JAVA FULL STACK DEVELOPMENT PROGRAM

Week 9 Day 5: Spring boot, JUnit

OUTLINE

- Spring Boot
  - Introduction
  - Auto-Configuration
- JUnit
  - Spring Testing
  - Mockito

SPRING

- Spring is a light-weighted enterprise level develop framework; A replacement of Enterprise JavaBean (EJB)

- It achieves the functionality of EJB by implementing POJO together with **Dependency Injection** & **Aspect Oriented Programming**

- Although the component code is light-weighted, its configurations are heavy-weighted. It needs a huge amount of configurations

- Spring 2.5 introduces annotation based scan; Spring 3.0 introduces Java based configuration as an alternative to the standard XML file. These features make the configuration work eaiser but not enough

- We still need to configure lots of XML files when dealing with other transaction managements such as Spring MVC, or Thymeleaf Web views

SPRING BOOT

- Spring Boot makes it easy to create stand-alone, production-grade Spring-based Applications that you can run.
- It takes an opinionated view of the Spring platform and third-party libraries, so that you can get started with minimum configuration
- Goal of Spring Boot
  - Provide a radically faster and widely accessible getting-started experience for all Spring development.
  - Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults. (Easy to override the default configuration)
  - Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
  - Absolutely no code generation and no requirement for XML configuration. (Avoid Configuration Boiler Template)

SPRING BOOT

- Benefits:

1. It’s easy to develop Spring based application
2. Spring Boot takes less time, improve overall productivity
3. No need to write template code, XML configuration or redundant annotations
4. Easy to integrate with other Spring features such as Spring JDBC, Spring ORM, Spring Data, Spring Security and etc
5. Provides embedded HTTP server such as Tomcat and Jetty, easy for web app development and test.

