5 is composed of several different modules from three different sub-projects.

JUnit 5 = *JUnit Platform + JUnit Jupiter + JUnit Vintage*

The **JUnit Platform** serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform. Furthermore, the platform provides a Console Launcher to launch the platform from the command line and a JUnit 4 based Runner for running any TestEngine on the platform in a JUnit 4 based environment. First-class support for the JUnit Platform also exists in popular IDEs (see IntelliJ IDEA, Eclipse, NetBeans, and Visual Studio Code) and build tools (see Gradle, Maven, and Ant).

JUnit Jupiter is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.

JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

JUnit 5 Architecture

JUnit 5 architecture

It's time for a new approach. It hasn't come instantly; it required reflection, and the shortcomings of JUnit 4 are a good input for the needed improvements. Architects know the problems, and they decided to go on the path of reduced sizes and modularity.

JUnit 5 modularity

A new approach, a modular one, was necessary in order to allow the evolution of the JUnit framework. Its architecture had to allow JUnit to interact with different programmatic clients, with different tools and IDEs. The logical separation of concerns required:

- An API to write tests, dedicated mainly to the developers.
- A mechanism for discovering and running the tests.
- An API to allow the easy interaction with IDEs and tools and to run the tests from them.

As a consequence, the resulting JUnit 5 architecture contained three modules (fig. 1):

- JUnit Platform, which serves as a foundation for launching testing frameworks on the JVM (Java Virtual Machine), also provides an API to launch tests from either the console, IDEs, or build tools.
- JUnit Jupiter, the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. The name has been chosen from the fifth planet of our Solar System, which is also the largest one.
- JUnit Vintage, a test engine for running JUnit 3 and JUnit 4 based tests on the platform, ensuring the necessary backwards compatibility.

JUnit Platform
JUnit Jupiter
JUnit Vintage

Figure 1 The modular architecture of JUnit 5

JUnit 5 Platform

Going further with the modularity idea, we'll have a brief look at the artifacts contained into the JUnit 5 Platform (fig. 2):

- junit-platform-commons, an internal common library of JUnit, intended solely for usage within the JUnit framework itself. Any usage by external parties isn't supported.
- junit-platform-console, which provides support for discovering and executing tests on the JUnit Platform from the console.
- junit-platform-console-standalone an executable JAR with all dependencies included. It's used by Console Launcher, a command-line Java application that lets you launch the JUnit Platform from the

console. For example, it can be used to run JUnit Vintage and JUnit Jupiter tests and print test execution results to the console.

- junit-platform-engine, a public API for test engines.
- junit-platform-launcher, a public API for configuring and launching test plans, typically used by IDEs and build tools.
- junit-platform-runner, a runner for executing tests and test suites on the JUnit Platform in a JUnit 4 environment.
- junit-platform-suite-api, which contains the annotations for configuring test suites on the JUnit Platform.
- junit-platform-surefire-provider, which provides support for discovering and executing tests on the JUnit Platform using Maven Surefire.
- junit-platform-gradle-plugin, which provides support for discovering and executing tests on the JUnit Platform using Gradle.

JUnit 5 Jupiter

JUnit Jupiter is the combination of the new programming model (annotations, classes, methods) and extension model for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform. In contrast to the previously existing runners and rules extension points in JUnit 4, the JUnit Jupiter extension model consists of a single, coherent concept: the Extension API.

The artifacts contained into the JUnit Jupiter are:

- junit-jupiter-api, the JUnit Jupiter API for writing tests and extensions.
- junit-jupiter-engine, the JUnit Jupiter test engine implementation, only required at runtime.
- junit-jupiter-params, which provides support for parameterized tests in JUnit Jupiter.
- junit-jupiter-migrationsupport, which provides migration support from JUnit 4 to JUnit Jupiter, and it’s required only for running selected JUnit 4 rules.

JUnit 5 Vintage

JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform. JUnit 5 Vintage only contains junit-vintage-engine, the engine implementation to execute tests written in JUnit 3 or 4. For this you also need the JUnit 3 or 4 JARs.

This is useful in order to interact with the old tests through JUnit 5. It's likely that you may need to work on your projects with JUnit 5, but still support many old tests. JUnit 5 Vintage is the solution for this situation!

The big picture of the JUnit 5 architecture

To put everything head to head and show how the full architecture works, we'll say that JUnit Platform provides the facilities to run different kinds of tests: JUnit 5 tests, old JUnit 3 and 4 tests, third-party tests (figure 2).

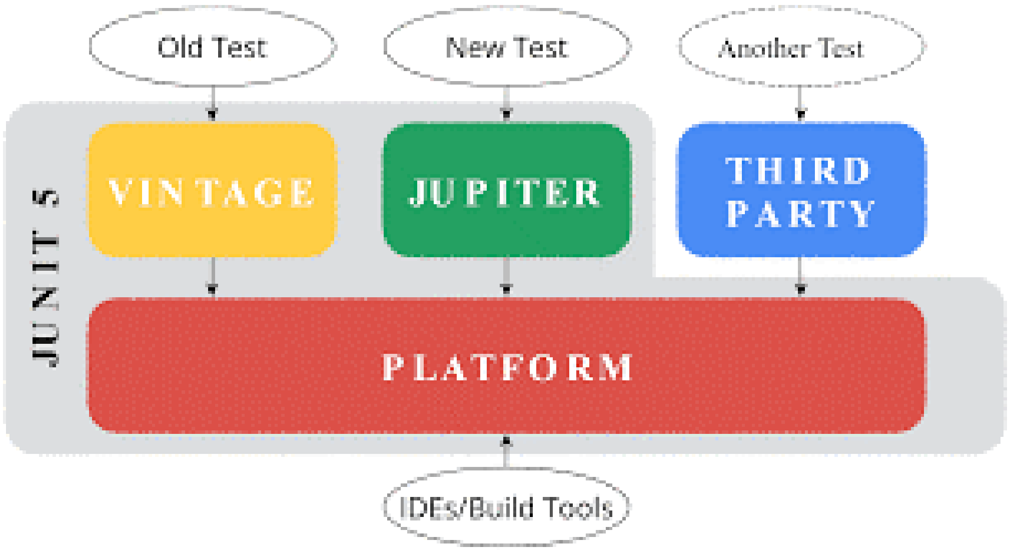


Figure 2 The big picture of the JUnit 5 architecture.

In more detail (figure 3):

- The test APIs provide the facilities for different test engines: junit-jupiter-api for JUnit 5 tests; junit-4.12 for legacy tests; custom engines for third-party tests.
- The test engines mentioned above are created by extending the junit-platform-engine public API, part of the JUnit 5 Platform.
- The junit-platform-launcher public API provide the facilities to discover tests inside the JUnit 5 Platform, for build tools like Maven or Gradle or for IDEs.

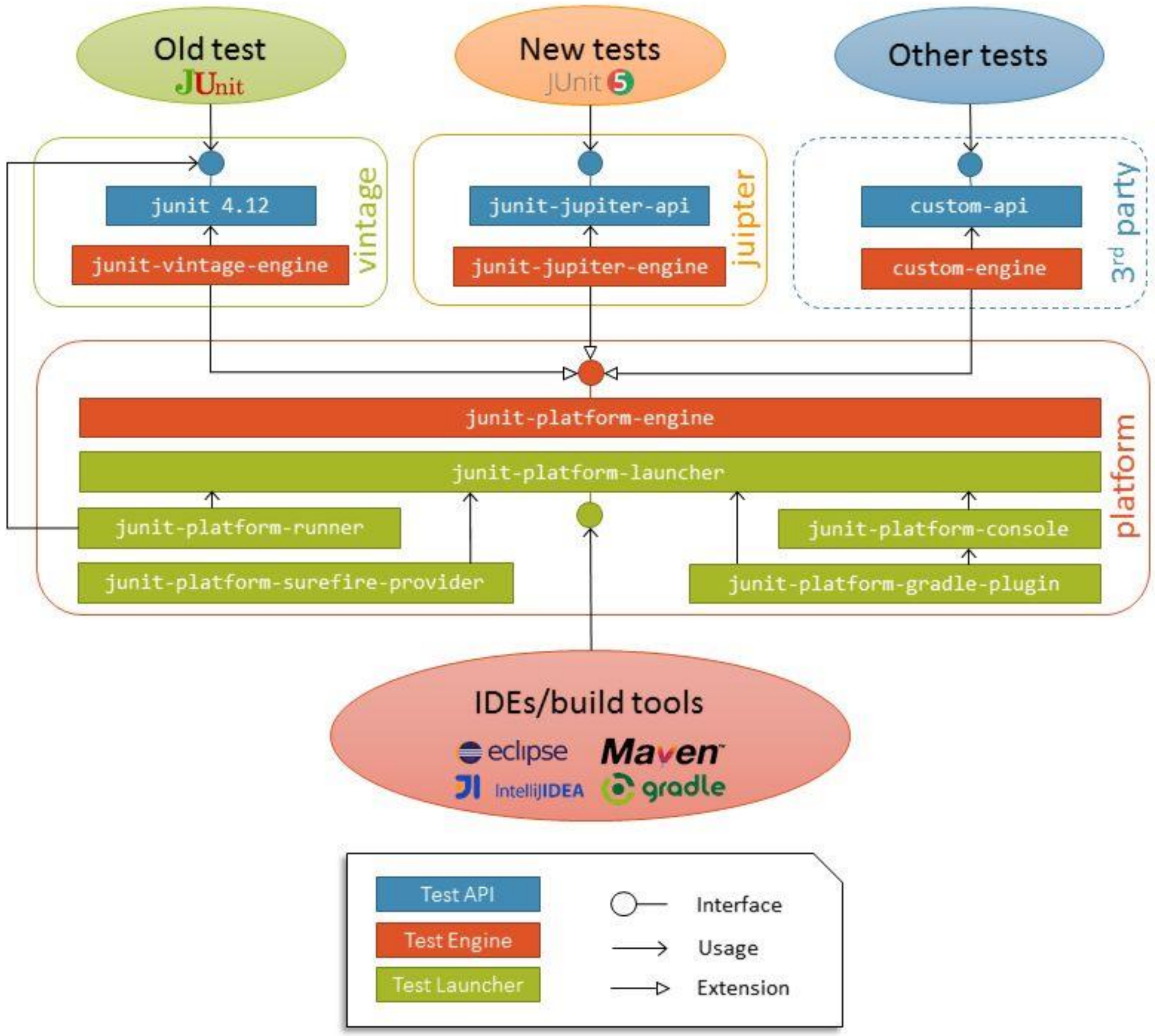


Figure 3 The detailed picture of the JUnit 5 architecture.

Besides the modular architecture, JUnit 5 also provides the extensions mechanism.

We underline the fact that the architecture of a system strongly determines its capabilities and its behavior. Understanding the architecture of both JUnit 4 and JUnit 5 helps you easily apply their capabilities in practice, write efficient tests, and analyze the implementation alternatives. They help you fasten the pace at which you gain the skills of a programmer who masters unit testing.

Rules vs the extension model

In order to put face-to-face the rules model of JUnit 4 and the extension model of JUnit 5, let's use an extended Calculator class (listing 1). It's used by the developers at *Test It Inc.* to execute mathematical operations, from verifying their systems under test. They're interested in testing the methods that may throw exceptions. One rule which has been extensively used by the *Test It Inc* tests code is ExpectedException. It can be easily replaced by the JUnit 5 assertThrows method.

Listing 1 The extended Calculator class

```
public class Calculator {

   …

   public double sqrt(double x) {                              (1)
      if (x < 0) {
         throw new
           IllegalArgumentException("Cannot extract the square    (2)
                              root of a negative value");  (2)
      }
```

```
        return Math.sqrt(x);                                    (1)
    }

    public double divide(double x, double y) {                      (3)
        if (y == 0) {
            throw new ArithmeticException("Cannot divide by zero");   (4)
        }
        return x/y;                                     (3)
    }
}
```

The logic that may throw exceptions into the Calculator class does the following:

1. Declares a method to calculate the square root of a number (1). In case the number is negative, an exception containing a particular message is created and thrown (2).
2. Declares a method to divide two numbers (3). In case the second number is zero, an exception containing a particular message is created and thrown (4).

Listing 2 provides an example that specifies which exception message is expected during the execution of the test code using the new functionality of the Calculator class above.

Listing 2 The JUnit4RuleExceptionTester class

```
public class JUnit4RuleExceptionTester {

    @Rule                                           (1)

    public ExpectedException expectedException =              (1)

                            ExpectedException.none();       (1)

     private Calculator calculator = new Calculator();          (2)

    @Test
    public void expectIllegalArgumentException() {
        expectedException.expect(IllegalArgumentException.class);     (3)
        expectedException.expectMessage("Cannot extract the square root (4)
                                of a negative value");        (4)
        calculator.sqrt(-1);                              (5)
    }

    @Test
    public void expectArithmeticException() {
        expectedException.expect(ArithmeticException.class);         (6)
        expectedException.expectMessage("Cannot divide by zero");      (7)
        calculator.divide(1, 0);                                    (8)
    }
}
```

Into the previous JUnit 4 example, we do the following:

1. We declare an ExpectedException field annotated with @Rule. The @Rule annotation must be applied either on a public non-static field or on a public non-static method (1).
   The ExpectedException.none() factory method creates an unconfigured ExpectedException.
2. We initialize an instance of the Calculator class whose functionality we're testing (2).
3. The ExpectedException is configured to keep the type of exception (3) and the message (4), before being thrown by invoking the sqrt method at line (5).
4. The ExpectedException is configured to keep the type of exception (6) and the message (7), before being thrown by invoking the divide method at line (8).

Now, we move our attention to the new JUnit 5 approach.

Listing 3 The JUnit5ExceptionTester class

```
public class JUnit5ExceptionTester {

    private Calculator calculator = new Calculator();                 (1')



    @Test

    public void expectIllegalArgumentException() {

        assertThrows(IllegalArgumentException.class,                  (2')
                     () -> calculator.sqrt(-1));              (2')
    }

    @Test
    public void expectArithmeticException() {
        assertThrows(ArithmeticException.class,                    (3')
                     () -> calculator.divide(1, 0));          (3')
    }
}
```

Into this JUnit 5 example, we do the following:

1. We initialize an instance of the Calculator class whose functionality we're testing (1').

2. We assert that the execution of the supplied calculator.sqrt(-1) executable throws an IllegalArgumentException (2').
3. We assert that the execution of the supplied calculator.divide(1, 0) executable throws an ArithmeticException (3').

We remark the clear difference in code clarity and code length between JUnit 4 and JUnit 5. The effective testing JUnit 5 code is 13 lines, the effective JUnit 4 code is twenty lines. We don't need to initialize and manage any additional rule. The testing JUnit 5 methods contain one line each.

Another largely used rule that *Test It Inc* would like to migrate is TemporaryFolder.

The TemporaryFolder rule allows the creation of files and folders that should be deleted when the test method finishes (whether it passes or fails). As the tests of the *Test It Inc* projects work intensively with temporary resources, this step is required. The JUnit 4 rule has been replaced with the @TempDir annotation in JUnit 5. Listing 4 presents the JUnit 4 approach.

Listing 4 The JUnit4RuleTester class

```
public class JUnit4RuleTester {

    @Rule

    public TemporaryFolder folder = new TemporaryFolder();              (1)



    @Test

    public void testTemporaryFolder() throws IOException {

        File createdFolder = folder.newFolder("createdFolder");         (2)

        File createdFile = folder.newFile("createdFile.txt");           (2)

        assertTrue(createdFolder.exists());                             (3)
        assertTrue(createdFile.exists());                               (3)
    }
}
```

Into this example, we do the following:

1. We declare a TemporaryFolder field annotated with @Rule and initialize it. The @Rule annotation must be applied either on a public field or on a public method (1).
2. We use the TemporaryFolder field to create a folder and a file (2). These ones are to be found into the Temp folder of your user profile into the operating system.
3. We check the existence of the temporary folder and of the temporary file (3).

Now, we move our attention to the new JUnit 5 approach (listing 5).

Listing 5 The JUnit5TempDirTester class

```
public class JUnit5TempDirTester {

    @TempDir                                          (1')

    Path tempDir;                                     (1')



    @Test

    public void testTemporaryFolder() throws IOException {

        assertTrue(Files.isDirectory(tempDir));                 (2')
        Path createdFile = Files.createFile(                (3')
                tempDir.resolve("createdFile.txt")             (3')
        );                                          (3')

        assertTrue(createdFile.toFile().exists());            (3')

    }
}
```

Into the previous JUnit 5 example, we do the following:

1. We declare a @TempDir annotated field (1’).
2. We check the creation of this temporary directory before the execution of the test (2’).
3. We create a file within this directory and check its existence (3’).

The advantage of the JUnit 5 extension approach is that we don’t have to create the folder by ourselves through a constructor, but the folder is automatically created once we annotate a field with @TempDir.

We move our attention to replacing our own custom rule. *Test It Inc.* has defined some own rules for its tests. This is particularly useful when some types of tests need similar additional actions before and after their execution.

In JUnit 4, the *Test It Inc.* engineers needed their additional own actions to be executed before and after the execution of a test. Consequently, they have created their own classes which implement the TestRule interface. To do this, one has to override the apply(Statement, Description) method which returns an instance of Statement. Such an object represents the tests within the JUnit runtime and Statement#evaluate()runs them. The Description object describes the individual test. We can use it to read information about the test through reflection.

Listing 6 The CustomRule class

```
public class CustomRule implements TestRule {                        (1)
    private Statement base;                                          (2)
    private Description description;                                 (2)

    @Override
    public Statement apply(Statement base, Description description) {
        this.base = base;                                            (3)
        this.description = description;                              (3)
        return new CustomStatement(base, description);               (3)
    }

}
```

To clearly show how to define our own rules, we look at listing 6, where we do the following:

1. We declare our CustomRule class that implements the TestRule interface (1).
2. We keep references to a Statement field and to a Description field (2) and we use them into the apply method that returns a CustomStatement (3).

