Parameterized tests

Introduction

There is often a need to run one unit test several times with different inputs. Such a problem can be solved by duplicating an existing test or extracting a separate method that we call in separate tests. However, such a solution causes the number of unit tests to grow. This problem is solved by parameterized tests, which implement the test code only **once**. The code can be executed multiple times with different sets of input data.

Tests parameterized in JUnit 5

JUnit version 5 provides a set of tools for implementing parameterized tests. For this purpose, the following dependency should be introduced into the project:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.6.2</version> <!-- the current version may change -->
    <scope>test</scope>
<!-- scope test means that libraries will be visible only in the test bundle -->
</dependency>
```

@ParameterizedTest

Parameterized tests are implemented almost identically to regular unit tests, except for the way they are marked. **Instead of** the `@Test` annotation, we use `@ParameterizedTest`.

The parameterized test may look like this:

```
@ParameterizedTest
@ValueSource(ints = {1, 3, 5, -3, 15})
void shouldReturnTrueForOddNumbers(int number) {
    assertTrue(number % 2 != 0);
}
```

The `@ValueSource` defines the set of inputs that will be passed to the test method as its arguments.

It is the programmer's responsibility to match the number and types of arguments to the source of the arguments. The above example will trigger the unit test five times.

@CsvSource

The `@CsvSource` annotation allows you to define test parameters using CSV literals (*comma separated value*). Also:

- Data strings are separated by certain characters (*commas* by default).
- Each separated element is a *separate* parameter taken in the test.
- This mechanism may be used when we want to provide input parameters and the expected value for the purposes of a given unit test.
- The limit of the `@CsvSource` annotation is a limited number of types that can be used in the test. All parameter types we want to use in the test must be convertible from the `String` object.

```
@ParameterizedTest
@CsvSource({"  test  ,TEST", "tEst ,TEST", "   Java,JAVA"})
void shouldTrimAndUppercaseInput(String input, String expected) {
   String actualValue = input.toUpperCase();
   assertEquals(expected, actualValue);
}
```

@CsvFileSource

The `@CsvFileSource` annotation is very similar to the `@CsvSource`, except the data is loaded directly from the file. It can define the following parameters:

- `numLinesToSkip` - specifies the number of ignored lines in the source file. This is useful if the data comes from a table that has headers in addition to the value.
- `delimeter` - means a separator between each *element*.
- `lineSeparator` - means the separator between each *sets* of parameters.
- `encoding` - means the file content encoding method.

```
@ParameterizedTest
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1)
    // the data.csv file must be in the classpath root, we skip the first line in the file
void shouldUppercaseAndBeEqualToExpected(String input, String expected) {
    String actualValue = input.toUpperCase();
    assertEquals(expected, actualValue);
}
```

@MethodSource

The @ValueSource,@EnumSource, @CsvSource and @CsvFileSource annotations have limitations - the number of parameters or their types. This problem is solved by the argument source defined by the @MethodSource annotation. The only parameter for this annotation is the * name * of a **static** method in the same class. This method should be argumentless and return stream:

- objects of any type in the case of tests with a single parameter,
- Arguments objects for tests with multiple parameters.

```
@ParameterizedTest
@MethodSource("provideNumbers")
void shouldBeOdd(final Integer number) {
   assertNotEquals(0, number % 2);
}

private static Stream<Integer> provideNumbers() {
   return Stream.of(1, 13, 101, 11, 121);
}
```

```
@ParameterizedTest
@MethodSource("provideNumbersWithInfoAboutParity")
void shouldReturnExpectedValue(int number, boolean expected) {
   assertEquals(expected, number % 2 == 1);
}

private static Stream<Arguments> provideNumbersWithInfoAboutParity() {
   return Stream.of(Arguments.of(1, true),
         Arguments.of(2, false),
         Arguments.of(10, false),
         Arguments.of(11, true));
}
```

Mockito

Introduction

Mockito is a library that provides an API for creating the so-called **mocks**. Mocking is nothing else than creating a *dummy* which is an implementation of a given object. Such a dummy can be modeled **freely**. This way specific behaviors of objects are programmed without going into the implementation details. An example use of mocking can be observed when a user service class uses a database. If you want to test the service class, you should also provide a database configuration, although it should not be tested. In this case, it is much more effective to prepare the so-called *mock* which will only model the behavior of the database, without *actually* implementing its functionality.

Integration

In order to start working with mockito, the following dependencies should be included in an existing project:

```
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-core</artifactId>
   <version>3.3.3</version>
   <scope>test</scope>
</dependency>
```

```
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-junit-jupiter</artifactId>
   <version>3.3.3</version>
   <scope>test</scope>
</dependency>
```

Ways of integration

We can use Mockito in two ways. In order to create *mocks* and inject them, we can choose:

- the declarative way, i.e. using annotations
- the programmatic way, i.e. using appropriate methods.

When we choose the declarative way, JUnit "understands" and can handle additional annotations. You need to add the extension @ExtendWith (MockitoExtension.class), responsible for integrating the Mockito API within the implemented unit tests.

```
@ExtendWith(MockitoExtension.class)
class UserServiceTest {
   //...
}
```

```java
public class MockUserService {
    private final UserRepository userRepository;

    public MockUserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public MedicalUser findById(Long id){
        return userRepository.findById(id).orElse(null);
    }
    public MedicalUser findByUsername(String username){
        return userRepository.findByUsername(username).orElseThrow(()-> GenericException.notFound(username));
    }
}
```

```java
@ExtendWith(MockitoExtension.class)
public class MockUserServiceTest {
    @Mock
    private UserRepository userRepository;
    @InjectMocks
    private MockUserService mockUserService;
```

```java
@Test
void shouldFindById(){
    Mockito.when(userRepository.findById(1L)).thenReturn(Optional.of(new MedicalUser()));
    assertNotNull(mockUserService.findById(1L));
}
```

Congratulations on finishing this course!