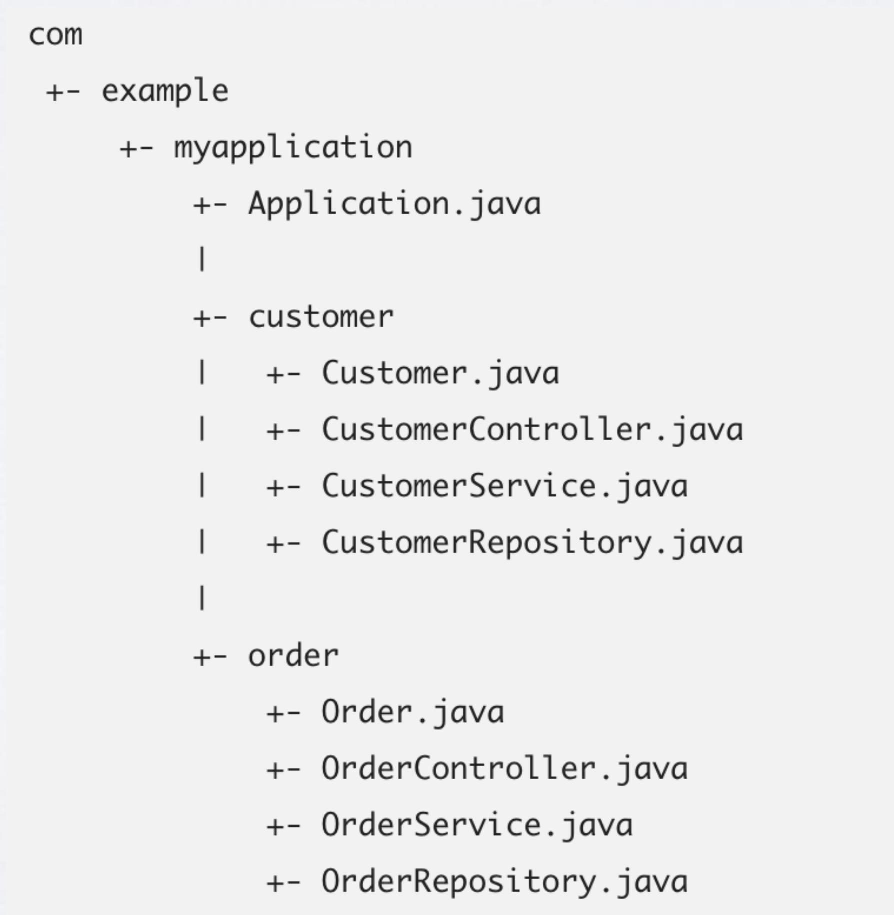
SPRING BOOT

- How does it work?
  - The entry point of the Spring boot application is a class with @SpringBootApplication and a main method calls SpringApplication.run()
  - @SpringBootApplication is the core of Spring Boot auto configuration, it is a combined annotation
    - @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
    - @ComponentScan: enable @Component scan on the package where the application is located
    - @Configuration: allow to register extra beans in the context or import additional configuration classes

SPRING BOOT

- Auto-Configuration

  - The @SpringBootApplication annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items



```
com
 +- example
     +- myapplication
         +- Application.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

@CONFIGURATION

- Included in @SpringBootConfiguration
- Indicates the class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions
- An alternative to XML configuration

@COMPONENTSCAN

- Automatically scan and load bean definition, finally add these definitions to container
- We can use **basePackages** property to specify scan range. By default, Spring framework will start from the class where it is declared
- Will scan current package and all sub packages

@ENABLEAUTOCONFIGURATION

- Import all the qualified beans to Ioc container
- @Import(AutoConfigurationImportSelector.class)
- AutoConfigurationImportSelector will dynamically import the beans

SPRING BOOT

- Auto-Configuration
  - Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added
  - Gradually Replacing Auto-configuration
    - Auto-configuration is non-invasive — For example, if you add your own DataSource bean, the default embedded database support backs away
  - Disabling Specific Auto-configuration Classes
    - If you find that specific auto-configuration classes that you do not want are being applied, you can use the exclude attribute of @EnableAutoConfiguration to disable them

```
@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

- Here is the link for all auto-configuration classes: https://docs.spring.io/spring-boot/docs/2.2.3.RELEASE/reference/html/appendix-auto-configuration-classes.html#auto-configuration-classes

TESTING

- Testing is an important part during software development
- With testing, we can check if our code works as the way we expected.
- There are two major types of testing:
  - Unit test
  - Integration test

UNIT TEST

- Unit testing is one of important part during software development
  - Agile Development — When you add more and more features to a software, you sometimes need to change old design and code
  - Documentation — Unit test is another place where the new hires can learn the logic of the application
  - TDD(Test Driven Design) — When writing cases first, it forces developer to think through the business logic clearly and come up with different corner cases
  - Quality of Code — When we have good coverage of unit test case for application, the quality of code is higher.