Listing 7 The CustomStatement class

```
public class CustomStatement extends Statement {                        (1)

    private Statement base;                                         (2)

    private Description description;                                 (2)



    public CustomStatement(Statement base, Description description) {

        this.base = base;                                       (3)

        this.description = description;                              (3)

    }



    @Override                                                  (4)

    public void evaluate() throws Throwable {                          (4)

        System.out.println(this.getClass().getSimpleName() + " " +      (4)
                description.getMethodName() + " has started" );      (4)
        try {                                                  (4)
            base.evaluate();                                       (4)
        } finally {                                               (4)
            System.out.println(this.getClass().getSimpleName() + " " +  (4)
                description.getMethodName() + " has finished");      (4)
        }                                                      (4)
    }                                                          (4)
}
```

Into listing 7, we do the following:

1. We declare our CustomStatement class that extends the Statement class (1).
2. We keep references to a Statement field and to a Description field (2) and we use them as arguments of the constructor (3).
3. We override the inherited evaluate method and call base.evaluate() inside it (4).

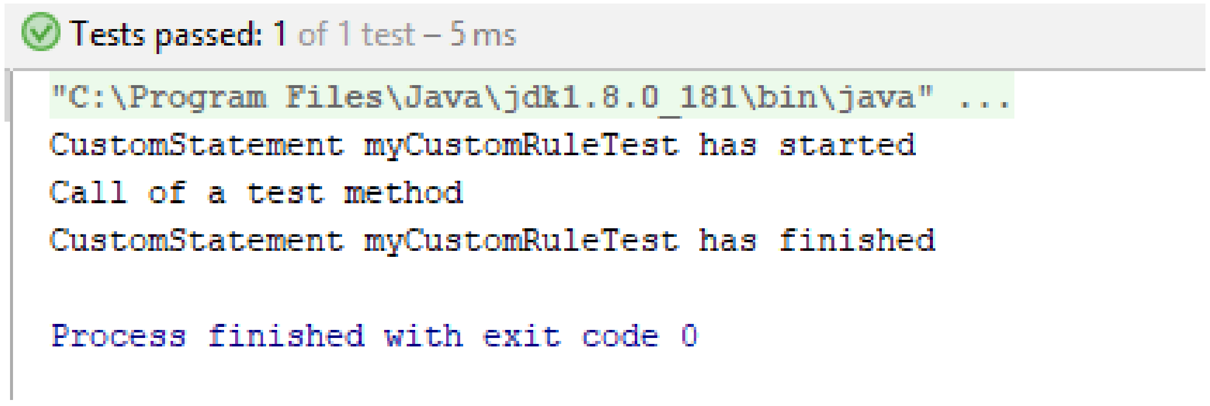
Listing 8 The JUnit4CustomRuleTester class

```
public class JUnit4CustomRuleTester {



    @Rule                                                   (1)

    public CustomRule myRule = new CustomRule();                    (1)



    @Test                                                   (2)

    public void myCustomRuleTest() {                        (2)

        System.out.println("Call of a test method");
    }
}
```

Into listing 8, we use the previously defined CustomRule by doing the following:

1. We declare a public CustomRule field and we annotate it with @Rule (1).
2. We create the myCustomRuleTest method and annotate it with @Test (2).

The result of the execution of this test is shown in figure 1. As the engineers from *Test It Inc.* needed, the effective execution of the test is surrounded by the additional messages provided into the evaluate method of the CustomStatement class.



```
Tests passed: 1 of 1 test – 5 ms
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...
CustomStatement myCustomRuleTest has started
Call of a test method
CustomStatement myCustomRuleTest has finished

Process finished with exit code 0
```

Figure 4 The result of the execution of JUnit4CustomRuleTester.

We now turn our attention to the JUnit 5 approach. The engineers from *Test It Inc* would like to migrate their own rules as well. JUnit 5 allows similar effects as in the case of the JUnit 4 rules by introducing the own extensions. The code is shorter and it relies on the declarative annotations style. We first define the CustomExtension class, which is used as an argument of the @ExtendWith annotation on the tested class.

Listing 9 The CustomExtension class

```
 public class CustomExtension implements AfterEachCallback,          (1') BeforeEachCallback
{                                            (1')

    @Override                                             (2')

    public void afterEach(ExtensionContext extensionContext)         (2')

                                    throws Exception {         (2')

       System.out.println(this.getClass().getSimpleName() + " " +     (2')
            extensionContext.getDisplayName() + " has started" );   (2')
    }

    @Override                                             (3')
    public void beforeEach(ExtensionContext extensionContext)         (3')
                                     throws Exception {         (3')
       System.out.println(this.getClass().getSimpleName() + " " +     (3')
            extensionContext.getDisplayName() + " has finished");   (3')
    }
 }
```

In listing 9, we do the following:

1. We declare CustomExtension as implementing the AfterEachCallback and BeforeEachCallback interfaces (1').
2. We override the afterEach method, to be executed after each test method from the testing class which is extended with CustomExtension (2').
3. We override the beforeEach method, to be executed before each test method from the testing class is extended with CustomExtension (3').

Listing 10 The JUnit5CustomExtensionTester class

```
@ExtendWith(CustomExtension.class)                              (1’)

public class JUnit5CustomExtensionTester {



    @Test                                               (2’)

    public void myCustomRuleTest() {                            (2’)

        System.out.println("Call of a test method");                (2’)
    }
}
```

In listing 10, we do the following:

1. We extend JUnit5CustomExtensionTester with the CustomExtension class (1’).
2. We create the myCustomRuleTest method and annotate it with @Test (2).

As the test class is extended with the CustomExtension class, the previously defined beforeEach and afterEach methods are executed before and after each test method respectively. We remark the clear difference in code clarity and code length between JUnit 4 and JUnit 5. The JUnit 4 approach needs to work with three classes, the JUnit 5 approach needs to work with only two classes. The code to be executed before and after each test method is isolated into a dedicated method with a clear name. On the side of the testing class, you only need to annotate it with @ExtendWith.

The JUnit 5 extension model may also be used to gradually replace the runners from JUnit 4. For the extensions which have already been created, the migration process is simple. For example:

- To migrate the Mockito tests, you need to replace, on the tested class, the annotation @RunWith(MockitoJUnitRunner.class) with the annotation @ExtendWith(MockitoExtension.class).
- To migrate the Spring tests, you need to replace, on the tested class, the annotation @RunWith(SpringJUnit4ClassRunner.class) with the annotation @ExtendWith(SpringExtension.class).

IDEs and Build Tool Support

IDE Support

Eclipse and IntelliJ natively support JUnit 5, but for NetBeans I couldn't even find an issue.

❑IntelliJ IDEA

IntelliJ IDEA supports JUnit 5 since 2016.2, but I strongly recommend to use at least 2017.3. Until then, IntelliJ used to come with its own version of the Jupiter engine, which leads to problems if your project does not depend on the matching API version. Since 2017.3, IntelliJ selects the engine based on the API version you depend on.

❑Eclipse

Eclipse supports JUnit 5 since Oxygen.1a (4.7.1a), but I didn't figure out how it picks up the engine.

Build Tool Support

Initially, the JUnit 5 team implemented a rudimentary Gradle plugin and Maven Surefire provider as proofs of concept. In the meantime, both tools have implemented native support, so there's no need to use junit-platform-gradle-plugin or junit-platform-surefire-provider anymore - you can remove them.

❑Gradle

Native JUnit 5 support is available since Gradle 4.6. All you need to do is activate it in the test task:

```
test {
        useJUnitPlatform()
}
```

As I explained, you need the engine at test run time, so the tests can actually be executed:

```
testRuntime "org.junit.jupiter:junit-jupiter-engine:5.2.0"
```

For more details on the Gradle integration, check its documentation.

❑Maven

Maven's surefire provider has native support for JUnit 5 since version 2.22.0. It picks up the test engine from your regular dependencies:

```
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.2.0</version>
        <scope>test</scope>
</dependency>
```

Setting Up Junit With MAVEN

Configuring JUnit Platform

To get started with JUnit Platform, you need to add at least a single `TestEngine` implementation to your project. For example, if you want to write tests with Jupiter, add the test artifact `junit-jupiter-engine` to the dependencies in POM:

```
<dependencies>
    [...]
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
    </dependency>
    [...]
</dependencies>
```

This will pull in all required dependencies. Among those dependencies is `junit-jupiter-api` which contains the classes and interfaces your test source requires to compile. `junit-platform-engine` is also resolved and added.

This is the only step that is required to get started - you can now create tests in your test source directory (e.g., `src/test/java`).

If you want to write and execute JUnit 3 or 4 tests via the JUnit Platform add the Vintage Engine to the dependencies, which transitively pulls in (and requires) `junit:junit:4.12`:

```
<dependencies>
    [...]
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
    </dependency>
    [...]
</dependencies>
```

For more information on using JUnit 5, that is the JUnit Platform, JUnit Jupiter, and JUnit Vintage, see the JUnit 5 web site and the JUnit 5 User Guide.

Smart Resolution of Jupiter Engine and Vintage Engine for JUnit4

JUnit5 API artifact and your test sources become isolated from engine. In these chapters you will see how you can segregate, combine, select the APIs and Engines miscellaneous way. You can find integration tests with JUnit4/5, with JUnit5/TestNG and with the JUnit4 Runner for Jupiter tests. (See the Maven profiles.)

How to run only one API

Normally, the developer does not want to access internal classes of JUnit5 engine (e.g. `5.4.0`). In the next chapters you can find your way to use the Jupiter or JUnit5 API where the plugin would resolve the engine.

Jupiter API in test dependencies

In this example the POM has only Jupiter API dependency in test classpath. The plugin will resolve and download the `junit-jupiter-engine` with the version related to the version of `junit-jupiter-api`. Similar principles can be found in the following chapters as well.

```
<dependencies>
    [...]
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
    </dependency>
    [...]
</dependencies>
...
<build>
    <plugins>
        [...]
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            [... configuration or goals and executions ...]
        </plugin>
        [...]
    </plugins>
```

```
</build>
...
```

API-Engine versions segregation

In the following example the engine artifact appears in plugin dependencies and the engine is resolved by the plugin and downloaded from a remote repository for plugins. You may want to update the version of engine with fixed bugs in `5.3.2` but the API version `5.3.0` stays intact!

```
<dependencies>
    [...]
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.3.0</version>
        <scope>test</scope>
    </dependency>
    [...]
</dependencies>
...
<build>
    <plugins>
        [...]
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>5.3.2</version>
                </dependency>
            </dependencies>
        </plugin>
        [...]
    </plugins>
</build>
...
```

JUnit4 API in test dependencies

This is similar example with JUnit4 in test dependencies of your project POM. The Vintage engine artifact has to be in the plugin dependencies; otherwise the plugin would use `surefire-junit4` provider instead of the `surefire-junit-platform` provider.

```
<dependencies>
    [...]
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
    [...]
</dependencies>
...
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                    <version>5.4.0</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
...
```

How to run multiple APIs or Engines

In the following example you can use both JUnit4 and JUnit5 tests.

Jupiter API and JUnit4

Once you define any JUnit5 API in the dependencies, the provider `surefire-junit-platform` is selected and you can always add the JUnit4 dependency.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Jupiter API and Vintage Engine

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Jupiter and Vintage Engine

```
<dependencies>
    <dependency>
```

```
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Select engine and use multiple APIs

In these examples you use both API, i.e. Jupiter and JUnit4, in the test dependencies but you want to select the engine via plugin dependencies.

Select Jupiter

Here your tests import the packages from JUnit4 and Jupiter but you want to select only one Maven profile with JUnit4 tests.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<profile>
    <id>select-junit5</id>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <dependencies>
                    <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter-engine</artifactId>
                        <version>5.6.2</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</profile>
```

Select JUnit4

Here your tests import the packages from JUnit4 and Jupiter but you want to select only one Maven profile with Jupiter tests.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<profile>
    <id>select-junit4</id>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <dependencies>
                    <dependency>
                        <groupId>org.junit.vintage</groupId>
                        <artifactId>junit-vintage-engine</artifactId>
                        <version>5.6.2</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</profile>
```

How to run TestNG tests within Jupiter engine

You can run TestNG tests combined with JUnit5 tests.

For more information see this example.

```
<dependencies>
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.1.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.github.testng-team</groupId>
        <artifactId>testng-junit5</artifactId>
        <version>0.0.1</version>
        <scope>test</scope>
        <exclusions>
            <exclusion>
                <groupId>org.junit.platform</groupId>
                <artifactId>junit-platform-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The Maven does not take any responsibility for broken compatibilities in this case and the responsibility for the dependency `com.github.testng-team:testng-junit5`.

JUnit Runner

The JUnit4 library has the Runner implemented in the JUnit5's artifact `junit-platform-runner`.

For more information see this example.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.platform</groupId>
        <artifactId>junit-platform-runner</artifactId>
        <version>1.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Provider Selection

If nothing is configured, Surefire detects which JUnit version to use by the following algorithm:

```
if the JUnit 5 Platform Engine is present in the project
    use junit-platform
if the JUnit version in the project >= 4.7 and the <<<parallel>>> configuration parameter has ANY value
    use junit47 provider
if JUnit >= 4.0 is present
    use junit4 provider
else
    use junit3.8.1
```

When using this technique there is no check that the proper test-frameworks are present on your project's classpath. Failing to add the proper test-frameworks will result in a build failure.

Running Tests in Parallel

From JUnit Platform does not support running tests in parallel.

Running a Single Test Class

The JUnit Platform Provider supports the `test` JVM system property supported by the Maven Surefire Plugin. For example, to run only test methods in the `org.example.MyTest` test class you can execute `mvn -Dtest=org.example.MyTest test` from the command line.

Filtering by Test Class Names for Maven Surefire

The Maven Surefire Plugin will scan for test classes whose fully qualified names match the following patterns.

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

Moreover, it will exclude all nested classes (including static member classes) by default.

Note, however, that you can override this default behavior by configuring explicit `include` and `exclude` rules in your `pom.xml` file. For example, to keep Maven Surefire from excluding static member classes, you can override its exclude rules.

Overriding exclude rules of Maven Surefire

```
...
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <excludes>
                    <exclude>some test to exclude here</exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
...
```

Filtering by Tags

You can use JUnit5 Tags and filter tests by tags or tag expressions.

- To include `tags` or `tag expressions`, use `groups`.
- To exclude `tags` or `tag expressions`, use `excludedGroups`.

```
...
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <groups>acceptance | !feature-a</groups>
                <excludedGroups>integration, regression</excludedGroups>
            </configuration>
        </plugin>
    </plugins>
</build>
...
```

Configuration Parameters

You can set JUnit Platform configuration parameters to influence test discovery and execution by declaring the `configurationParameters` property and providing key-value pairs using the Java `Properties` file syntax (as shown below) or via the `junit-platform.properties` file.

```
...
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <properties>
                    <configurationParameters>
                        junit.jupiter.conditions.deactivate = *
                        junit.jupiter.extensions.autodetection.enabled = true
                        junit.jupiter.testinstance.lifecycle.default = per_class
                        junit.jupiter.execution.parallel.enabled = true
                    </configurationParameters>
                </properties>
            </configuration>
        </plugin>
    </plugins>
</build>
...
```

Surefire Extensions and Reports Configuration for @DisplayName

Since plugin version 3.0.0-M4 you can use fine grained configuration of reports and enable phrased names together with `@DisplayName` in you tests. This is the complete list of attributes of particular objects. You do not have to specify e.g. `disable`, `version` and `encoding`. The boolean values reach default values `false` if not specified otherwise.

```
...
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <configuration>
                <statelessTestsetReporter implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5Xml30StatelessReporter">
                    <disable>false</disable>
                    <version>3.0</version>
                    <usePhrasedFileName>false</usePhrasedFileName>
                    <usePhrasedTestSuiteClassName>true</usePhrasedTestSuiteClassName>
                    <usePhrasedTestCaseClassName>true</usePhrasedTestCaseClassName>
                    <usePhrasedTestCaseMethodName>true</usePhrasedTestCaseMethodName>
                </statelessTestsetReporter>
                <consoleOutputReporter implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5ConsoleOutputReporter">
                    <disable>false</disable>
                    <encoding>UTF-8</encoding>
                    <usePhrasedFileName>false</usePhrasedFileName>
                </consoleOutputReporter>
                <statelessTestsetInfoReporter implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5StatelessTestsetInfoReporter">
                    <disable>false</disable>
                    <usePhrasedFileName>false</usePhrasedFileName>
                    <usePhrasedClassNameInRunning>true</usePhrasedClassNameInRunning>
                    <usePhrasedClassNameInTestCaseSummary>true</usePhrasedClassNameInTestCaseSummary>
                </statelessTestsetInfoReporter>
            </configuration>
        </plugin>
    </plugins>
</build>
...
```

Default implementations of these extensions are `org.apache.maven.plugin.surefire.extensions.SurefireStatelessReporter`, `org.apache.maven.plugin.surefire.extensions.SurefireConsoleOutputReporter`, and `org.apache.maven.plugin.surefire.extensions.SurefireStatelessTestsetInfoReporter`.

The aim of extensions is to let the users customizing the default behavior. We are keen on listing useful extensions on Apache Maven Surefire site if you propagate your extensions on GitHub.

LIFECYCLE METHODS:

Lifecycle Methods

A lifecycle method is any method that is annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach` or `@AfterEach` annotation. The lifecycle methods execute before or after executing the actual test methods.

The @BeforeAll and @AfterAll annotations denote that the annotated method should be executed before or after all test methods in the test class.

Respectively, @BeforeEach and @AfterEach mean the annotated method should be executed before or after each test method in the test class.

If you have ten test methods in the test class, @BeforeEach and @AfterEach will be executed ten times, but @BeforeAll and @AfterAll only once.

