Test lifecycle annotations

Getting started

We will use the following User class in our example:

```
public class User {
    private static final int MIN_PASSWORD_LENGTH = 8;
    private String username;
    private String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public boolean hasStrongPassword() {
        return password != null && password.length() >=
MIN_PASSWORD_LENGTH;
    }

    public boolean hasValidUsername() {
        return username != null && !username.isBlank();
    }

    public boolean isValid() {
        return hasValidUsername() && hasStrongPassword();
    }

    // getters and setters
}
```

It has two private String fields, username and password, a constructor that takes two String arguments, and three public methods to check if an instance of the class User has a valid username, a strong password, which is considered strong if it is at least 8 characters long, and represents a valid User, which means that it has a valid username and a strong password.

Also, we have the following test suite to test the correctness of the implementation of the User class methods:

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

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

class UserTest {

    @Test
    void hasStrongPassword() {
        String username = "Alice";
```

```
        String password = "12345678";
        User user = new User(username, password);

        assertTrue(user.hasStrongPassword());
    }

    @Test
    void hasValidUsername() {
        String username = "Alice";
        String password = "12345678";
        User user = new User(username, password);

        assertTrue(user.hasValidUsername());
    }

    @Test
    void isValid() {
        String username = "Alice";
        String password = "12345678";
        User user = new User(username, password);

        assertTrue(user.isValid());
    }
}
```

Look at the implementation of these tests. Each of them is completely independent of the others. We are not calling any test in another test and each test uses a new instance of the User class.

This may not make much sense in this particular case, but if we are testing more complex classes we will want to start each time with a clean state of the object being tested.

This means that we should not create a single instance of a class and share it among all tests, since a certain test may leave some state in the object that might affect the results of other tests. That is why we should execute each test using a new instance of the tested class.

We initialized a new object of the User class in each test method repeating the same code in multiple places, which generally is a bad idea.

Also, notice that we tested just the happy path of execution of the User class methods, but in a real project we will have to check every execution path of every method, and this will require us to write many tests.

Test class instance lifecycle

It creates a new instance of the test class before executing each test method. This way it ensures execution of individual test methods in isolation to avoid possible side effects produced by any changes of state of the test class instance.

@TestInstance annotation controls the test class instance lifecycle. It is set to TestInstance.Lifecycle.PER_METHOD by default but can be changed if necessary.

Also, JUnit5 has special annotations to designate any methods as lifecycle methods, such as @BeforeAll, @AfterAll, @BeforeEach or @AfterEach, to instruct the framework to execute the designated methods before or after executing actual test methods.

The annotations @BeforeEach and @AfterEach indicate that the annotated method will be executed before and, respectively, after each method of the test class annotated with @Test, while @BeforeAll and @AfterAll methods will be executed before or after all the @Test methods in the test class.

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

public class LifeCycleTest {

    LifeCycleTest() {
        System.out.println("Test Class Constructor");
    }

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

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

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

    @AfterEach
```