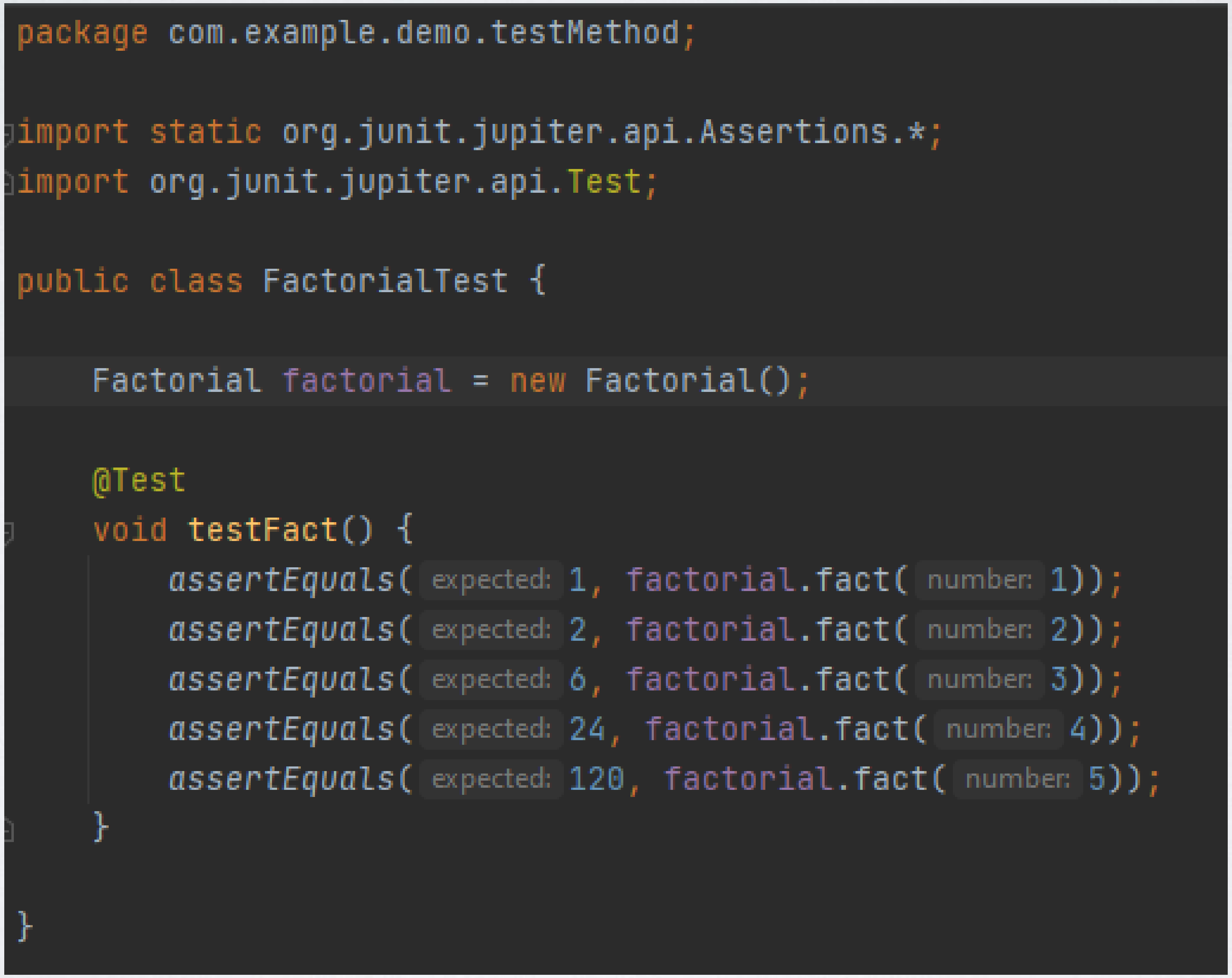
JUNIT

- JUnit 4 and JUnit 5
  - JUnit 4 was divided into modules that comprise JUnit 5
    - JUnit Platform – this module scopes all the extension frameworks we might be interested in test execution, discovery, and reporting
    - JUnit Vintage – this module allows backward compatibility with JUnit 4 or even JUnit 3
  - JUnit 5 support Java 8 features
  - Differences in annotations:
    - *@Before* annotation is renamed to *@BeforeEach*
    - *@After* annotation is renamed to *@AfterEach*
    - *@BeforeClass* annotation is renamed to *@BeforeAll*
    - *@AfterClass* annotation is renamed to *@AfterAll*
    - *@Ignore* annotation is renamed to *@Disabled*

TEST METHOD

- Let's say we implemented a Fact() method which will take a long, and return the factorial of this long number. How can we test it?
- We could write a main() method to do it, or we create a controller to output the result
- These methods have many drawbacks. With main() method, we cannot separate the testing code; with controller, we are not able to create a general testing case which can be reused

CREATE TEST CLASS



```java
package com.example.demo.testMethod;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class FactorialTest {

    Factorial factorial = new Factorial();

    @Test
    void testFact() {
        assertEquals( expected: 1, factorial.fact( number: 1));
        assertEquals( expected: 2, factorial.fact( number: 2));
        assertEquals( expected: 6, factorial.fact( number: 3));
        assertEquals( expected: 24, factorial.fact( number: 4));
        assertEquals( expected: 120, factorial.fact( number: 5));
    }

}
```