Test Instance Lifecycle

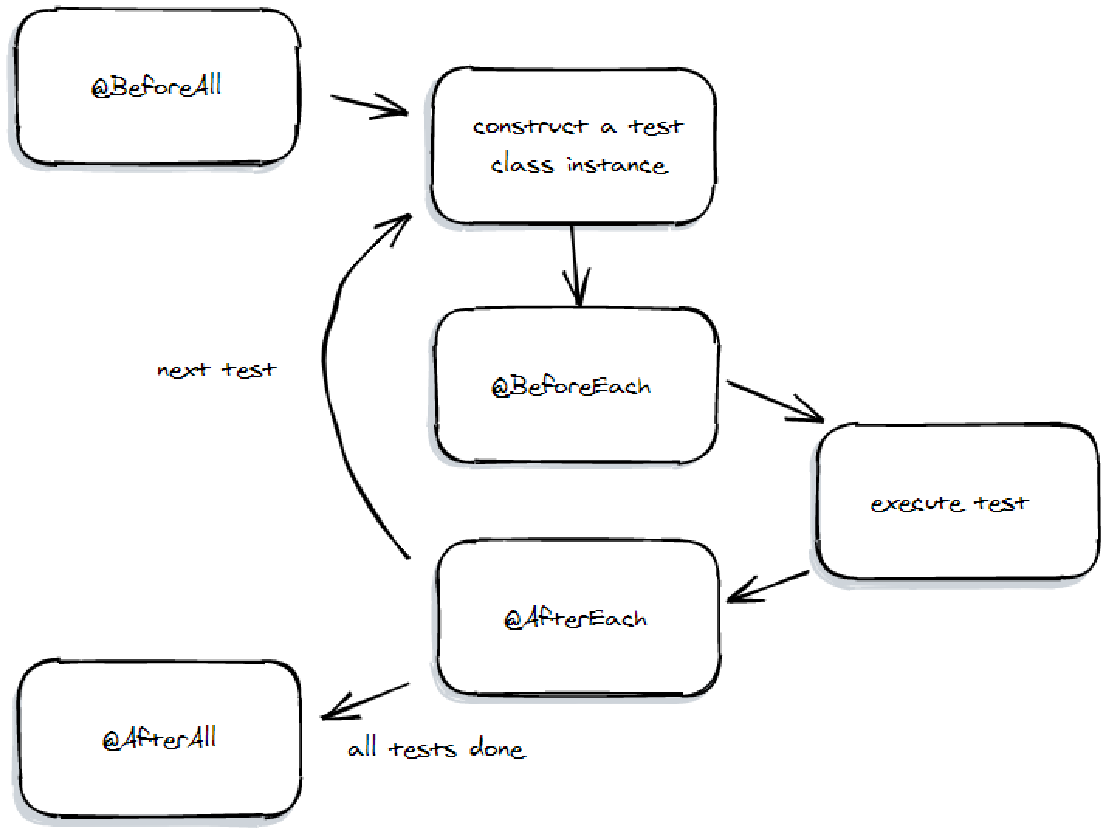
By default, JUnit creates a new instance of the test class before executing each test method. This helps us to run individual test methods in isolation and avoids unexpected side effects.

To see how this works, let's take a look at the following example:

```
class PerMethodLifecycleTest {
    public PerMethodLifecycleTest() {
        System.out.println("Constructor");
    }

    @BeforeAll
    static void beforeTheEntireTestFixture() {
        System.out.println("Before the entire test fixture");
    }

    @AfterAll
    static void afterTheEntireTestFixture() {
        System.out.println("After the entire test fixture");
    }

    @BeforeEach
    void beforeEachTest() {
        System.out.println("Before each test");
    }

    @AfterEach
    void afterEachTest() {
        System.out.println("After each test");
    }

    @Test
    void firstTest() {
        System.out.println("First test");
    }

    @Test
    void secondTest() {
        System.out.println("Second test");
    }
}
```

Notice how the methods annotated with @BeforeAll and @AfterAll are static methods. This is because when creating a new test instance per test method, there is no shared state otherwise.

The following illustration makes it easier to understand.



Per method lifecycle

Executing the tests in the test class gives us the following output (actually, the output has been formatted to make it more obvious):

```
Before the entire test fixture
  Constructor
    Before each test
      First test
    After each test
  Constructor
    Before each test
      Second test
    After each test
After the entire test fixture
```

Looking at the output, we can see that JUnit constructs the test class once per each test method. The lifecycle methods for the entire fixture have been executed once, and the lifecycle methods for tests have been executed multiple times.

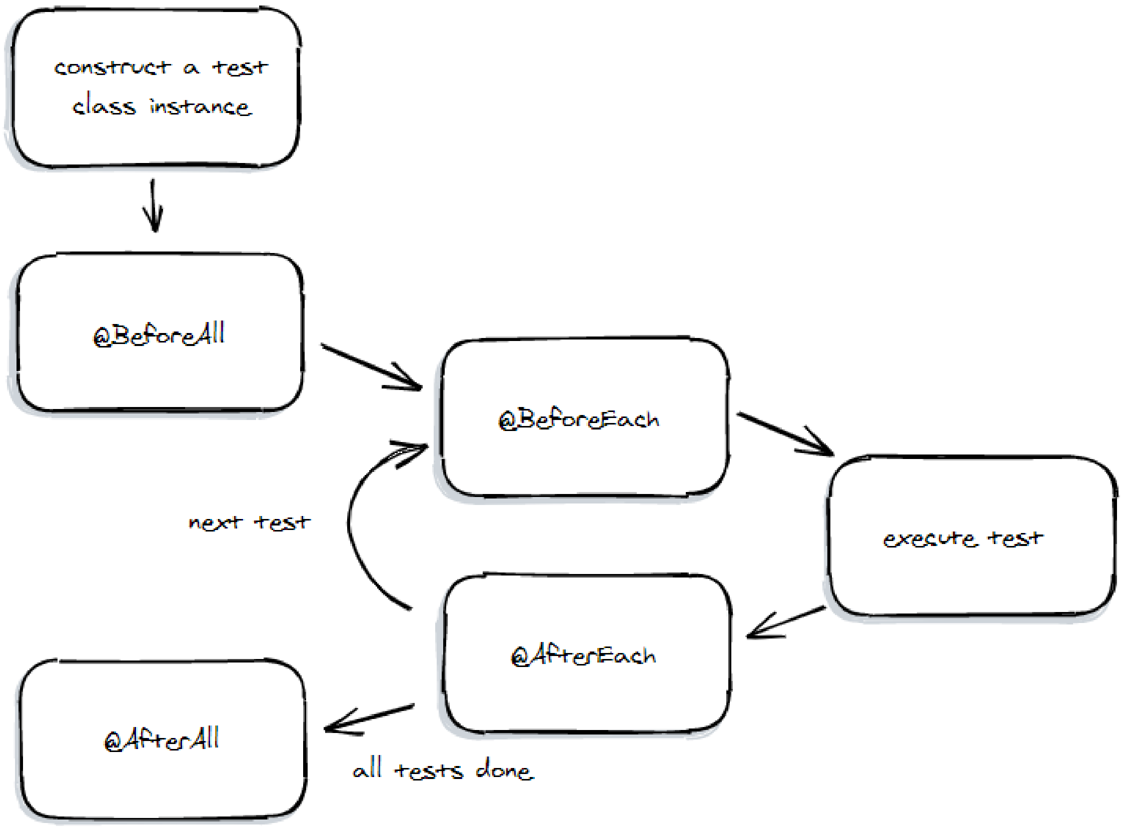
Test Instance Per Class

It is also possible to make JUnit execute all test methods on the same test instance. If we annotate the test class with @TestInstance(Lifecycle.PER_CLASS), JUnit will create a new test instance once per test class.

Because of the shared instance, there is now no need for the methods to be static:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class PerClassLifecycleTest {
   @BeforeAll
   void beforeTheEntireTestFixture() {
      System.out.println("Before the entire test fixture");
   }

   @AfterAll
   void afterTheEntireTestFixture() {
      System.out.println("After the entire test fixture");
   }

   // ...
}
```

Again, the following illustration helps to understand what is happening.



Per class lifecycle

Consequently, the output of the test execution is now a little different:

```
Constructor
  Before the entire test fixture
    Before each test
      First test
    After each test
    Before each test
      Second test
    After each test
  After the entire test fixture
```

From the results, we can see that the execution order of the lifecycle methods has not changed However, the difference is that JUnit constructs the test class only once.

The fundamental difference is that in the default lifecycle method constructing a new test class instance resets the state stored in instance variables. When the per class lifecycle method constructs the instance only once, state stored in instance variables is shared between tests.

If your test methods rely on state stored in instance variables, you may need to reset the state in @BeforeEach or @AfterEach lifecycle methods.

Try to avoid writing tests that rely on state stored in instance variables.

More Concrete Example

Now that we know how the lifecycle methods work, it’s good to explore how they can be used in practice. Usually, if we have something computationally expensive, we might want to share that with multiple tests.

Examples of this are opening a database connection, retrieving a context from a dependency injection framework, or reading a file.

In the following example, we start up a server only once per test instance:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class ExpensiveResourceTest {
    private JettyServer jettyServer;

    @BeforeAll
    void startServer() throws Exception {
        jettyServer = new JettyServer();
        jettyServer.start();
    }

    @AfterAll
    void stopServer() throws Exception {
        jettyServer.stop();
    }

    @Test
    void checkServerStatus() throws IOException {
        URL url = new URL("http://localhost:8080/status");
        HttpURLConnection connection =
                (HttpURLConnection) url.openConnection();
        int response = connection.getResponseCode();

        assertEquals(200, response);
    }

    @Test
    void checkInvalidEndpoint() throws IOException {
        URL url = new URL("http://localhost:8080/invalid");
        HttpURLConnection connection =
                (HttpURLConnection) url.openConnection();
        int response = connection.getResponseCode();

        assertEquals(404, response);
    }
}
```

Nested Test Lifecycle

Lifecycle methods can be applied to nested tests as well. However, by default, @BeforeAll and @AfterAll methods do not work. This is because nested tests need to be inner classes, and Java does not support static methods for inner classes.

The way to make it work is to annotate the nested class with @TestInstance(Lifecycle.PER_CLASS):

```
public class NestedLifecycleTest {
    @Nested
    class HappyPath {
        @BeforeEach
        void beforeEachHappyPath() {
            System.out.println("Before each happy path");
        }

        @AfterEach
        void afterEachHappyPath() {
            System.out.println("After each happy path");
        }

        @Test
        void happyPathOne() {
            System.out.println("Happy path one");
        }

        @Test
        void happyPathTwo() {
            System.out.println("Happy path two");
        }
    }

    @Nested
    @TestInstance(TestInstance.Lifecycle.PER_CLASS)
    class ExceptionalPath {
        @BeforeAll
        void beforeEntireExceptionalPath() {
            System.out.println("Before entire exceptional path");
        }

        @AfterAll
        void afterEntireExceptionalPath() {
            System.out.println("After entire exceptional path");
        }

        @Test
        void exceptionalPathOne() {
            System.out.println("Exceptional path one");
        }

        @Test
        void exceptionalPathTwo() {
            System.out.println("Exceptional path two");
        }
    }
}
```

As we can see, the lifecycle methods apply to the nested tests individually:

```
Before entire exceptional path
  Exceptional path one
  Exceptional path two
After entire exceptional path
Before each happy path
  Happy path one
After each happy path
Before each happy path
  Happy path two
After each happy path
```

Extension Lifecycle

When using extensions, JUnit calls extension lifecycle callbacks in addition to the lifecycle methods of the test class.

JUnit guarantees wrapping behavior for multiple registered extensions. Given extensions ExtensionOne and ExtensionTwo, it’s guaranteed the “before” callbacks of ExtensionOne execute before ExtensionTwo. Similarly, it’s guaranteed any “after” callbacks of ExtensionOne execute after ExtensionTwo.

Again, let’s take a look at an example:

```
public class ExtensionOne implements BeforeEachCallback, AfterEachCallback {
    @Override
    public void beforeEach(ExtensionContext context) {
        System.out.println("Before each from ExtensionOne");
    }

    @Override
    public void afterEach(ExtensionContext context) {
        System.out.println("After each from ExtensionOne");
    }
}

@ExtendWith(ExtensionOne.class)
@ExtendWith(ExtensionTwo.class)
public class ExtensionLifecycleTest {
    @BeforeEach
    void beforeEachTest() {
        System.out.println("Before each test");
    }

    @AfterEach
    void afterEachTest() {
        System.out.println("After each test");
    }

    @Test
    void firstTest() {
        System.out.println("First test");
    }

    @Test
    void secondTest() {
        System.out.println("Second test");
    }
}
```

Executing the example, we can see the wrapping behavior of the extensions:

```
Before each from ExtensionOne
  Before each from ExtensionTwo
    Before each test
      First test
    After each test
  After each from ExtensionTwo
After each from ExtensionOne
Before each from ExtensionOne
  Before each from ExtensionTwo
    Before each test
      Second test
    After each test
  After each from ExtensionTwo
After each from ExtensionOne
```

ASSERTIONS:

JUnit 5 Assertions Examples

JUnit 5 assertions help in validating the expected output with actual output of a testcase. To keep things simple, all **JUnit Jupiter assertions** are static methods in the org.junit.jupiter.Assertions class.

Table of Contents

Assertions.assertEquals() and Assertions.assertNotEquals() Example

Use Assertions.assertEquals() to assert that **expected value and actual value are equal**. assertEquals() has many overloaded methods for different data types e.g. int, short, float, char etc. It also support passing error message to be printed in case test fails. e.g.

```
public static void assertEquals(int expected, int actual)
public static void assertEquals(int expected, int actual, String message)
public static void assertEquals(int expected, int actual, Supplier<String< messageSupplier)
```

```
void testCase()
{
    //Test will pass
    Assertions.assertEquals(4, Calculator.add(2, 2));


    //Test will fail
    Assertions.assertEquals(3, Calculator.add(2, 2), "Calculator.add(2, 2) test failed");
    //Test will fail

    Supplier&lt;String&gt; messageSupplier  = ()-> "Calculator.add(2, 2) test failed";

    Assertions.assertEquals(3, Calculator.add(2, 2), messageSupplier);

}
```

Similarly, Assertions.assertNotEquals() method is used to assert that **expected value and actual value are NOT equal**. In contrast to assertEquals(), assertNotEquals() does not overloaded methods for different data types but only Object is accepted.

```
public static void assertNotEquals(Object expected, Object actual)
public static void assertNotEquals(Object expected, Object actual, String message)
public static void assertNotEquals(Object expected, Object actual, Supplier<String> messageSupplier)
```

```
void testCase()

{

    //Test will pass

    Assertions.assertNotEquals(3, Calculator.add(2, 2));


    //Test will fail

    Assertions.assertNotEquals(4, Calculator.add(2, 2), "Calculator.add(2, 2) test failed");


    //Test will fail

    Supplier&lt;String&gt; messageSupplier  = ()-> "Calculator.add(2, 2) test failed";

    Assertions.assertNotEquals(4, Calculator.add(2, 2), messageSupplier);

}
```

Assertions.assertArrayEquals() Example

Similar to assertEquals(), assertArrayEquals() does the same for arrays i.e. asserts that **expected and actual arrays are equal**. It also has overloaded methods for different data types e.g. boolean[], char[], int[] etc. It also support passing error message to be printed in case test fails. e.g.

```
public static void assertArrayEquals(int[] expected, int[] actual)
public static void assertArrayEquals(int[] expected, int[] actual, String message)
public static void assertArrayEquals(int[] expected, int[] actual, Supplier<String> messageSupplier)
```

```
void testCase()
{
    //Test will pass
    Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3}, "Array Equal Test");

    //Test will fail because element order is different
    Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,3,2}, "Array Equal Test");

    //Test will fail because number of elements are different
    Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3,4}, "Array Equal Test");
}
```

Assertions.assertIterableEquals() Example

It asserts that **expected and actual iterables are deeply equal**. Deeply equal means that number and order of elements in collection must be same; as well as iterated elements must be equal.

It also has 3 overloaded methods.

```
public static void assertIterableEquals(Iterable<?> expected, Iterable> actual)
public static void assertIterableEquals(Iterable<?> expected, Iterable> actual, String message)
public static void assertIterableEquals(Iterable<?> expected, Iterable> actual, Supplier<String>
messageSupplier)
```

```
@Test
void testCase()
{
     Iterable<Integer> listOne = new ArrayList<>(Arrays.asList(1,2,3,4));
     Iterable<Integer> listTwo = new ArrayList<>(Arrays.asList(1,2,3,4));
     Iterable<Integer> listThree = new ArrayList<>(Arrays.asList(1,2,3));
     Iterable<Integer> listFour = new ArrayList<>(Arrays.asList(1,2,4,3));

    //Test will pass
    Assertions.assertIterableEquals(listOne, listTwo);
    //Test will fail

    Assertions.assertIterableEquals(listOne, listThree);


    //Test will fail

    Assertions.assertIterableEquals(listOne, listFour);

}
```

Assertions.assertLinesMatch() Example

It asserts that **expected list of Strings matches actual list**. The logic to match a string with another string is :

1. check if `expected.equals(actual)` – if yes, continue with next pair
2. otherwise treat `expected` as a regular expression and check via `String.matches(String)` – if yes, continue with next pair
3. otherwise check if `expected` line is a fast-forward marker, if yes apply fast-forward actual lines accordingly and goto 1.

