

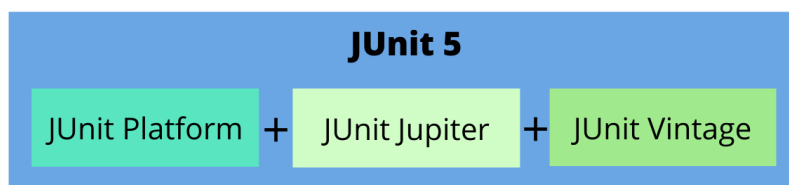
Notatki - Testowanie - JUnit

Spis treści

JUnit.....	1
Dodanie zależności	2
Konfiguracja dla Maven	2
Konfiguracja dla Gradle	3
Stworzenie klasy testowej.....	3
Metoda testowa.....	4
Maven	6
Gradle	7
Konwencja given, when, then	7
@DisplayName	8
Metody before i after	9
Asercje	11
Jakie mamy dostępne podstawowe asercje?	12
AssertAll	13
Sprawdzanie wyjątków	14
AssertJ	14
Wyłączenie testów	15
Testy parametrized.....	15
Arguments	18
Błędy	19
Logowanie.....	19

JUnit

Bardzo popularnym (są inne, ale o dziwo, zawsze w praktyce widziałem tylko ten) frameworkiem, który służy do tworzenia testów jednostkowych jest JUnit. Do przykładów posłuży nam JUnit w wersji 5. JUnit 5 składa się z 3 modułów, które wymieniam poniżej, a [tutaj](#) dokumentacja.



Obraz 1. JUnit 5

- **JUnit Platform** - jest platformą służącą do uruchamiania różnych frameworków testowych na maszynie wirtualnej Javy. Oznacza to, że przy wykorzystaniu platformy JUnit Platform możemy również uruchamiać inne biblioteki testujące takie jak Spock, Cucumber lub FitNesse, ale o nich nie będziemy rozmawiać. JUnit Platform zapewnia również interface pomiędzy JUnit a np. narzędziami

do budowania.

- **JUnit Jupiter** - moduł zawierający nowy model (np. nowe adnotacje), który został wprowadzony w JUnit w wersji 5.
- **JUnit Vintage** - moduł umożliwiający uruchomienie testów napisanych w JUnit 3 lub 4.

Dodanie zależności

Jeżeli chodzi o paczki, to będziemy mieli do czynienia z 2 nazwami:

- **junit-jupiter-api** - zależność, która umożliwia dodanie do projektu adnotacji oznaczających, które metody w kodzie mają zostać wykorzystane jako metody testowe.
- **junit-jupiter-engine** - zależność służąca do uruchomienia testów.

Zależność **junit-jupiter-engine** jest zależna pod spodem od **junit-jupiter-api**, więc wystarczy dodać tylko tę pierwszą.

Konfiguracja dla Maven

Aby wykorzystać zależność JUnit w projekcie przy użyciu Maven, należy dodać następującą konfigurację w pliku **pom.xml**:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Pamiętasz **scope** w Maven? Teraz dopiero przyda nam się **scope: test**. Oznaczało to, że dependencja nie jest wymagana podczas normalnego działania programu, potrzebujemy jej natomiast jeżeli będziemy pisać testy do naszego programu. Najczęściej stosujemy to do bibliotek, które są używane tylko do testów. Czyli jeżeli dodamy dependencję jako **test**, to będzie ona używana w testach, ale nie będziemy mieli dostępu do tej zależności z poziomu normalnego kodu.

Ponadto, w przypadku Maven, jeżeli chcemy, aby testy uruchamiały się podczas buildu aplikacji, musimy dodać **Maven Surefire Plugin**. Traktuje on jako testy wszystkie klasy, które są umieszczone w ścieżce **src/test/java** oraz spełniają warunki:

- Ich nazwa zaczyna się lub kończy Stringiem "Test",
- Ich nazwa kończy się Stringiem "Tests",
- Ich nazwa kończy się Stringiem "TestCase".

Z kolei pod ścieżką **src/test/resources** możemy umieścić pliki z konfiguracją, która ma być używana w testach.

Konfiguracja pluginu Maven Surefire Plugin

```
<build>
```

```

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
  </plugin>
</plugins>
</build>

```

Konfiguracja dla Gradle

Aby wykorzystać zależność JUnit w projekcie przy użyciu Gradle, należy dodać następującą konfigurację w pliku **build.gradle**:

```

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.2'
}
test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}

```

Ponownie, oznaczenie **test** oznacza, że zależność będzie używana tylko w testach, nie będziemy mieli do niej dostępu w normalnym kodzie.