JUNIT

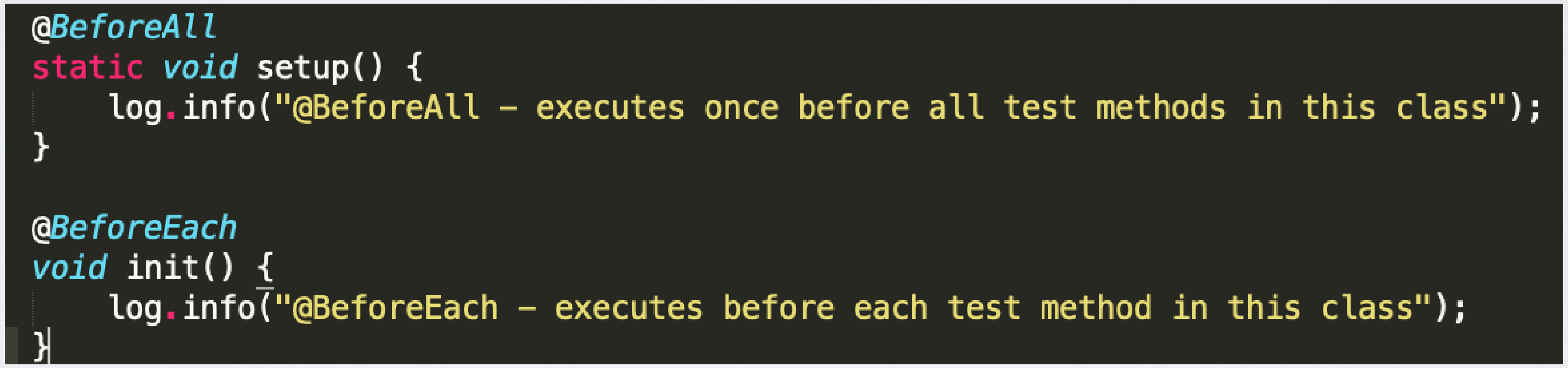
- @Test – tell system this method is a testing method
- assertEquals: expect the two input arguments are the same
    - assertTrue: expect the input = true
    - assertFalse: expect the input = false
    - assertNotNull: expect the input is not null object
    - assertArrayEquals: expected the two input array are the same

FIXTURE

- In unit testing, we usually code multiple @Test to group methods together
- During testing, we usually need to initialize certain objects. Once the testing is done, we may also need to remove those objects
- If we create and remove these object each time we have a @Test method, we will do lots of redundant works
- Instead, JUnit provides us with code to setup the testing environment and remove the objects after testing. These are called fixtures

JUNIT

- Basic Annotations
  - @BeforeAll and @BeforeEach



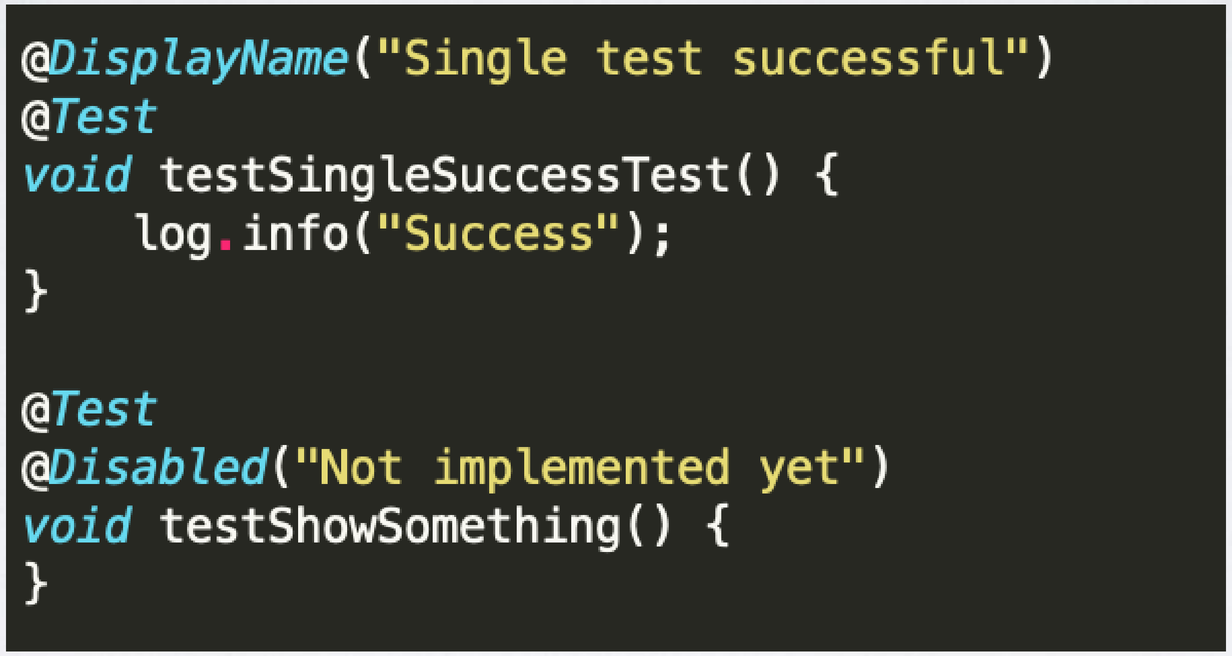
```java
@BeforeAll
static void setup() {
    log.info("@BeforeAll - executes once before all test methods in this class");
}

@BeforeEach
void init() {
    log.info("@BeforeEach - executes before each test method in this class");
}
```