A valid fast-forward marker is string which start and end with `>>` and and contains at least 4 characters. Any character between the fast-forward literals are discarded.

```
>>>>
>> stacktrace >>
>> single line, non Integer.parse()-able comment >>
```

Assertions.assertNotNull() and Assertions.assertNull() Example

`assertNotNull()` asserts that **actual is NOT null**. Similarly, `assertNull()` method asserts that **actual is null**. Both has three overloaded methods.

```
public static void assertNotNull(Object actual)
public static void assertNotNull(Object actual, String message)
public static void assertNotNull(Object actual, Supplier<String> messageSupplier)

public static void assertEquals(Object actual)
public static void assertEquals(Object actual, String message)
public static void assertEquals(Object actual, Supplier<String> messageSupplier)
```

```
@Test

void testCase()

{
    String nullString = null;

    String notNullString = "howtodoinjava.com";


    //Test will pass

    Assertions.assertNotNull(notNullString);


    //Test will fail

    Assertions.assertNotNull(nullString);


    //Test will pass

    Assertions.assertNull(nullString);


    // Test will fail

    Assertions.assertNull(notNullString);

}
```

Assertions.assertNotSame() and Assertions.assertSame() Example

assertNotSame() asserts that **expected and actual DO NOT refer to the same object.**.
Similarly, assertSame() method asserts that **expected and actual refer to exactly same object.**.
Both has three overloaded methods.

```
public static void assertNotSame(Object actual)
public static void assertNotSame(Object actual, String message)
public static void assertNotSame(Object actual, Supplier<> messageSupplier)

public static void assertSame(Object actual)
public static void assertSame(Object actual, String message)
public static void assertSame(Object actual, Supplier<String> messageSupplier)
```

```
@Test

void testCase()

{

    String originalObject = "howtodoinjava.com";

    String cloneObject = originalObject;
    String otherObject = "example.com";

    //Test will pass
    Assertions.assertNotSame(originalObject, otherObject);

    //Test will fail
    Assertions.assertNotSame(originalObject, cloneObject);

    //Test will pass
    Assertions.assertSame(originalObject, cloneObject);

    // Test will fail
    Assertions.assertSame(originalObject, otherObject);
}
```

Assertions.assertTimeout() and Assertions.assertTimeoutPreemptively() Example

assertTimeout() and assertTimeoutPreemptively() both are used to test long running tasks. If given task inside testcase takes more than specified duration, then test will fail.
Only different between both methods is that in assertTimeoutPreemptively(), execution of the Executable or ThrowingSupplier will be preemptively aborted if the timeout is exceeded. In case of assertTimeout(), Executable or ThrowingSupplier will NOT be aborted.

```
public static void assertTimeout(Duration timeout, Executable executable)
public static void assertTimeout(Duration timeout, Executable executable, String message)
public static void assertTimeout(Duration timeout, Executable executable, Supplier<String>
messageSupplier)
public static void assertTimeout(Duration timeout, ThrowingSupplier<T> supplier, String message)
public static void assertTimeout(Duration timeout, ThrowingSupplier<T> supplier, Supplier<String>
messageSupplier)
```

```
@Test
void testCase() {

    //This will pass
    Assertions.assertTimeout(Duration.ofMinutes(1), () -> {
        return "result";
    });

    //This will fail
    Assertions.assertTimeout(Duration.ofMillis(100), () -> {
        Thread.sleep(200);
        return "result";
    });

    //This will fail
    Assertions.assertTimeoutPreemptively(Duration.ofMillis(100), () -> {
        Thread.sleep(200);
        return "result";
    });
}
```

Assertions.assertTrue() and Assertions.assertFalse() Example

assertTrue() asserts that the supplied condition is true or boolean condition supplied by BooleanSupplier is true. Similarly, assertFalse() asserts that **supplied condition is false**. It has following overloaded methods:

```
public static void assertTrue(boolean condition)
public static void assertTrue(boolean condition, String message)
public static void assertTrue(boolean condition, Supplier<String> messageSupplier)
public static void assertTrue(BooleanSupplier booleanSupplier)
public static void assertTrue(BooleanSupplier booleanSupplier, String message)
public static void assertTrue(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier)

public static void assertFalse(boolean condition)
public static void assertFalse(boolean condition, String message)
public static void assertFalse(boolean condition, Supplier<String> messageSupplier)
public static void assertFalse(BooleanSupplier booleanSupplier)
public static void assertFalse(BooleanSupplier booleanSupplier, String message)
public static void assertFalse(BooleanSupplier booleanSupplier, Supplier<String> messageSupplier)
```

```
@Test
void testCase() {
    boolean trueBool = true;

    boolean falseBool = false;


    Assertions.assertTrue(trueBool);

    Assertions.assertTrue(falseBool, "test execution message");

    Assertions.assertTrue(falseBool, AppTest::message);

    Assertions.assertTrue(AppTest::getResult, AppTest::message);


    Assertions.assertFalse(falseBool);

    Assertions.assertFalse(trueBool, "test execution message");

    Assertions.assertFalse(trueBool, AppTest::message);

    Assertions.assertFalse(AppTest::getResult, AppTest::message);

}


private static String message () {

    return "Test execution result";

}


private static boolean getResult () {

    return true;

}
```

Assertions.assertThrows() Example

It asserts that execution of the supplied Executable throws an exception of the expectedType and returns the exception.

```
public static <T extends Throwable> T assertThrows(Class<T> expectedType,
            Executable executable)
```

```
@Test

void testCase() {
    Throwable exception = Assertions.assertThrows(IllegalArgumentException.class, () -
> {

        throw new IllegalArgumentException("error message");

    });

}
```

Assertions.fail() Example

fail() method simply fails the test. It has following overloaded methods:

```
public static void fail(String message)
public static void fail(Throwable cause)
public static void fail(String message, Throwable cause)
public static void fail(Supplier<String> messageSupplier)
```

```
public class AppTest {

    @Test

    void testCase() {


        Assertions.fail("not found good reason to pass");

        Assertions.fail(AppTest::message);

    }


    private static String message () {

        return "not found good reason to pass";

    }

}
```

DISABLING TESTS

JUnit 5 @Disabled Test Example

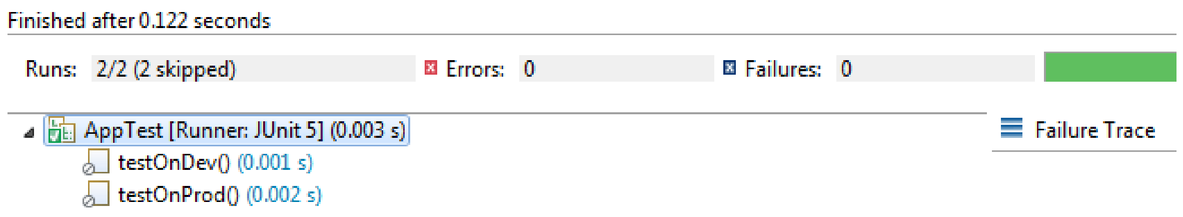
JUnit @Disabled annotation can be used to disable the test methods from test suite. This annotation can be applied over a test class as well as over individual test methods.

It accepts only **one optional parameter**, which indicates the reason this test is disabled.

@Disabled Test Class

When @Disabled is applied over test class, **all test methods within that class are automatically disabled** as well.

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

import org.junit.jupiter.api.Disabled;

import org.junit.jupiter.api.Test;


@Disabled

public class AppTest {


    @Test

    void testOnDev()

    {

        System.setProperty("ENV", "DEV");

        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));

    }


    @Test

    void testOnProd()

    {

        System.setProperty("ENV", "PROD");

        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));

    }

}
```


Finished after 0.122 seconds

Runs: 2/2 (2 skipped) Errors: 0 Failures: 0

```
AppTest [Runner: JUnit 5] (0.003 s)
    testOnDev() (0.001 s)
    testOnProd() (0.002 s)
```

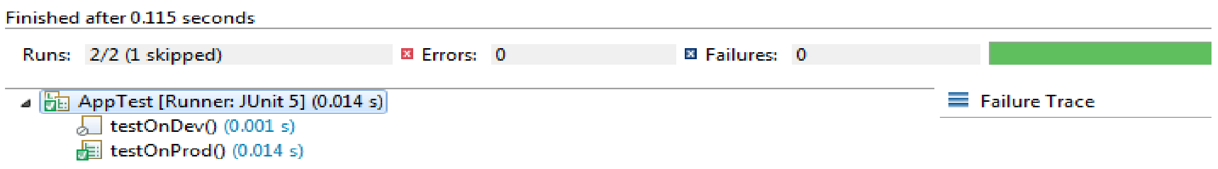
JUnit 5 Disabled Annotation Over Class

Notice the count of Runs: 2/2 (2 skipped). Clearly both tests are disabled so not executed.

JUnit 5 @Disabled Test Method

@Disabled is used to signal that the annotated **test method is currently disabled and should not be executed**.

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

import org.junit.jupiter.api.Disabled;

import org.junit.jupiter.api.Test;


public class AppTest {


    @Disabled("Do not run in lower environment")

    @Test

    void testOnDev()

    {

        System.setProperty("ENV", "DEV");

        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));

    }


    @Test

    void testOnProd()

    {

        System.setProperty("ENV", "PROD");

        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));
    }
}
```



```
Finished after 0.115 seconds
Runs: 2/2 (1 skipped)    Errors: 0    Failures: 0

AppTest [Runner: JUnit 5] (0.014 s)
    testOnDev() (0.001 s)
    testOnProd() (0.014 s)
```

JUnit 5

Disabled Annotation Over Method

ASSUMPTIONS

JUnit 5 Assumptions Examples

JUnit 5 Assumptions class provides static methods to support conditional test execution based on assumptions. A failed assumption results in a test being aborted. Assumptions are typically used whenever it does not make sense to continue execution of a given test method. In test report, these test will be marked as passed.

JUnit jupiter Assumptions class has two such methods: assumeFalse(), assumeTrue().
A third method assumeThat() is in Experimental state and might be confirmed in future.

Table of Contents

JUnit 5 Assumptions.assumeTrue()

assumeTrue() validates the given assumption to true and if assumption is true – test proceed, otherwise test execution is aborted.

It has following overloaded methods.

```
public static void assumeTrue(boolean assumption) throws TestAbortedException
public static void assumeTrue(boolean assumption, Supplier<String> messageSupplier) throws
TestAbortedException
```

```
public static void assumeTrue(boolean assumption, String message) throws TestAbortedException

public static void assumeTrue(BooleanSupplier assumptionSupplier) throws TestAbortedException
public static void assumeTrue(BooleanSupplier assumptionSupplier, String message) throws
TestAbortedException
public static void assumeTrue(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier)
throws TestAbortedException
```

```
public class AppTest {

    @Test

    void testOnDev()

    {

        System.setProperty("ENV", "DEV");

        Assumptions.assumeTrue("DEV".equals(System.getProperty("ENV")));

        //remainder of test will proceed

    }


    @Test

    void testOnProd()

    {

        System.setProperty("ENV", "PROD");

        Assumptions.assumeTrue("DEV".equals(System.getProperty("ENV")),
AppTest::message);

        //remainder of test will be aborted

    }


    private static String message () {

        return "TEST Execution Failed :: ";

    }

}
```

JUnit 5 Assumptions.assumeFalse()

assumeFalse() validates the given assumption to false and if assumption is false – test proceed, otherwise test execution is aborted. It works just opposite to assumeTrue().

It has following overloaded methods.

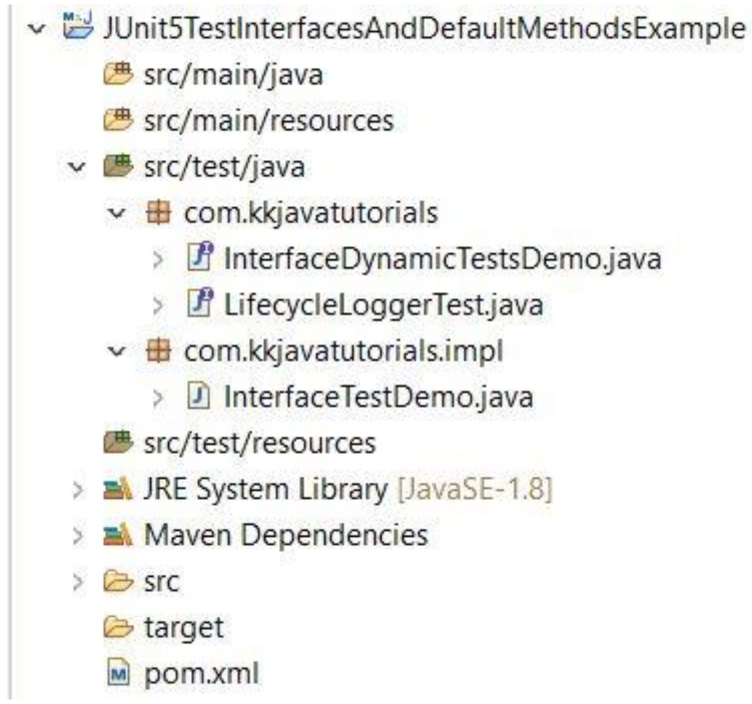
```
public static void assumeFalse(boolean assumption) throws TestAbortedException
public static void assumeFalse(boolean assumption, Supplier<String> messageSupplier) throws
TestAbortedException
public static void assumeFalse(boolean assumption, String message) throws TestAbortedException

public static void assumeFalse(BooleanSupplier assumptionSupplier) throws TestAbortedException
public static void assumeFalse(BooleanSupplier assumptionSupplier, String message) throws
TestAbortedException
public static void assumeFalse(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier)
throws TestAbortedException
```

```
public class AppTest {

    @Test

    void testOnDev()

    {

        System.setProperty("ENV", "DEV");

        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")),
AppTest::message);

      //remainder of test will be aborted

    }


    @Test

    void testOnProd()

    {

        System.setProperty("ENV", "PROD");

        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));

      //remainder of test will proceed


    }


    private static String message () {
        return "TEST Execution Failed :: ";
    }
}
```

Test Interfaces and Default Methods in JUnit 5

JUnit Jupiter allows @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @TestTemplate, @BeforeEach, and @AfterEach to be declared on interface default methods.
@BeforeAll and @AfterAll can either be declared on static methods in a test interface or on interface default methods if the test interface or test class is annotated with @TestInstance(Lifecycle.PER_CLASS)

@ExtendWith and @Tag can be declared on a test interface so that classes that implement the interface automatically inherit its tags and extensions.
Finally, In your test class, you can implement these test interfaces to have them applied.
Let's try to understand the above concept using a demo project:



```
JUnit5TestInterfacesAndDefaultMethodsExample
    src/main/java
    src/main/resources
    src/test/java
        com.kkjavatutorials
            InterfaceDynamicTestsDemo.java
            LifecycleLoggerTest.java
        com.kkjavatutorials.impl
            InterfaceTestDemo.java
    src/test/resources
    JRE System Library [JavaSE-1.8]
    Maven Dependencies
    src
    target
    pom.xml
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.kkjavatutorials</groupId>
        <artifactId>JUnit5TestInterfacesAndDefaultMethodsExample</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <properties>
                <maven.compiler.target>8</maven.compiler.target>
                <maven.compiler.source>8</maven.compiler.source>
                <junit.jupiter.version>5.6.2</junit.jupiter.version>
        </properties>

        <dependencies>

                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-api</artifactId>
                    <version>${junit.jupiter.version}</version>
                    <scope>test</scope>
                </dependency>

                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>${junit.jupiter.version}</version>
                    <scope>test</scope>
                </dependency>

        </dependencies>
</project>
```

LifecycleLoggerTest.java

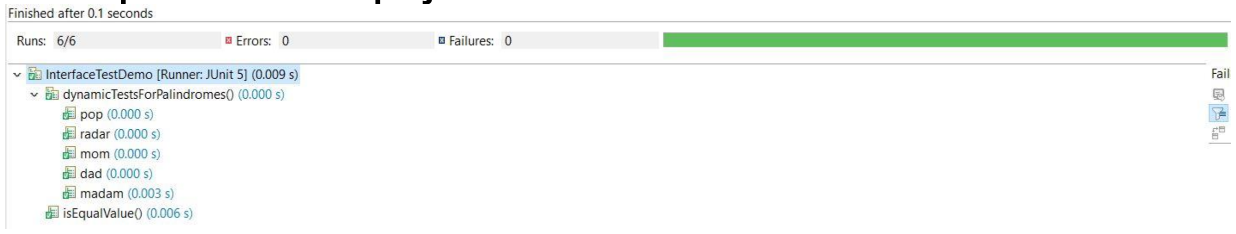
```
package com.kkjavatutorials;

```

```
import java.util.logging.Logger;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.TestInfo;
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.api.TestInstance.Lifecycle;

@TestInstance(value = Lifecycle.PER_CLASS)
public interface LifecycleLoggerTest {

    static final Logger LOG = Logger.getLogger(LifecycleLoggerTest.class.getName());

    @BeforeAll
    default void beforeAll() {
        LOG.info("This Method runs before all tests");
    }

    @AfterAll
    default void afterAll() {
        LOG.info("This Method runs after all tests");
    }

    @BeforeEach
    default void beforeEach(TestInfo testInfo) {
        LOG.info(() -> String.format("About to execute [%s]",
            testInfo.getDisplayName()));
    }

    @AfterEach
    default void afterEach(TestInfo testInfo) {
        LOG.info(() -> String.format("Finished executing [%s]",
            testInfo.getDisplayName()));
    }
}
```

InterfaceDynamicTestsDemo.java

```
package com.kkjavatutorials;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

import java.util.stream.Stream;

import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

public interface InterfaceDynamicTestsDemo {

    @TestFactory
    default Stream<DynamicTest> dynamicTestsForPalindromes() {
        return Stream.of("pop", "radar", "mom", "dad","madam")
            .map(inputText -> dynamicTest(inputText, () -> assertTrue(isPalindrome(inputText))));
    }

    /**
     * Method to check whether input String is Palindrome or not
     * @param inputText
     * @return return true if input is Palindrome else false
     */
    default boolean isPalindrome(String inputText) {
                return new StringBuffer(inputText).reverse().toString().equals(inputText);
    }
}
```

The output of the above project:



```
Finished after 0.1 seconds
Runs: 6/6        Errors: 0        Failures: 0

InterfaceTestDemo [Runner: JUnit 5] (0.009 s)
  dynamicTestsForPalindromes() (0.000 s)
    pop (0.000 s)
    radar (0.000 s)
    mom (0.000 s)
    dad (0.000 s)
    madam (0.003 s)
  isEqualValue() (0.006 s)
```

REPEATING TESTS

Maven Dependencies and Setup

The first thing to note is that JUnit 5 needs Java 8 to run. Having said that, let's have a look at the Maven dependency:

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
```

```
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

This is the main JUnit 5 dependency that we need to add to write our tests. Check out the latest version of the artifact here.