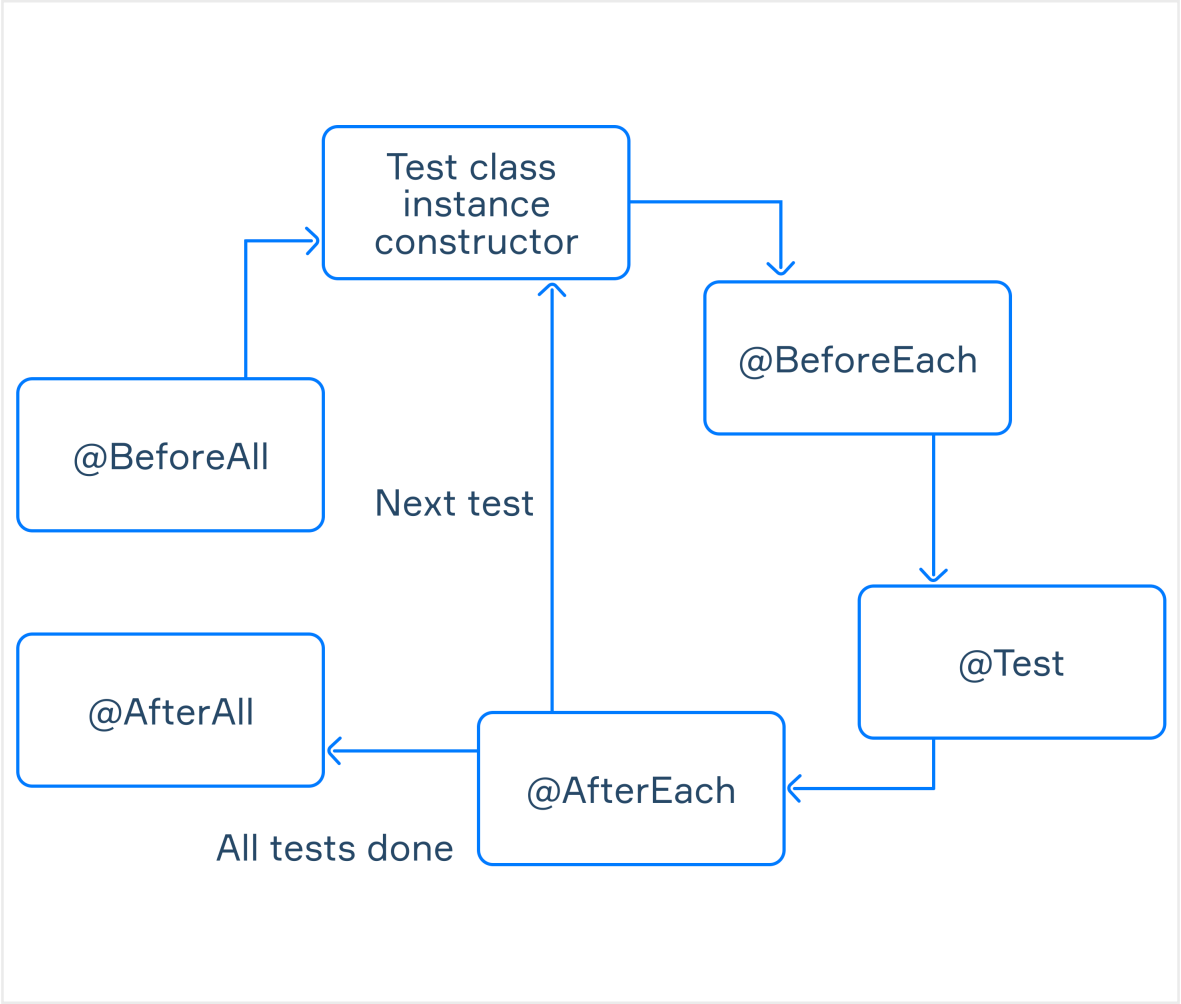
```
    void afterEach() {
        System.out.println("After each test");
    }

    @Test
    void test1() {
        System.out.println("Test 1");
    }

    @Test
    void test2() {
        System.out.println("Test 2");
    }
}
```

A test fixture is a fixed state of objects intended to provide a known and fixed environment for running tests.

Note that the methods annotated with @BeforeAll and @AfterAll are static, because this way they can be shared among new test class instances created for each test method.

Running the tests gives the following output:
Before the test fixture
Test Class Constructor
Before each test
Test 1
After each test
Test Class Constructor
Before each test
Test 2
After each test
After the test fixture

The following diagram illustrates this order to help you better understand the test class lifecycle:



Also, note that JUnit5 constructs a new instance of the test class before executing each test method.