  - Important to note is that the method with *@BeforeAll* annotation needs to be static, otherwise the code will not compile.

JUNIT

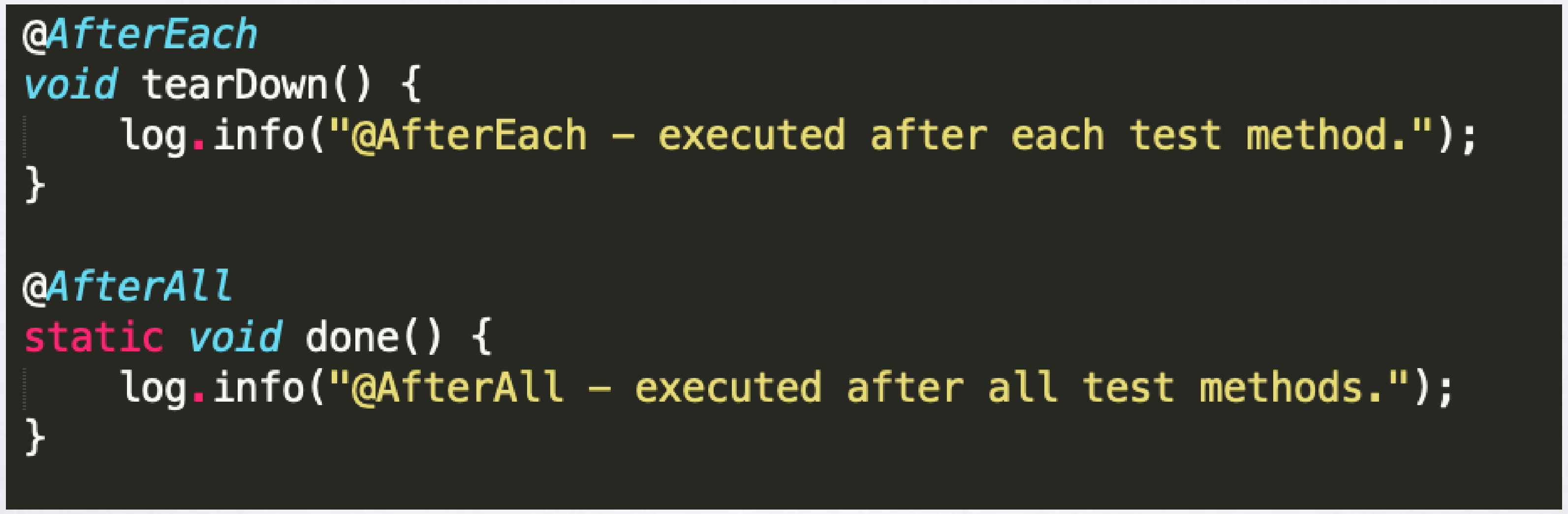
- @DisplayName and @Disabled



```
@DisplayName("Single test successful")
@Test
void testSingleSuccessTest() {
    log.info("Success");
}

@Test
@Disabled("Not implemented yet")
void testShowSomething() {
}
```