A Simple *@RepeatedTest* Example

Creating a repeated test is simple – just add the *@RepeatedTest* annotation on top of the test method:

```
@RepeatedTest(3)
void repeatedTest(TestInfo testInfo) {
    System.out.println("Executing repeated test");

    assertEquals(2, Math.addExact(1, 1), "1 + 1 should equal 2");
}
```

Note that instead of standard *@Test* annotation, we are using *@RepeatedTest* for our unit test. **The above test will be executed three times** as if the same test was written three times.

The test reports (the report files or the results in the JUnit tab of your IDE) will display all the executions:

```
repetition 1 of 3(repeatedTest(TestInfo))
repetition 2 of 3(repeatedTest(TestInfo))
repetition 3 of 3(repeatedTest(TestInfo))
```

Lifecycle Support for *@RepeatedTest*

Each execution of the *@RepeatedTest* will behave like a regular *@Test* having full JUnit test life cycle support. Meaning that, during each execution, the *@BeforeEach* and *@AfterEach* methods will be called. To demonstrate this, just add the appropriate methods in the test class:

```
@BeforeEach
void beforeEachTest() {
    System.out.println("Before Each Test");
}

@AfterEach
void afterEachTest() {
    System.out.println("After Each Test");
    System.out.println("=====================");
}
```

If we run our previous test, the results will be displayed on the console:

```
Before Each Test
Executing repeated test
After Each Test
=====================
Before Each Test
Executing repeated test
After Each Test
=====================
Before Each Test
```

```
Executing repeated test
After Each Test
=====================
```

As we can see, the **@*BeforeEach* and @*AfterEach* methods are called around each execution**.

Configuring the Test Name

In the first example, we have observed that the output of the test report does not contain any identifiers. This can be configured further using the *name* attribute:

```
@RepeatedTest(value = 3, name = RepeatedTest.LONG_DISPLAY_NAME)
void repeatedTestWithLongName() {
   System.out.println("Executing repeated test with long name");

   assertEquals(2, Math.addExact(1, 1), "1 + 1 should equal 2");
}
```

The output will now contain the method name along with the repetition index:

```
repeatedTestWithLongName() :: repetition 1 of 3(repeatedTestWithLongName())
repeatedTestWithLongName() :: repetition 2 of 3(repeatedTestWithLongName())
repeatedTestWithLongName() :: repetition 3 of 3(repeatedTestWithLongName())
```

Another option is to use *RepeatedTest.SHORT_DISPLAY_NAME* which will produce the short name of the test:

```
repetition 1 of 3(repeatedTestWithShortName())
repetition 2 of 3(repeatedTestWithShortName())
repetition 3 of 3(repeatedTestWithShortName())
```

If however, we need to use our customized name, it is very much possible:

```
@RepeatedTest(value = 3, name = "Custom name {currentRepetition}/{totalRepetitions}")
void repeatedTestWithCustomDisplayName(TestInfo testInfo) {
   assertEquals(2, Math.addExact(1, 1), "1 + 1 should equal 2");
}
```

The *{currentRepetition}* and *{totalRepetitions}* are the placeholders for the current repetition and the total number of repetitions. These values are automatically provided by JUnit at the runtime, and no additional configuration is required. The output is pretty much what we expected:

```
Custom name 1/3(repeatedTestWithCustomDisplayName())
Custom name 2/3(repeatedTestWithCustomDisplayName())
Custom name 3/3(repeatedTestWithCustomDisplayName())
```

Accessing the *RepetitionInfo*

Apart from the *name* attribute, JUnit provides access to the repetition metadata in our test code as well. This is achieved by adding a *RepetitionInfo* parameter to our test method:

```
@RepeatedTest(3)
void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
   System.out.println("Repetition #" + repetitionInfo.getCurrentRepetition());

   assertEquals(3, repetitionInfo.getTotalRepetitions());
}
```

The output will contain the current repetition index for each of the execution:

```
Repetition #1
Repetition #2
Repetition #3
```

The *RepetitionInfo* is provided by *RepetitionInfoParameterResolver* and is available only in the context of *@RepeatedTest.*

DYNAMIC TESTS :

The standard tests annotated with *@Test* annotation are static tests which are fully specified at the compile time. **A *DynamicTest* is a test generated during runtime**. These tests are generated by a factory method annotated with the *@TestFactory* annotation.

A *@TestFactory* method must return a *Stream*, *Collection*, *Iterable*, or *Iterator* of *DynamicTest* instances. Returning anything else will result in a *JUnitException* since the invalid return types cannot be detected at compile time. Apart from this, a *@TestFactory* method cannot be static or *private*.

The *DynamicTest*s are executed differently than the standard *@Test*s and do not support lifecycle callbacks. Meaning, the ***@BeforeEach* and the *@AfterEach* methods will not be called for the *DynamicTests***.

Creating *DynamicTests*

First, let's have a look at different ways of creating *DynamicTest*s.

The examples here are not dynamic in nature, but they'll provide a good starting point for creating truly dynamic ones.

We're going to create a *Collection* of *DynamicTest*:

```
@TestFactory
Collection<DynamicTest> dynamicTestsWithCollection() {
    return Arrays.asList(
      DynamicTest.dynamicTest("Add test",
        () -> assertEquals(2, Math.addExact(1, 1))),
      DynamicTest.dynamicTest("Multiply Test",
        () -> assertEquals(4, Math.multiplyExact(2, 2))));
}
```

The *@TestFactory* method tells JUnit that this is a factory for creating dynamic tests. As we can see, we're only returning a *Collection* of *DynamicTest*. **Each of the *DynamicTest* consists of two parts, the name of the test or the display name, and an *Executable***.

The output will contain the display name that we passed to the dynamic tests:

```
Add test(dynamicTestsWithCollection())
Multiply Test(dynamicTestsWithCollection())
```

The same test can be modified to return an *Iterable*, *Iterator*, or a *Stream*:

```
@TestFactory
Iterable<DynamicTest> dynamicTestsWithIterable() {
    return Arrays.asList(
      DynamicTest.dynamicTest("Add test",
        () -> assertEquals(2, Math.addExact(1, 1))),
      DynamicTest.dynamicTest("Multiply Test",
        () -> assertEquals(4, Math.multiplyExact(2, 2))));
}

@TestFactory
Iterator<DynamicTest> dynamicTestsWithIterator() {
    return Arrays.asList(
      DynamicTest.dynamicTest("Add test",
        () -> assertEquals(2, Math.addExact(1, 1))),
      DynamicTest.dynamicTest("Multiply Test",
        () -> assertEquals(4, Math.multiplyExact(2, 2))))
        .iterator();
}

@TestFactory
Stream<DynamicTest> dynamicTestsFromIntStream() {
    return IntStream.iterate(0, n -> n + 2).limit(10)
      .mapToObj(n -> DynamicTest.dynamicTest("test" + n,
        () -> assertTrue(n % 2 == 0)));
}
```

Please note that if the *@TestFactory* returns a *Stream*, then it will be automatically closed once all the tests are executed.

The output will be pretty much the same as the first example. It will contain the display name that we pass to the dynamic test.

Creating a *Stream* of *DynamicTests*

For the demonstration purposes, consider a *DomainNameResolver* which returns an IP address when we pass the domain name as input.

For the sake of simplicity, let's have a look at the high-level skeleton of our factory method:

```
@TestFactory
Stream<DynamicTest> dynamicTestsFromStream() {

    // sample input and output
    List<String> inputList = Arrays.asList(
      "www.somedomain.com", "www.anotherdomain.com", "www.yetanotherdomain.com");
    List<String> outputList = Arrays.asList(
      "154.174.10.56", "211.152.104.132", "178.144.120.156");

    // input generator that generates inputs using inputList
    /*...code here...*/

    // a display name generator that creates a
    // different name based on the input
    /*...code here...*/
    // the test executor, which actually has the
    // logic to execute the test case
    /*...code here...*/

    // combine everything and return a Stream of DynamicTest
    /*...code here...*/
}
```

There isn't much code related to *DynamicTest* here apart from the *@TestFactory* annotation, which we're already familiar with.

The two *ArrayList*s will be used as input to *DomainNameResolver* and expected output respectively.

Let's now have a look at the input generator:

```
Iterator<String> inputGenerator = inputList.iterator();
```

The input generator is nothing but an *Iterator* of *String*. It uses our *inputList* and returns the domain name one by one.

The display name generator is fairly simple:

```
Function<String, String> displayNameGenerator
  = (input) -> "Resolving: " + input;
```

The task of a display name generator is just to provide a display name for the test case that will be used in JUnit reports or the JUnit tab of our IDE.

Here we are just utilizing the domain name to generate unique names for each test. It's not required to create unique names, but it will help in case of any failure. Having this, we'll be able to tell the domain name for which the test case failed.

Now let's have a look at the central part of our test – the test execution code:

```
DomainNameResolver resolver = new DomainNameResolver();
ThrowingConsumer<String> testExecutor = (input) -> {
    int id = inputList.indexOf(input);

    assertEquals(outputList.get(id), resolver.resolveDomain(input));
};
```

We have used the *ThrowingConsumer*, which is a *@FunctionalInterface* for writing the test case. For each input generated by the data generator, we're fetching the expected output from the *outputList* and the actual output from an instance of *DomainNameResolver*.

Now the last part is simply to assemble all the pieces and return as a *Stream* of *DynamicTest*:

```
return DynamicTest.stream(
  inputGenerator, displayNameGenerator, testExecutor);
```

That's it. Running the test will display the report containing the names defined by our display name generator:

```
Resolving: www.somedomain.com(dynamicTestsFromStream())
Resolving: www.anotherdomain.com(dynamicTestsFromStream())
Resolving: www.yetanotherdomain.com(dynamicTestsFromStream())
```

Improving the *DynamicTest* Using Java 8 Features

The test factory written in the previous section can be drastically improved by using the features of Java 8. The resultant code will be much cleaner and can be written in a lesser number of lines:

```
@TestFactory
Stream<DynamicTest> dynamicTestsFromStreamInJava8() {

   DomainNameResolver resolver = new DomainNameResolver();

   List<String> domainNames = Arrays.asList(
     "www.somedomain.com", "www.anotherdomain.com", "www.yetanotherdomain.com");
   List<String> outputList = Arrays.asList(
     "154.174.10.56", "211.152.104.132", "178.144.120.156");

   return inputList.stream()
     .map(dom -> DynamicTest.dynamicTest("Resolving: " + dom,
       () -> {int id = inputList.indexOf(dom);

     assertEquals(outputList.get(id), resolver.resolveDomain(dom));
   }));
}
```

The above code has the same effect as the one we saw in the previous section.
The *inputList.stream().map()* provides the stream of inputs (input generator). The first argument to *dynamicTest()* is our display name generator ("Resolving: " + *dom*) while the second argument, a *lambda*, is our test executor.

The output will be the same as the one from the previous section.

parameterized tests with JUnit 5.

After we have finished this, we:

- Can get the required dependencies with Maven and Gradle.
- Know how we can customize the display name of each method invocation.
- Understand how we can use different argument sources.
- Can write custom argument converters.

Getting the Required Dependencies

Before we can write parameterized tests with JUnit 5, we have to declare the junit-jupiter-params dependency in our build script.
If we are using Maven, we have to add the following snippet to our POM file:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.1.0</version>
    <scope>test</scope>
</dependency>
```

If we are using Gradle, we have to add the junit-jupiter-params dependency to the testCompile dependency configuration. We can do this by adding the following snippet to our *build.gradle* file:

```

testCompile(
        'org.junit.jupiter:junit-jupiter-params:5.1.0'
)
```

Let’s move on and write our first parameterized test with JUnit 5.

Writing Our First Parameterized Tests

If our test method takes only one method parameter that is either a String or a primitive type supported by the @ValueSource annotation (int, long, or double), we can write a parameterized test with JUnit 5 by following these steps:

1. Add a new test method to our test class and ensure that this method takes a String object as a method parameter.
2. Configure the display name of the test method.
3. Annotate the test method with the @ParameterizedTest annotation. This annotation identifies parameterized test methods.
4. Provide the method parameters that are passed to our test method. Because our test method takes one String object as a method parameter, we can provide its method parameters by annotating our test method with the @ValueSource annotation.

After we have added a new parameterized test to our test class, its source code looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;
```

```
@DisplayName("Pass the method parameters provided by the
@ValueSource annotation")
class ValueSourceExampleTest {

    @DisplayName("Should pass a non-null message to our test method")
    @ParameterizedTest
    @ValueSource(strings = {"Hello", "World"})
    void shouldPassNonNullMessageAsMethodParameter(String message) {
        assertNotNull(message);
    }
}
```

When we run our parameterized test, we should see an output that looks as follows:

```
Pass the method parameters provided by the @ValueSource annotation
|_ Should pass a non-null message to our test method
   |_ [1] Hello
   |_ [2] World
```

Even though this output looks quite clean, sometimes we want to provide our own display name for each method invocation. Let’s find out how we can do it.

Customizing the Display Name of Each Method Invocation

We can customize the display name of each method invocation by setting the value of the @ParameterizedTest annotation’s name attribute. This attribute supports the following placeholders:

- {index}: The index of the current invocation. Note that the index of the first invocation is one.
- {arguments}: A comma separated list that contains all method parameters.
- {i}: The actual method parameter (i specifies the index of the method parameter). Note that the index of the first method parameter is zero.

Let’s provide a custom display name to our test method. This display name must display the index of the current invocation and the provided method parameter. After we have configured the custom display name of each method invocation, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;

@DisplayName("Pass the method parameters provided by the @ValueSource
annotation")
class ValueSourceExampleTest {

    @DisplayName("Should pass a non-null message to our test method")
    @ParameterizedTest(name = "{index} => message=''{0}''")
    @ValueSource(strings = {"Hello", "World"})
    void shouldPassNonNullMessageAsMethodParameter(String message) {
        assertNotNull(message);
    }
}
```

When we run our parameterized test, we should see an output that looks as follows:

```
Pass the method parameters provided by the @ValueSource annotation
|_ Should pass a non-null message to our test method
   |_ 1 => message='Hello'
   |_ 2 => message='World'
```

As we remember, the @ValueSource annotation is a good choice if our test method takes only one method parameter that is supported by the @ValueSource annotation. However, most of the time this is not the case. Next, we will find out how we can solve this problem by using different argument sources.

Using Argument Sources

The @ValueSource annotation is the simplest argument source that is supported by JUnit 5. However, JUnit 5 support other argument sources as well. All supported argument sources are configured by using annotations found from the org.junit.jupiter.params.provider package.
This section describes how we can use the more complex argument sources provided by JUnit 5. Let's start by finding out how we can pass enum values to our parameterized test.

Passing Enum Values to Our Parameterized Test

If our parameterized test takes one enum value as a method parameter, we have to annotate our test method with the @EnumSource annotation and specify the enum values which are passed to our test method.
Let’s assume that we have to write a parameterized test that takes a value of the Pet enum as a method parameter. The source code of the Pet enum looks as follows:

```
enum Pet {
    CAT,
    DOG;
}
```

If we want to pass all enum values to our test method, we have to annotate our test method with the @EnumSource annotation and specify the enum whose values are passed to our test method. After we have done this, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;

@DisplayName("Pass enum values to our test method")
class EnumSourceExampleTest {

    @DisplayName("Should pass non-null enum values as method parameters")
    @ParameterizedTest(name = "{index} => pet=''{0}''")
    @EnumSource(Pet.class)
    void shouldPassNonNullEnumValuesAsMethodParameter(Pet pet) {
        assertNotNull(pet);
    }
}
```

When we run this test method, we see that JUnit 5 passes all values of the Pet enum to our test method as method parameters:

```
Pass enum values to our test method

|_ Should pass non-null enum values as method parameters
```

```
  |_ 1 => pet='CAT'

  |_ 2 => pet='DOG'
```

If we want to specify the enum values that are passed to our test method, we can specify the enum values by setting the value of the @EnumSource annotation's names attribute. Let's ensure that the value: Pet.CAT is passed to our test method.
After we have specified the used enum value, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;

import static org.junit.jupiter.api.Assertions.assertNotNull;

@DisplayName("Pass enum values to our test method")
class EnumSourceExampleTest {

    @DisplayName("Should pass only the specified enum value as a method parameter")
    @ParameterizedTest(name = "{index} => pet=''{0}''")
    @EnumSource(value = Pet.class, names = {"CAT"})
    void shouldPassNonNullEnumValueAsMethodParameter(Pet pet) {
        assertNotNull(pet);
    }
}
```

When we run this test method, we see that JUnit 5 passes only the value: Pet.CAT to our test method as a method parameter:

```
Pass enum values to our test method

|_ Should pass non-null enum values as method parameters

  |_ 1 => pet='CAT'
```

We have now learned how we can use two different argument sources that allow us to pass one method parameter to our test method. However, most of the time we want to pass multiple method parameters to our parameterized test. Next, we will find out how we can solve this problem by using the CSV format.

Passing Method Parameters by Using the CSV Format

If we have to pass multiple method parameters to our parameterized test and the provided test data is used by only one test method (or a few test methods), we can configure our test data by using the @CsvSource annotation. When we add this annotation to a test method, we have to configure the test data by using an array of String objects. When we specify our test data, we have to follow these rules:

- One String object must contain all method parameters of one method invocation.
- The different method parameters must be separated with a comma.
- The values found from each line must use the same order as the method parameters of our test method.

Let’s configure the method parameters that are passed to the sum() method. This method takes three method parameters: the first two method parameters contain two int values and the third method parameter specifies the expected sum of the provided int values.
After we have configured the method parameters of our parameterized test, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Should pass the method parameters provided by the @CsvSource
annotation")
class CsvSourceExampleTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    @CsvSource({
            "1, 1, 2",
            "2, 3, 5"
    })
    void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }
}
```

Even though this looks quite clean, sometimes we have so much test data that it doesn’t make sense to add it to our test class because our test class would become unreadable. Let’s find out how we can load the test data that is passed to the sum() method from a CSV file.