Using lifecycle annotations

Now, we may rewrite our UserTest class and get rid of initialization of User instances in each test method:

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

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

class UserTest {

    User user;

    @BeforeEach
    void createUser() {
        String username = "Alice";
        String password = "12345678";
```

```
        user = new User(username, password);
    }

    @Test
    void hasStrongPassword() {
        assertTrue(user.hasStrongPassword());
    }

    @Test
    void hasValidUsername() {
        assertTrue(user.hasValidUsername());
    }

    @Test
    void isValid() {
        assertTrue(user.isValid());
    }
}
```

@BeforeEach , as you have seen, may be used to set up new instances of the classes being tested.

@AfterEach is handy to clean up any side effects of the execution of the tests or to provide detailed information about their execution and results.

@BeforeAll and @AfterAll are great for setting up and tearing down the entire test fixture.

You may use methods annotated by @BeforeAll to create and initialize big data structures, establish connections to data sources, fetch data from databases, remote repositories, or hard drives, and after that close resources and clean everything up in @AfterAll methods.

Putting it all together, we can write the following implementation of our test class with a pre-defined set of input data:

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

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

class UserTest {

    static String[] names;
    static String[] passwords;
    static boolean[] expectedOutcomes;
```

```
    static int index = 0;

    User user;
    boolean expected;

    @BeforeAll
    static void setUp() {
        names = new String[] {"Alice", "Alice", "Alice", "", null, "    "};
        passwords = new String[] {"12345678", "123", null, "12345678",
"12345678", "12345678"};
        expectedOutcomes = new boolean[] {true, false, false, false, false, false};
    }

    @BeforeEach
    void createUser() {
        user = new User(names[index], passwords[index]);
        expected = expectedOutcomes[index];
    }

    @AfterEach
    void incrementIndex() {
        index++;
    }

    @RepeatedTest(value = 6, name = "user.isValid() test {currentRepetition}/
{totalRepetitions}")
    void isValid() {
        assertEquals(expected, user.isValid());
    }
}
```

Here we used @RepeatedTest to run the annotated test 6 times (value = 6) and defined a custom name for displaying test
results. {currentRepetition} and {totalRepetitions} are placeholders for displaying the current run and the total number of test runs. Here is the output:

```
UserTest > user.isValid() test 1/6 PASSED
UserTest > user.isValid() test 2/6 PASSED
UserTest > user.isValid() test 3/6 PASSED
UserTest > user.isValid() test 4/6 PASSED
UserTest > user.isValid() test 5/6 PASSED
UserTest > user.isValid() test 6/6 PASSED
```

Even in this simple example lifecycle annotations help us create multiple test cases with ease.

Test instance per class

Test instance per class
If for any reason you would like to execute all test methods on the same instance of the test class, JUnit5 allows you to do so by annotating the test class with @TestInstance(Lifecycle.PER_CLASS).

In this mode, a new instance of the test class will be created only once, therefore if your test methods rely on a state stored in its non-static variables, you may need to reset that state in @BeforeEach or @AfterEach methods.

We will see how PER_CLASS works in another example. First, let's add this annotation to our LifeCycleTest class:

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

@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class LifeCycleTest {

    LifeCycleTest() {
        System.out.println("Test Class Constructor");
    }

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

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

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

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

    @Test
    void test1() {
        System.out.println("Test 1");
```

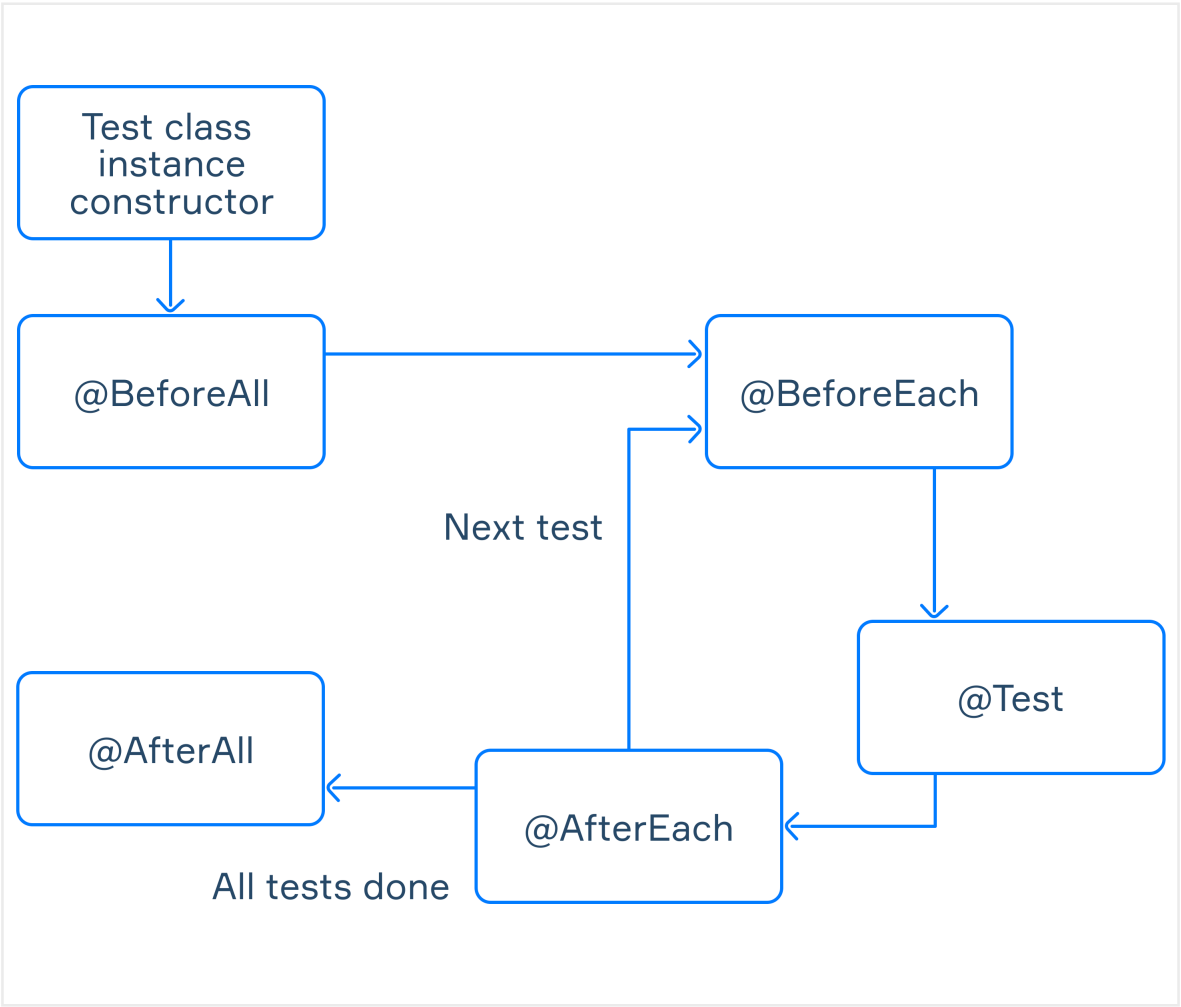
```
    }

    @Test
    void test2() {
        System.out.println("Test 2");
    }
}
```

Note that since the test class instance is shared among all test methods, there is no need for the @BeforeAll and @AfterAll methods to be static. Now let's run it and see what has changed compared to the new instance per test method:

```
Test Class Constructor
Before the test fixture
Before each test
Test 1
After each test
Before each test
Test 2
After each test
After the test fixture
```

For the per test method test instance lifecycle, remove the @TestInstance annotation from your test class or explicitly use @TestInstance(TestInstance.Lifecycle.PER_METHOD).

The following diagram illustrates the method call sequence:



If you are using this mode and your test methods rely on state stored in instance variables, you may need to reset that state in @BeforeEach or @AfterEach methods to avoid unexpected side effects.