- @AfterEach and @AfterAll

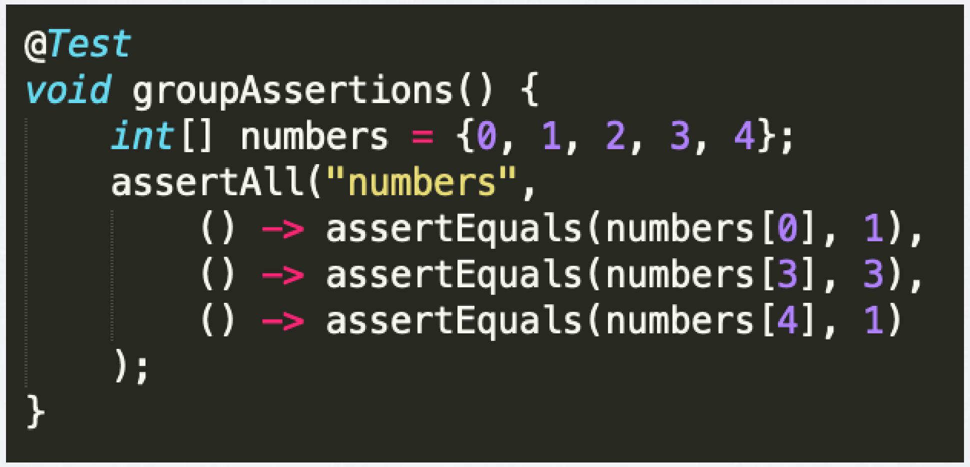


```
@AfterEach
void tearDown() {
    log.info("@AfterEach - executed after each test method.");
}

@AfterAll
static void done() {
    log.info("@AfterAll - executed after all test methods.");
}
```

JUNIT

- Assertion — Assertion is the key to validate if the output from the code is expected
  - Here is the full list of supported assertion statement: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html
- From JUnit 5, we can use lambdas in assertion
- From JUnit 5, It is also now possible to group assertions with *assertAll()* which will report any failed assertions within the group with a *MultipleFailuresError*



```
@Test
void groupAssertions() {
    int[] numbers = {0, 1, 2, 3, 4};
    assertAll("numbers",
        () -> assertEquals(numbers[0], 1),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 1)
    );
}
```

- Before JUnit 5, it will stop processing the rest of assertions if one of them failed — It creates a problem where we might have to run a test cases multiple times to fix the issues

MOCKITO

- There is a fundamental problem in writing unit test cases where there are external dependencies.
  - For example, if we want to write test cases for our service classes, we have to mock proper DAO injection(Thanks to DI)
  - We can’t reply on external system such as database calls — There might be no network connection during the build phase of the application
- In real-world applications, where components often depend on accessing external systems, it is important to provide proper test isolation so that we can focus on testing the functionality of a given unit without having to involve the whole class hierarchy for each test
  - Injecting a mock is a clean way to introduce such isolation.
    - Mockito provides us a easier way to mock the dependencies we need.

MOCKITO

- To get started, we have to include the dependency

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.21.0</version>
</dependency>
```

- Basic Annotations:
  - *@RunWith(MockitoJUnitRunner.class)* — annotate the JUnit test with a MockitoJUnitRunner
    - MockitoAnnotations.initMocks(this) — enable Mockito annotations programmatically
  - @Mock — create and inject mocked instances
  - *@Spy* — create and inject partially mocked instances (parameterized constructor)
  - *@InjectMocks* — inject mock fields into the tested object automatically (The class needs to be tested)
  - *verify* — provide more ways to validate the business logic

JACOCO

- Code coverage is a software metric used to measure how many lines of our code are executed during automated tests
- Jacoco is good tool to check the testing code coverage.
    - Maven Plugin for Jacoco
- Let's take a look at an example

Any questions?