Loading Our Test Data From a CSV File

We can load our test data from a CSV file by following these steps:

First, we have to create a CSV file that contains our test data and put this file to the classpath. When we add our test data to the created CSV file, we have to follow these rules:

- One line must contain all method parameters of one method invocation.
- The different method parameters must be separated with a comma.
- The values found from each line must use the same order as the method parameters of our test method.

The *test-data.csv* file configures the test data that is passed to our test method. This file is found from the *src/test/resources* directory, and its content looks as follows:

```
1,1,2

2,3,5

3,5,8
```

Second, we have to annotate our test method with the @CsvFileSource annotation and configure the location of our CSV file. After we have done this, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Should pass the method parameters provided by the test-data.csv
file")
class CsvFileSourceExampleTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    @CsvFileSource(resources = "/test-data.csv")
    void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }
}
```

We can now pass multiple method parameters to our parameterized test. However, the catch is that the method parameters of our parameterized tests must be supported by the DefaultArgumentConverter class. Its Javadoc states that:
The DefaultArgumentConverter is able to convert from strings to a number of primitive types and their corresponding wrapper types (Byte, Short, Integer, Long, Float, and Double) as well as date and time types from the java.time package.
Next, we will find out how we can solve this problem by using a factory method and a Custom ArgumentsProvider.

Creating the Method Parameters by Using a Factory Method

If all parameterized tests which use the created method parameters are found from the same test class and the logic that creates these method parameters is not "too complex", we should create these method parameters by using a factory method.

If we want to use this approach, we have to add a static factory method to our test class and implement this method by following these rules:

- The factory method must not take any method parameters.
- The factory method must return a Stream, Iterable, Iterator, or an array of Arguments objects. The object returned by our factory method contains the method parameters of all test method invocations.
- An Arguments object must contain all method parameters of a single test method invocation.
- We can create a new Arguments object by invoking the static of() method of the Arguments interface. The method parameters provided to the of() method are passed to our test method when it is invoked by JUnit 5. That's why the provided method parameters must use the same order as the method parameters of our test method.

There are two things I want to point out:

- The factory method must be static only if we use the default lifecycle configuration.
- The factory method can also return a Stream that contain primitive types. This blog post doesn't use this approach because I wanted to demonstrate how we can pass "complex" objects to our parameterized tests.

Let's demonstrate these rules by implementing a factory method that creates the method parameters of the sum() method (we have already used this method in the previous examples). After we have implemented this factory method, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;

import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
@DisplayName("Should pass the method parameters provided by the sumProvider()
method")
class MethodSourceExampleTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }

    private static Stream<Arguments> sumProvider() {
        return Stream.of(
                Arguments.of(1, 1, 2),
                Arguments.of(2, 3, 5)
        );
    }
}
```

After we have implemented this method, we must ensure that its return value is used when JUnit 5 determines the method parameters of our parameterized test. We can do this by annotating our test method with the @MethodSource annotation. When we do this, we must remember to configure the name of the factory method.
After we have made the required changes to our test class, its source code looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Should pass the method parameters provided by the sumProvider()
method")
class MethodSourceExampleTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    @MethodSource("sumProvider")
    void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }

    private static Stream<Arguments> sumProvider() {
        return Stream.of(
                Arguments.of(1, 1, 2),
                Arguments.of(2, 3, 5)
        );
    }
}
```

This approach work relatively well as long as the factory method is simple and all test methods that use the factory method are found from the same test class. If either of these conditions is false, we have to implement a custom ArgumentsProvider.

Creating the Method Parameters by Using a Custom ArgumentsProvider

If the test methods that use our test data are found from different test classes or the logic which creates the required test data is so complex that we don’t want to add it to our test class, we have to create a custom ArgumentsProvider.
We can do this by creating class that implements the ArgumentsProvider interface. After we have created this class, we have to implement the provideArguments() method that returns a Stream of Arguments objects. When we create the returned Stream, we must follow these rules:

- An Arguments object must contain all method parameters of a single test method invocation.
- We can create a new Arguments object by invoking the static of() method of the Arguments interface. The method parameters provided to the of() method are passed to our test method when it is invoked by JUnit 5. That’s why the provided method parameters must use the same order as the method parameters of our test method.

Let’s create a custom ArgumentsProvider which provides the test data that is required by the sum() method. We can do this by following these steps:
First, we have write a custom ArgumentsProvider class which returns the test data that is required by the sum() method.
After we have created a custom ArgumentsProvider class, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;

import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Should pass the method parameters provided by the
CustomArgumentProvider class")
class ArgumentsSourceExampleTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }

    static class CustomArgumentProvider implements ArgumentsProvider {

        @Override
        public Stream<? extends Arguments> provideArguments(ExtensionContext
context) throws Exception {
            return Stream.of(
                    Arguments.of(1, 1, 2),
                    Arguments.of(2, 3, 5)
            );
        }
    }
}
```

By the way, most of the time we don't want to use inner classes. I used an inner class here because this is just an example.

Second, we have to configure the used ArgumentsProvider by annotating our test method with the @ArgumentsSource annotation. After we have done this, the source code of test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;
import org.junit.jupiter.params.provider.ArgumentsSource;

import java.util.stream.Stream;

import static org.junit.jupiter.api.Assertions.assertEquals;

```

```
@DisplayName("Should pass the method parameters provided by the
CustomArgumentProvider class")
class ArgumentsSourceExampleTest {

    @DisplayName("Should calculate the correct sum")
    @ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
    @ArgumentsSource(CustomArgumentProvider.class)
    void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
    }

    static class CustomArgumentProvider implements ArgumentsProvider {

        @Override
        public Stream<? extends Arguments> provideArguments(ExtensionContext
context) throws Exception {
            return Stream.of(
                    Arguments.of(1, 1, 2),
                    Arguments.of(2, 3, 5)
            );
        }
    }
}
```

We can now create our test data by using factory methods and custom ArgumentsProvider classes. However, even though these methods allow us to ignore the limitations of the DefaultArgumentConverter class, sometimes we want to provide our test data by using strings because this helps us to write tests that are easier to read than tests which use factory methods or custom ArgumentsProvider classes.
Next, we will find out how we can solve this problem by using a custom ArgumentConverter.

Using a Custom ArgumentConverter

An ArgumentConverter has only one responsibility: it converts the source object into an instance of another type. If the conversion fails, it should throw an ArgumentConversionException.

Let’s create an ArgumentConverter that can convert a String object into a Message object.
The Message class is a simple wrapper class that simply wraps the message given as a constructor argument. Its source code looks as follows:

```
final class Message {

    private final String message;

    Message(String message) {
        this.message = message;
    }

    String getMessage() {
        return message;
    }
}
```

We can create our custom ArgumentConverter by following these steps:
First, we have to create a class called MessageConverter that implements the ArgumentConverter interface. After we have created this class, its source code looks as follows:

```
import org.junit.jupiter.api.extension.ParameterContext;
import org.junit.jupiter.params.converter.ArgumentConversionException;
import org.junit.jupiter.params.converter.ArgumentConverter;

final class MessageConverter implements ArgumentConverter {

    @Override
    public Object convert(Object source, ParameterContext context) throws
ArgumentConversionException {

    }
}
```

Second, we have to implement the convert() method by following these steps:
1. Throw a new ArgumentConversionException if the source object is not valid. The source object must be a String that is not null or empty.
2. Create a new Message object and return the created object.

After we have implemented the convert() method, the source code of the MessageConverter class looks as follows:

```
import org.junit.jupiter.api.extension.ParameterContext;
import org.junit.jupiter.params.converter.ArgumentConversionException;
import org.junit.jupiter.params.converter.ArgumentConverter;

final class MessageConverter implements ArgumentConverter {

    @Override
    public Object convert(Object source, ParameterContext context) throws ArgumentConversionException {
        checkSource(source);

        String sourceString = (String) source;
        return new Message(sourceString);
    }

    private void checkSource(Object source) {
        if (source == null) {
            throw new ArgumentConversionException("Cannot convert null source object");
        }

        if (!source.getClass().equals(String.class)) {
            throw new ArgumentConversionException(
                    "Cannot convert source object because it's not a string"
            );
        }

        String sourceString = (String) source;
        if (sourceString.trim().isEmpty()) {
            throw new ArgumentConversionException(
                    "Cannot convert an empty source string"
            );
        }
    }
}
```

After we have created our custom ArgumentConverter, we have to create a parameterized test which uses our custom ArgumentConverter. We can create this test by following these steps:
First, we have to create a new parameterized test method by following these steps:

1. Add a new parameterized test method to our test class and ensure that the method takes two Message objects as method parameters.
2. Annotate the test method with the @CsvSource annotation and configure the test data by using the CSV format.
3. Verify that the Message objects given as method parameters contain the same message.

After we have created our test method, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Pass converted Message objects to our test method")
class MessageConverterExampleTest {

    @DisplayName("Should pass same messages as method parameters")
    @ParameterizedTest(name = "{index} => actual={0}, expected={1}")
    @CsvSource({
            "Hello, Hello",
            "Hi, Hi",
    })
    void shouldPassMessages(Message actual, Message expected) {
        assertEquals(expected.getMessage(), actual.getMessage());
    }
}
```

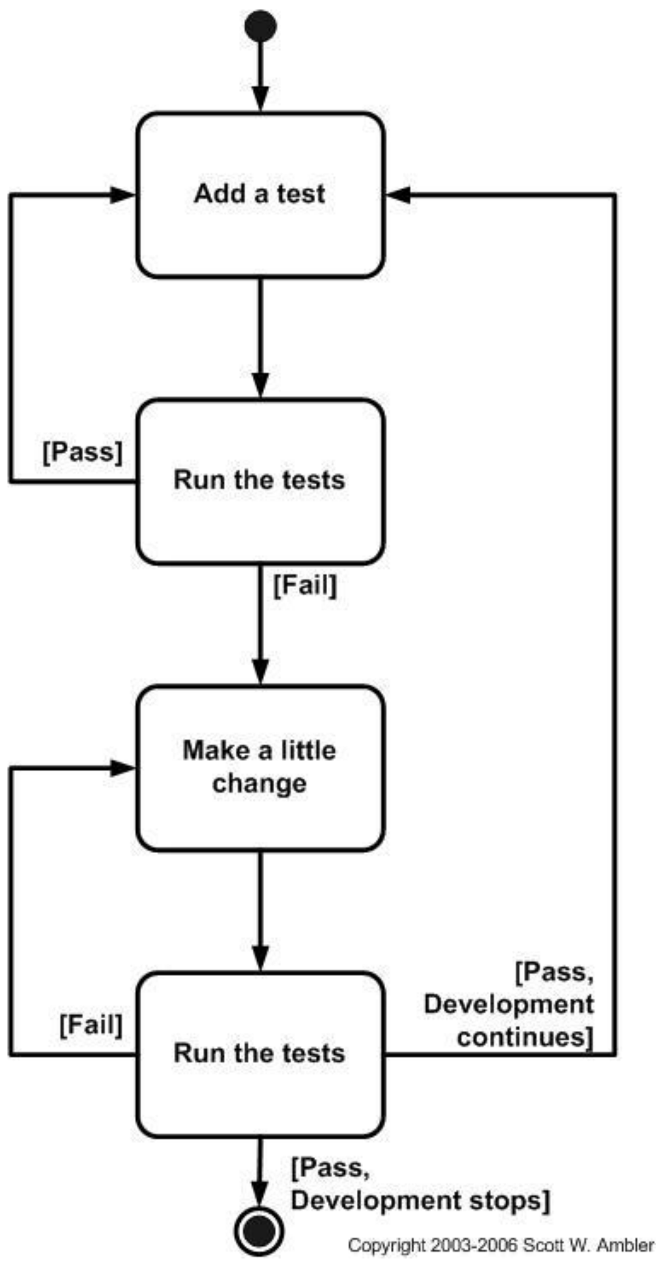
Second, we have to configure the ArgumentConverter that creates the actual method parameters. We can do this by annotating the method parameters with the @ConvertWith annotation. When we do this, we have to configure the used ArgumentConverter by setting the value of the @ConvertWith annotation's value attribute.
After we have configured the used ArgumentConverter, the source code of our test class looks as follows:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.converter.ConvertWith;
import org.junit.jupiter.params.provider.CsvSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

@DisplayName("Pass converted Message objects to our test method")
class MessageConverterExampleTest {

    @DisplayName("Should pass same messages as method parameters")
    @ParameterizedTest(name = "{index} => actual={0}, expected={1}")
    @CsvSource({
            "Hello, Hello",
            "Hi, Hi",
    })
    void shouldPassMessages(@ConvertWith(MessageConverter.class) Message actual,
                            @ConvertWith(MessageConverter.class) Message expected) {
        assertEquals(expected.getMessage(), actual.getMessage());
    }
}
```

TDD

What is TDD?

The steps of test first development (TFD) are overviewed in the UML activity diagram of Figure 1. The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).

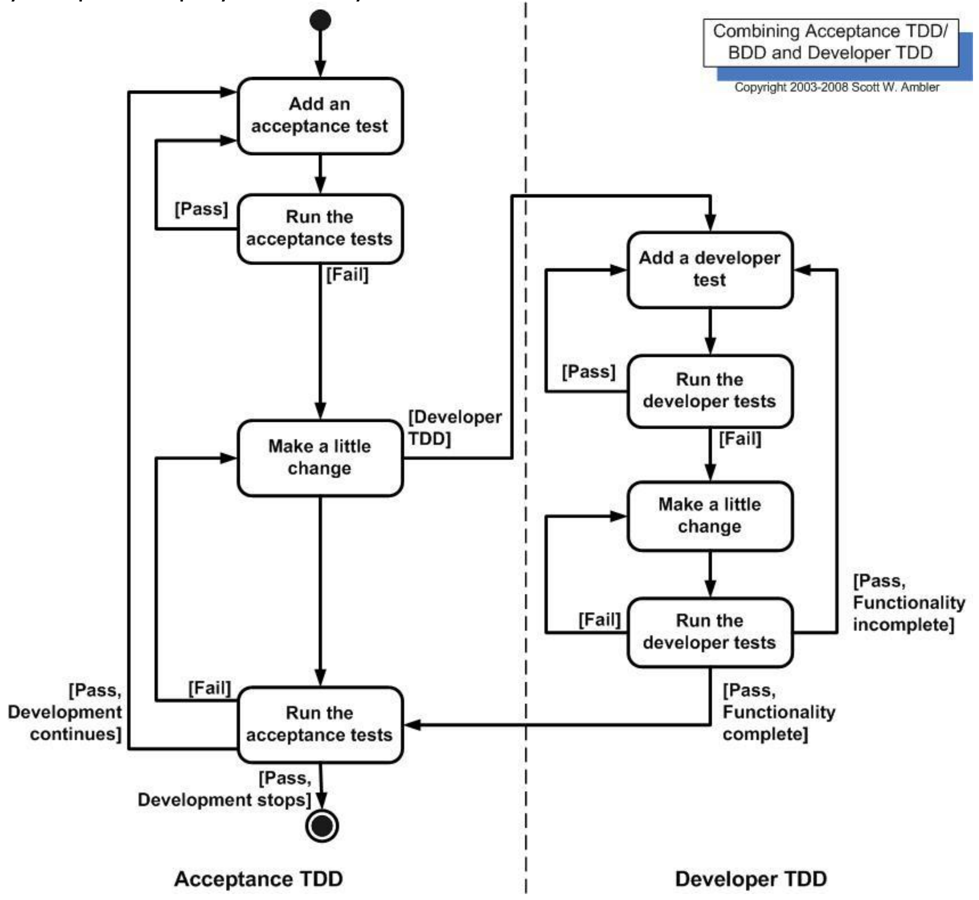

Add a test
Run the tests
[Pass]
[Fail]
Make a little change
Run the tests
[Fail]
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

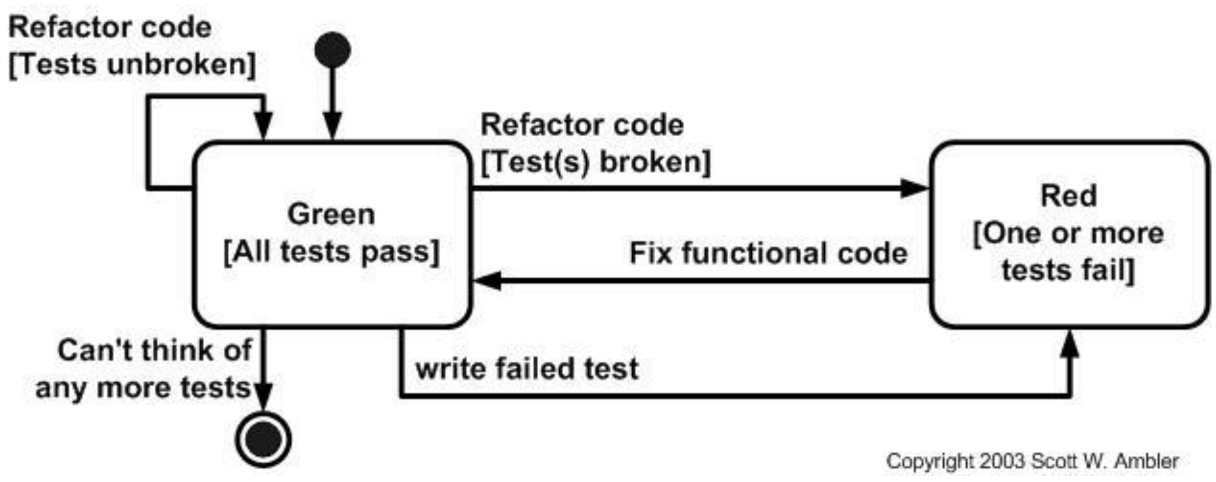
I like to describe TDD with this simple formula:

TDD = Refactoring + TFD.

TDD completely turns traditional development around. When you first go to implement a new feature, the first question that you ask is whether the existing design is the best design possible that enables you to implement that functionality. If so, you proceed via a TFD approach. If not, you refactor it locally to change the portion of the design affected by the new feature, enabling you to add that feature as easy as possible. As a result you will always be improving the quality of your design, thereby making it easier to work with in the future.

Instead of writing functional code first and then your testing code as an afterthought, if you write it at all, you instead write your test code before your functional code. Furthermore, you do so in very small steps – one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn't present. In fact, they refuse to add even a single line of code until a test exists for it. Once the test is in place they then do the work required to ensure that the test suite now passes (your new code may break several existing tests as well as the new one). This sounds simple in principle, but when you are first learning to take a TDD approach it proves require great discipline because it is easy to "slip" and write functional code without first writing a new test. One of the advantages of pair programming is that your pair helps you to stay on track.





History of TDD

Generally, the modern "rediscovery"" of TDD is attributed to Kent Beck, who is also known as the creator of Extreme Programming. It was through both the Extreme Programming and Agile software development movements that TDD came to be widely accepted in the software development community.

It is said that Kent Beck "rediscovered" TDD since a prototype of TDD dates back to the early days of computing in the 1960s. During the mainframe era when program code would be entered onto punch cards, programmers had limited time with the machine and thus would need to maximize the time they had. One documented practice was to write the expected output of whatever operation you were doing before entering the punch cards into the computer. Then when the mainframe would output the results of your program, you could immediately see whether the results you got were correct by comparing the actual output with the expected output that had been documented earlier.

The big difference between modern TDD and those early days is that it used to be a completely manual testing process, whereas the re-birth of TDD in recent years was facilitated due to automated testing. Today TDD refers solely to automated test-driven development.

Modern TDD was first practiced in the Smalltalk community and they used the Smalltalk SUnit suite for their automated testing. However, the Smalltalk community was always quite a small group and while influential, it took years for much of their brilliance to reach the wider industry. Additionally, since Smalltalk never really took off, that also hampered the initial spread of TDD.

It was in the Java community that TDD really started to take off thanks to the JUnit tool. JUnit was a port of SUnit written by Kent Beck and a couple others, and it brought automated testing to Java. At this time, many Java developers who were practicing either Agile or Extreme programming methodologies began to embrace TDD thanks to JUnit. Since then, a class of tools known as XUnit have been created for almost every programming language from PHP, Ruby, Python to JavaScript. Today, regardless of the programming language you are using, you should be able to find the tools and resources to implement TDD as part of your software development process.

WHY PRACTICE TDD ?

Test-Driven Development (TDD) in practice

In the past, usually when a company decided to start a project, they would go by waterfall approach: define the specifications first, then implement features later, then test and maintain. This methodology has many strong points such as tight control, extensive written documents, and approval between each phase, etc.; and is still used at some companies nowadays. Despite those strong points; however, the major disadvantage is that the methodology is highly resistant to changes. Once a software piece moves to the testing phase, it is very difficult to go back.

In an ever-changing world today, this particular drawback may result in great loss of money and trust. We want some methodologies that can allow us to develop software in a timely manner, yet assure its quality. One such methodology is Test-Driven Development (TDD).

TDD is a process that relies on a very short development cycle: first, the developer writes an automation test case for a particular requirement, then writes the **minimum** amount of code to pass that test (DO NOT do any optimization at this stage). After the code passes that test, the developer will refactor the code to acceptable standards, then start working on the next requirement.

There are many guides and examples on the internet to demonstrate how to implement TDD in projects, so you can search them easily.

Opposite to the simple definition and how-to, practicing TDD in reality could be quite intimidating. Among different issues, the most frequent complaint is that TDD slowed down the project. For newcomers, it may look like so. At the first look, your pacing is slowed down, you implement less features than before in the same time period. But do not be scared, in the long run, you actually save time for the project. By writing the test and implementing the code in small chunks, it is very easy to spot the uncertainty in the requirements early, then make changes on the spot.

At Manabie, we implement TDD by following these activities:

- Review features

- Break down features into requirements as small as possible

(Quick note: A requirement is a work order for an engineer who constructs some parts of the system).

- Get a requirement

- Write a test case

- Write minimum code to pass the test case

- Write another test case

- Write more codes to pass test case

- Repeat writing test cases and codes until the requirement is fulfilled

- Continue to implement other requirements until every requirement is fulfilled

- Refactor the whole feature

We want to promote communication between members, that's why we prioritize working code before refactoring. We do not want to spend time on refactoring while requirements are still unstable. We want to know if the implementation is feasible and make changes as soon as possible if necessary. The refactoring will then occur when everything is stable at an acceptable level. This way, we lower the risk of misunderstanding.

When we break down the feature into requirements, ideally, a requirement should contain only 1 test case and a very small chunk of code just to pass that test case. For example, we have a simple feature:

- If A then C, if B then D

We will break this feature into smaller requirements:

- Requirement 1: If A, then C

- Requirement 2: If B, then D

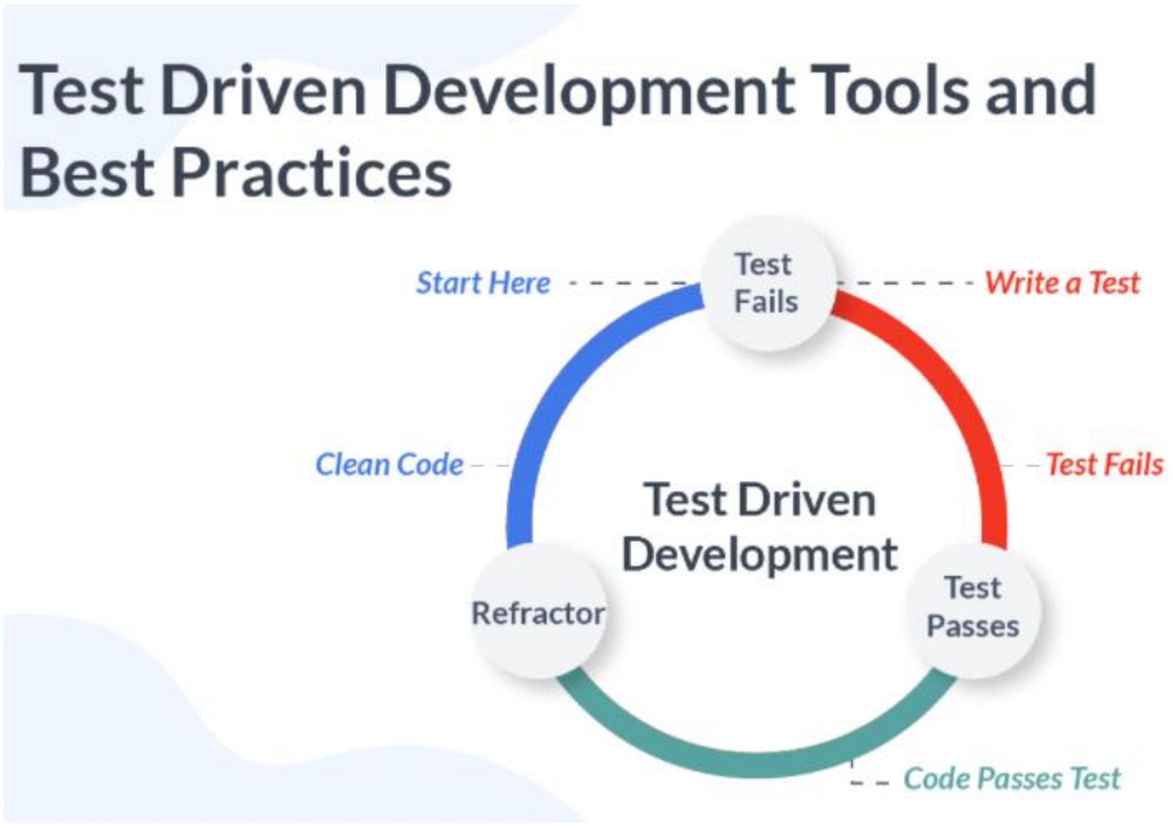
When implementing requirement 1, we do not implement requirement 2 ahead, or doing any other logic, just follow KISS and YAGNI principles, write test and code to meet the exact requirement 1.

If mistakes are found, like no other case handling on the example above, our developers will ask the Product Manager to fix the feature and add more requirements for it later. Of course, in real projects, we often need more than 1 test case to cover a requirement. In this case, developers will write just a minimum amount of code to pass the tests one by one and do not skip any step.

At first, TDD can be irritating. However, when we look back, more than just code, we deliver a set of test cases, which are basically requirements in the form of test codes. Those test cases are way more detailed than any design document, since they tell exactly how software responds in every specific case. If we reread everything, we can write a whole detailed document about a specific feature we implemented. If there is a need for such a document, we should write the test cases in Behavioral-Driven Development style, which will save a lot of effort documenting.

So far, we only discuss TDD with unit tests, and people involved in this methodology are developers only. When there are more people involved in development (customers, QAs, etc.), simple TDD like what we discussed will not work. In that case, Acceptance Test Driven Development (ATDD) or Behavioral-Driven Development (BDD) should be considered. The principle is the same: test-first, but they focus on system-scale, rather than feature implementation.

Testing frameworks and Tools



What is Test Driven Development?

Software testing plays a vital role in the life cycle of software and Test Driven Development. It is imperative to identify bugs and errors during software development and increase the quality of the product. Therefore, one must focus on software testing. There are many approaches and Test Driven approach is one of them.

Test Driven Development is a key practice for extreme programming; it suggests that the code is developed or changed exclusively by the unit testing. Test Driven Development (TDD) is a software-driven process which includes test-first development. It means that the developer first writes a fully automated test case before writing the production code to fulfil that test and refactoring. Benefits of Adopting Test Driven Development (TDD) –

- Development expenses are reduced
- Improved Code Quality
- Quality is enhanced
- Shortened Time to Market

How Test Driven Development (TDD) Works?

- Firstly, add a test.
- Run all the tests and see if any new test fails.

- Update the code to make it pass the new tests.
- Rerun the test and if they fail then refactor again and repeat.

Benefits of Test Driven Development

- It gives a way to think through one's requirements or design before the developer writes functional code.
- It is a programming technique that enables the developer to take a small step during building software.
- It is more productive as compared attempting to code in giant steps.
- Consider an example, developer write some code, then compile it, and then test it, maybe there are chances of failure. In this case, it becomes easy to find and fix those defects if a developer had written two new lines of code than a thousand.

A Test Driven Development is – The most efficient and attractive way to proceed in smaller and smaller steps.

Following Test Driven Development means –

- Fewer Bugs.
- Higher quality software.
- Focus on single functionality at a given point in time.

Why Test Driven Development Matters?

- **Requirements –** Drive out requirement issues early (more focus on requirements in depth).
- Rapid Feedback – Many small changes Vs. One significant change.
- Values Refactoring – Refactor often to lower impact and risk.
- Design to Test – Testing driving good design practice.
- Tests as information – Documenting decisions and assumptions.

Test Driven Development helps the programmer in several ways, such as –

- Improve the code.
- Side by side, increasing the programmer's productivity.

Using Test Driven Development concept in one's programming skills –

- Will save developer's time which is getting wasted for rework.
- Able to identify the error/problem quicker and faster.
- The programmer will be able to write small classes which will be focused only on a single functionality instead of writing the big classes.
- Whenever the code base gets more prominent, it becomes tough to change and debug the code. Moreover, there is a high chance of the code being messed up.

But, if developers are using Test Driven Development technique –

- Means developers have automated tests.
- Writing the test cases for the program which is a safe side for the programmers.
- It becomes easy to view what the error is, where it is and how it is paralyzing one's code.

Best Practices to Adopt Test Driven Development

- **Road Map –** One of the best practice is to clear out with thought and further break it down into the test case. Follow the red-green approach to build the test case. The first step is to create the red test and after exposing all the problem related to code, make some changes and make it a green test.
- **Implementation –** It is essential to implement both source code and test case separately. For both implementation and testing, there should be two directories. In every programming language, there should be different packages for both. Consider an example, in the case of Java "src/main/java" is used for implementation and on the other hand "src/test/java" is used for testing.
- **Structure –** Structure for writing test cases should be correct. It is common practice to write the test class with the same name used in production/implementation class, but there should be a change in the suffix. Consider an example; if the implementation/production class is "Student," then the test class should be "StudentTest." And similarly in case of methods, test methods are written with the same name as of production methods, but there should be a change in the prefix, like if the method name is "display student name," then in testing it should be "testDisplayStudentName."

Top Test Driven Development Tools

There are many tools available for testing and improving the overall design and implementation of the software system. Some of the most common testing tools are listed below –

JUnit for Unit Tests

Junit is a unit testing framework designed for Java programming language. Unit tests are the smallest elements in the test automation process. With the help of unit tests, the developer can check the business logic of any class. So JUnit plays a vital role in the development of a test-driven development framework. It is one of the families of unit testing frameworks which is collectively known as the xUnit that originated with SUnit.

JMeter for Load/Performance Testing

Apache JMeter may be used to test performance both on static and dynamic resources, Dynamic Web applications (Mainly for Load/Performance testing). It is used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.

Features of Apache JMeter –

Ability to load and performs tests on many different applications/server/protocol types, some of them are listed below –

- Web – HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET)
- SOAP/REST Webservices
- FTP
- Database via JDBC
- LDAP
- Message-oriented middleware (MOM) via JMS
- Mail – SMTP(S), POP3(S) and IMAP(S)
- Native commands or shell scripts
- TCP
- Java Objects

Mockito for Rest API Testing

Mockito designed as an open source testing framework for Java which is available under an MIT License. Mockito allows programmers to create and test double objects (mock objects) in automated unit tests for Test-driven Development (TDD). In simple words, Mockito is a framework that developers specifically use to write certain kind of tests efficiently.

There are also many other tools and frameworks available. It depends on the type of programming language, according to the language developer can pick up an appropriate tool and framework for Test Driven Development in Golang.

TESTS INSIGHTS

Test-driven development is a funny thing. Many developers, particularly the more experienced ones, hate the idea of writing tests for every small bit of behavior they code. It’s hard enough to get these developers to even try TDD in the first place (“I’m doing just fine, thank you”), and a small bit of exposure to TDD is usually not enough to convince people that its benefits outweigh the discipline it demands. A short demo or kata might impart the mechanics of TDD, but it will only rarely produce the itch to continue.

It takes time and firsthand experience for TDD to get under people’s skin to the point where they enjoy scratching out a test and regret when they cannot. Twenty years ago, we called this becoming “test-infected.”

Back in 1999, I didn’t think writing tests for my code sounded at all enjoyable. Still, I forced myself to play with TDD (then called “test-first design,” or TfD) for a few weeks. I remember initially thinking that it seemed so backward. But I slowly began to feel the gratification it created: I realized that not only did my software always work as expected, but also that I could clean up the bits of code that I wasn't proud of.

Countless developers have undergone the transformation from skeptic to test-infected over the past twenty or so years. They've matured in their practice of TDD, both individually and as a community. Some have advanced to the mastery level, where they understand the occasional times when they can produce code confidently without TDD. But I warrant that all of these test-infected would say that they would never give up the practice.

We Do TDD

Dave Schinkel created the site We Do TDD to capture the passion behind infected TDD practitioners — as well as to know where he could seek employment with like-minded developers. The site has links to three dozen companies and individuals practicing TDD; more than a third are located outside the United States.

Over two dozen of these parties provided answers to numerous questions about TDD:

- How did you learn TDD?
- What TDD learning resources do you recommend?
- How do you start to test-drive a feature?
- How do you refactor?
- How has TDD helped you design better code?
- How has TDD benefited your customers?
- What are some of the challenges you've faced with TDD?

All told, Dave asks more than a couple of dozen questions about team, environment, and development practices. Many respondents supplied detailed answers that provide great insights and enlightenment about TDD.

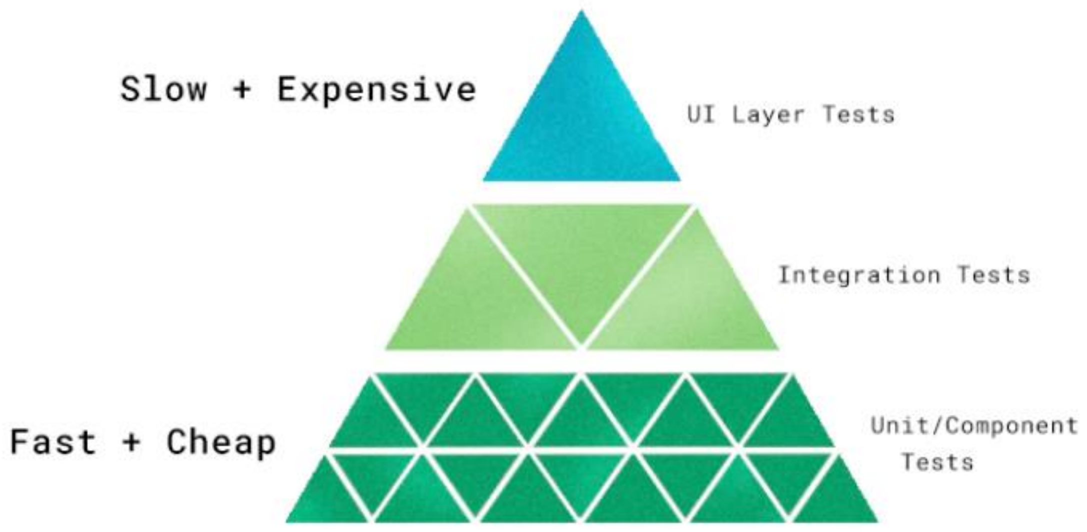
While many of the respondents provide training and coaching services around TDD, many are shops that create high-quality production software using TDD. You'll no doubt recognize many of the names of people and companies providing testimonials at We Do TDD, but you'll also discover some new places that might even make you want to work there.

Test-driven development using mocking and stubbing

In How and where to segregate test environments, I talked about building a structured path to production: which tests to include, when to do them, and why. In this post, we'll get into exactly how to do each kind of test.

We'll cover the techniques of **mocking and stubbing**, and **test-driven development** to help each testing layer. First, let's review a concept from the previous post: the test pyramid. This helps illustrate the difference between different kinds of tests and when it's advantageous to do them.

Unit or component tests (shown here at the bottom of our pyramid) are inexpensive and fast to perform. Rely heavily on these. Only once you've exhausted what these tests can do should you move on to more expensive tests (in both time and resources), such as integration tests, and UI layer tests.



In this series, I will cover an assortment of testing tools that should be in every developer's toolbox, and go over when, why, and how to use them. I'll cover testing across the layers of the pyramid, as well as the concepts of mocking, stubbing, and contract testing. In the second piece of this series, we'll get into test-driven development and behavior-driven development (TDD and BDD).

What is mocking and stubbing used for?

A lot of people think that mocking and stubbing are used just for unit and component tests. However, I want to show you how mock objects or stubs can be used in other layers of testing as well.

What is mock testing?

Mocking means creating a fake version of an external or internal service that can stand in for the real one, helping your tests run more quickly and more reliably. When your implementation interacts with an object's properties, rather than its function or behavior, a mock can be used.

What is stub testing?

Stubbing, like mocking, means creating a stand-in, but a stub only mocks the behavior, but not the entire object. This is used when your implementation only interacts with a certain behavior of the object.

A great blog post that covers the difference between mocking and stubbing can be found [here].(https://martinfowler.com/articles/mocksArentStubs.html){: target="_blank" rel="noreferrer noopener"}

Let's discuss how we can apply these methods to improve our testing in all levels of the pyramid above.

Using mocking and stubbing in unit + component tests

I recommend mocking or stubbing when your code uses external dependencies like system calls, or accessing a database. For example, whenever you run a test, you're exercising the implementation. So when a `delete` or `create` function happens, you're letting it create a file, or delete a file. This work is not efficient, and the data it creates and deletes is not actually useful. Furthermore, it's expensive to clean up, because now you have to manually delete something every time. This is a case where mocking/stubbing can help a lot.

Using mocks and stubs to fake the external functionality help you create tests that are independent. For instance, say that the test writes a file to /tmp/test_file.txt and then the system under the test deletes it. The problem then is not that the test is not independent; it is that the system calls take a lot of time. In this instance, you can stub the file system call's response, which will take a lot less time because it immediately returns.

Another benefit is that you can reproduce complex scenarios more easily. For instance, it is much easier to test the many error responses you might get from the filesystem then to actually create the condition. Say that you only wanted to delete corrupt files. Writing a corrupt file can be difficult programmatically, but returning the error code associated with a corrupt file is a matter of just changing what a stub returns.

Mock and stub testing example

```
def read_and_trim(file_path)

          return os.open(file_path).rstrip("\n") #method will call system call to look for the file from the
given file path and read the content from them and removing new line terminator.
```

The code above interacts with Python's built-in open function which interacts with a system call to actually look for the file from the given file path. Which means wherever and whenever you run the test for that function:

1. You will need to ensure that the file that the test will be looking for exists; when it does not exist, the test fails.
2. The test will need to wait for the system call's response; if the system call times out, the test fails.

Neither case of failure means your implementation failed to do its job. These tests are now neither isolated (since they're dependent on the system call's response) nor efficient (since the system call connection will take time to deliver the request and response).

The test code for the implementation above looks like this:

```
@unittest.mock.patch("builtins.open", new_callable=mock_open, read_data="fake file content\n")

def test_read_and_trim_content(self, mock_object):



    self.assertEqual(read_and_trim("/fake/file/path"), "fake file content")

    mock_object.assert_called_with("/fake/file/path")
```

We are using a Python mock patch to mock the built-in open call. In this way, we are only testing what we actually built.

Another good example of using mocks and stubs in unit testing is faking database calls. For example, let's say you are testing whether your function deletes the entity from a database. For the first test, you manually create a file so that there's one to be deleted. The test passes. But then, the second time, someone else (who isn't you) doesn't know that they have to manually create the entity. Now the test fails. There was no file to delete since they didn't know they had to create the entity, so this is not an independent test.

In cases like these, you'll want to prevent modifying the data or making operating system calls to remove the file. This will prevent tests from being flaky whenever someone accidentally fails to create test data.

Mocking and stubbing of internal functions

Mocks and stubs are very handy for unit tests. They help you to test a functionality or implementation independently, while also allowing unit tests to remain efficient and cheap, as we discussed in our previous post.

A great application of mocks and stubs in a unit/component test is when your implementation interacts with another method or class. You can mock the class object or stub the method behavior that your implementation is interacting with. Mocking or stubbing the other functionality or class, and therefore only testing your implementation logic, is the key benefit of unit tests, and the way to reap the biggest benefit from performing them.

Note: *Your tests should grow with your code. Since the unit test is focused more on implementation details than the overall functionality of the feature, it's the test that will change the most over time. It follows that when you are using a lot of mocked data in your testing, your mocking has to evolve the same way that your code evolves. Otherwise, it can potentially lead to unexpected bugs in the system. Tests aren't something you write once and expect to always work. As you change your code and refactor, it's your responsibility to maintain and evolve your tests to match.*

Mocking in integration testing

With integration tests, you are testing relationships between services. One approach might be to get all the dependent services up and running for the testing environment. But this is unnecessary. It can create a lot of potential failure points from services you do not control, adding time and complexity to your testing. I recommend narrowing it down by writing a few service integration tests using mocks and stubs. I'll show you how this makes your test suite more reliable.

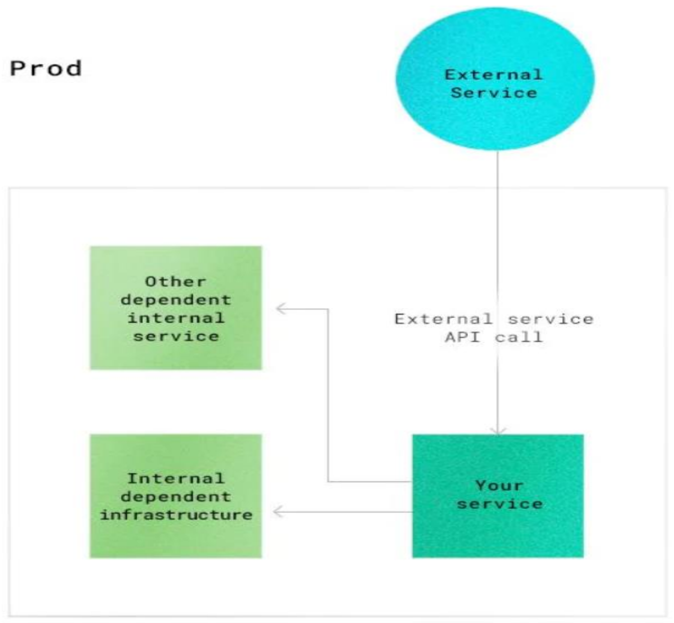
In integration testing, the rules are different from unit tests. Here, you should only test the implementation and functionality that you have the control to edit. Mocks and stubs can be used for this purpose. First, identify which integrations are important. Then, you can decide which external or internal services can be mocked.

Let's say your code interacts with the GitHub API, like in the example below. Since you personally can't change how the GitHub API is responding from your request call, you don't have to test it. Mocking the expected GitHub API's response lets you focus more on testing the interactions within your internal code base.

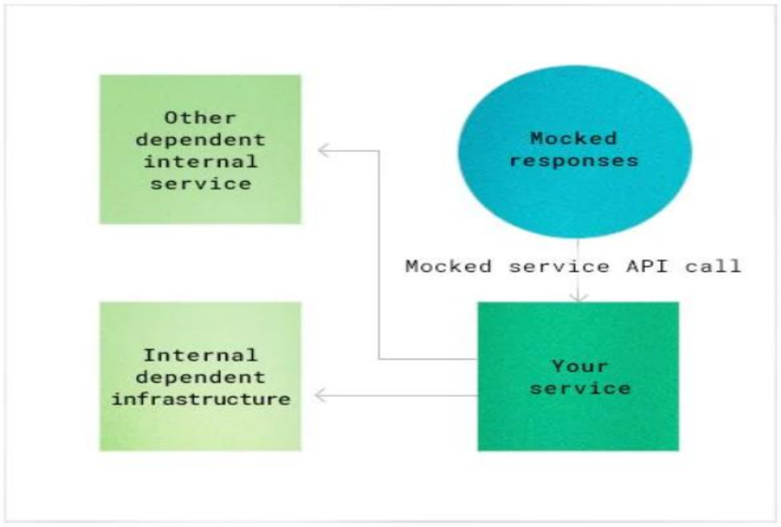
```
@unittest.mock.patch('Github')

def test_parsed_content_from_git(self, mocked_git):

   expected_decoded_content = "b'# Sample Hello World\n\n> How to run this app\n\n- installation\n\n
dependencies\n"

   mocked_git.get_repo.return_value = expected_decoded_content



   parsed_content = read_parse_from content(repo='my/repo',

                                           file_to_read='README.md')



   self.assertEqual(parsed_content['titles'], ['Sample Hello World'])
```

In the test code above, the read_parse_from_content method is integrated with the class that parses the JSON object from the GitHub API call. In this test, we are testing the integration in between two classes.

Since we are using a mock in the test above, your test will be faster and less dependent by avoiding making the call to the GitHub API. This will also save time and effort by not needing internet access for the environment that will run the test. However, in order for you to have reliable testing while mocking the dependent external services, it's extremely important for you to understand how external dependencies will behave in the real world. For example, if the expected_decoded_content in the code example above is not how GitHub returns the repo file content, incorrect assumptions from the mocked test can lead to unexpected breakage. Before writing the test that will have the mocked response, it's best to make the actual snapshot of the external dependency call and use it as a mocked response. Once you have created the mocked response with the snapshot, that should not change often since the Application Programming Interface should almost always be backward compatible. However, it is important to validate the API regularly for the occasional unexpected change.


Prod
External Service
Other dependent internal service
External service API call
Internal dependent infrastructure
Your service

Test
Environment


Other dependent internal service
Mocked responses
Mocked service API call
Internal dependent infrastructure
Your service

Mocks and stubs in contract-based testing (in a microservices architecture)

When two different services integrate with each other, they each have “expectations,” i.e. standards about what they’re giving and what they expect to get in return. We can think of these as contracts between integrated endpoints. Because of this standardization, contract tests can be used to test integrations.

Let’s walk through an example. As I mentioned, the version-tagged API should not change often, possibly not ever. For any API you choose, you will generally be able to find documentation about that API, and what to expect from it. And when you decide to use a certain version of an API, you can rely on the return of that API call. This is the presumed contract between the engineers who provide the API and the engineers who will use its data.

You can use the idea of contracts to test internal services as well. When testing a large scale application using microservices architecture it could be costly to install the entire system and infrastructure. Such applications can benefit greatly from using contract testing. In the testing pyramid, contract testing sits in between the unit/component testing and integration testing layers, depending on the coverage of the contract testing in your system. Some organizations utilize contract testing to completely replace end-to-end or functional testing.

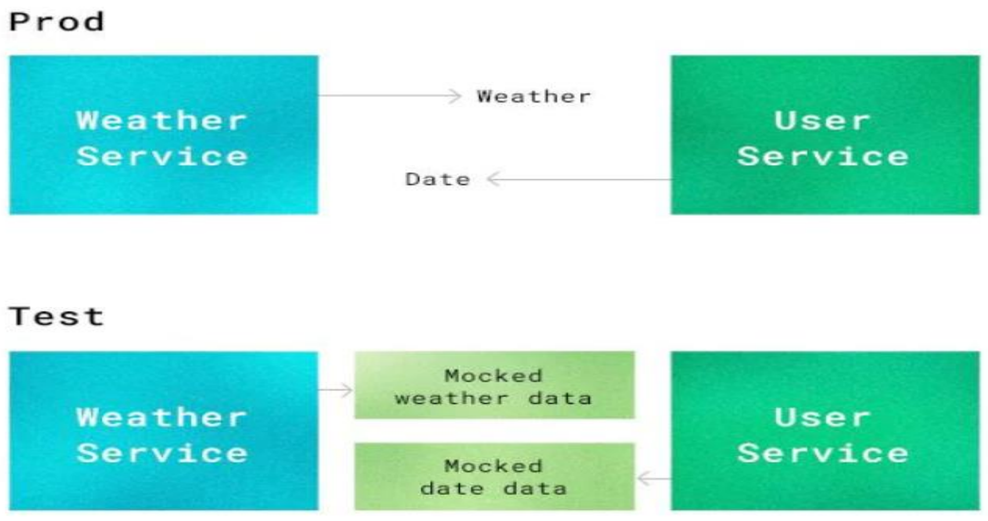
Contract-based testing can cover two important things:

1. Checking the connectivity of end point that has been agreed upon
2. Checking the response from the endpoint with a given argument

As an example, let’s imagine a weather-reporting application involving a weather service interacting with a user service. When the user service connects to the endpoint of the weather service with the date (the request), the user service processes the date data to get the weather for that date. These two services have a contract: the weather service will maintain the endpoint to be always accessible by the user service and provide the valid data that the user service is requesting, and in the same format.

Now, let's take a look at how we can utilize mocks and stubs in the contract test. Instead of the user service making the actual request call to the weather service in the test, you can create a mocked response. Since there is a contract between two services, the endpoint and response should not change. This will free both services from depending on each other during tests, allowing tests to be faster and more reliable.

In the last post, we talked about running different tests in different environments and how sometimes, it can be useful to run the same test in a different environment with a different configuration. Contract tests are one of the great examples of the latter case. We can achieve different goals when running contract tests in different environments with different configurations. When it's a lower layer environment such as Dev or CI, running the test with a mocked contract would serve the purpose of testing our internal implementation within the constraints of the environment. However, when it goes to an upper layer environment such as QA or Staging, the same test can be used without a mocked contract but with the actual external dependency connection. Mbtest is one tool that can help with the kind of contract testing and mocking response I explained above.



We've taken a look at examples of different layers of testing using mocks and stubs. Now let's recap why they are useful:

1. Tests with mocks and stubs go faster because you don't have to connect with external services. There's no delay waiting for them to respond.
2. You have the flexibility to scope the test to cover just the parts you can control and change. With external services, you are powerless in the case that they're wrong or the test fails. Mocking ensures you are scoping tests to work that you can do – and not giving yourself problems you can't fix.
3. Mocking external API calls helps your test to be more reliable
4. Contract testing empowers service teams to be more autonomous in development

In Test-driven development and behavior-driven development, we'll explore the principles of test-driven development (TDD) and behavior-driven development (BDD), and see how they can improve outcomes for everything from functional testing to unit testing.

MOCKITO OVERVIEW

Mockito

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. Mockito can do the same very easily, as its name suggests.

Benefits of Mockito

- **No Handwriting** – No need to write mock objects on your own.
- **Refactoring Safe** – Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
- **Return value support** – Supports return values.
- **Exception support** – Supports exceptions.
- **Order check support** – Supports check on order of method calls.
- **Annotation support** – Supports creating mocks using annotation.

Consider the following code snippet.

```
package com.tutorialspoint.mock;

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

import static org.mockito.Mockito.*;

public class PortfolioTester {
   public static void main(String[] args){

      //Create a portfolio object which is to be tested
      Portfolio portfolio = new Portfolio();

      //Creates a list of stocks to be added to the portfolio
      List<Stock> stocks = new ArrayList<Stock>();
      Stock googleStock = new Stock("1","Google", 10);
      Stock microsoftStock = new Stock("2","Microsoft",100);

      stocks.add(googleStock);
      stocks.add(microsoftStock);

      //Create the mock object of stock service
      StockService stockServiceMock = mock(StockService.class);

      // mock the behavior of stock service to return the value of various stocks
      when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);
      when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);

      //add stocks to the portfolio
      portfolio.setStocks(stocks);

      //set the stockService to the portfolio
      portfolio.setStockService(stockServiceMock);

      double marketValue = portfolio.getMarketValue();

      //verify the market value to be
      //10*50.00 + 100* 1000.00 = 500.00 + 100000.00 = 100500
      System.out.println("Market value of the portfolio: "+ marketValue);
   }
}
```

Let's understand the important concepts of the above program. The complete code is available in the chapter ***First Application***.

- **Portfolio** − An object to carry a list of stocks and to get the market value computed using stock prices and stock quantity.
- **Stock** − An object to carry the details of a stock such as its id, name, quantity, etc.
- **StockService** − A stock service returns the current price of a stock.
- **mock(...)** − Mockito created a mock of stock service.
- **when(...).thenReturn(...)** − Mock implementation of getPrice method of stockService interface. For googleStock, return 50.00 as price.
- **portfolio.setStocks(...)** − The portfolio now contains a list of two stocks.
- **portfolio.setStockService(...)** − Assigns the stockService Mock object to the portfolio.
- **portfolio.getMarketValue()** − The portfolio returns the market value based on its stocks using the mock stock service.