Pomimo, że Gradle w wersji wyższej niż **4.6** ma wbudowaną obsługę JUnit5, nie jest ona domyślnie włączona. Aby to zrobić, należy dodać dopisek:

```

test {
    useJUnitPlatform()
}

```

Natomiast fragment poniżej odpowiada za logowanie przebiegu wykonania testów. Bez niego Gradle nie loguje informacji, które testy zostały wykonane.

```

test {
    testLogging {
        events "passed", "skipped", "failed"
    }
}

```

Stworzenie klasy testowej

Klasy z testami powinny być umieszczone w ścieżce **src/test/java**, zatem możemy tam stworzyć taką ścieżkę ręcznie, lub wykorzystać do tego IntelliJ.

Stwórzmy najpierw taką klasę:

```
package pl.zajavka;

public class Calculator {

    public static Integer add(int left, int right) {
        return left + right;
    }
}
```

Jeżeli teraz użyjemy skrótu **CTRL + SHIFT + T**, IntelliJ podpowie nam, że możemy utworzyć test sprawdzający tę klasę. Jeżeli taki test już istnieje, zostanie nam to pokazane w oknie dialogowym (jeżeli klas testowych jest więcej niż jedna) lub zostaniemy do tego testu bezpośrednio przekierowani.



Dobłą praktyką jest, aby klasy testowe były umieszczone w tej samej paczce, co kod źródłowy. Czyli dla klasy `pl.zajavka.Calculator`, test powinien być umieszczony w paczce `pl.zajavka`, czyli pełna ścieżka to `pl.zajavka.CalculatorTest`.

Metoda testowa

Jako test rozumiemy metodę w klasie testowej, która jest oznaczona adnotacją `@Test`. Klasa taka jest używana tylko do testów, nie odnosimy się do niej w kodzie źródłowym (czyli tym w `src/main/java`). Jeżeli oznaczmy metodę adnotacją `@Test`, możemy ją wywołać tak samo jak wywoływaliśmy metodę `main()`. Co więcej, w jednej klasie testowej możemy mieć wiele metod testowych, zatem możemy uruchomić jednocześnie całą klasę testową, ze wszystkimi metodami w niej zawartymi.



Pamiętasz, jak kiedyś wspomniałem, że oprócz metody `main()`, możemy uruchomić aplikację jeszcze w inny sposób? Miałem na myśli to, że wcale nie musimy mieć w kodzie metody `main()`. Możemy też program uruchomić z poziomu testów. Należy jednak pamiętać o tym, że to zastosowanie ma sens tylko gdy piszemy mały program, który zostanie uruchomiony, coś policzy i zakończy swoje działanie. Wtedy możemy określić kilka sposobów uruchomienia takiego programu poprzez różne metody testowe z różnymi parametrami i w ten sposób go uruchamiać. W praktyce natomiast, gdy będziemy pisali aplikacje uruchamiane na serwerze (o czym jeszcze będzie), ten sposób nie zda egzaminu.

Poniżej przykład klasy testowej, z metodą testową `testAdd()`, która testuje metodę `Calculator.add()`:

```
package pl.zajavka;

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

class CalculatorTest {

    @Test
    void testAdd() {
        // given
        int left = 5;
        int right = 7;
        Integer expected = 12;
    }
}
```

```

    // when
    Integer result = Calculator.add(left, right);

    // then
    Assertions.assertEquals(expected, result);
}
}

```

Zwróć uwagę, że metoda testowa `testAdd()`, nic nie zwraca. Klasa `CalculatorTest` wcale nie musi być publiczna, tak samo jak metoda `testAdd()`. Jeżeli w tym momencie uruchomimy test z IntelliJ, zostanie wykonana metoda `Calculator.add()` z parametrami `int left = 5;` oraz `int right = 7;`, a na koniec nastąpi sprawdzenie, czy wynik metody `Calculator.add()` jest równy `Integer expected = 12;`.

Dodajmy teraz jeszcze 3 metody do klasy `Calculator`. Metodę odejmującą, mnożącą i dzielącą:

```

package pl.zajavka;

public class Calculator {

    public static Integer add(int left, int right) {
        return left + right;
    }

    public static Integer subtract(int left, int right) {
        return left - right;
    }

    public static Integer multiply(int left, int right) {
        return left * right;
    }

    public static Integer divide(int left, int right) {
        return left / right;
    }
}

```

Do tego, dodajmy metody testujące każdą z dopisanych metod:

```

package pl.zajavka;

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

class CalculatorTest {

    // test metody testAdd()

    @Test
    void testSubtract() {
        // given
        int left = 5;
        int right = 7;
        Integer expected = -2;

        // when
        Integer result = Calculator.subtract(left, right);
    }
}

```

```

        // then
        Assertions.assertEquals(expected, result);
    }

    @Test
    void testMultiply() {
        // given
        int left = 5;
        int right = 7;
        Integer expected = 35;

        // when
        Integer result = Calculator.multiply(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }

    @Test
    void testDivide() {
        // given
        int left = 15;
        int right = 7;
        Integer expected = 2;

        // when
        Integer result = Calculator.divide(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }
}

```

Możemy teraz uruchomić wszystkie metody testowe z poziomu klasy `CalculatorTest`.

Maven

Testy te będą teraz również uwzględnione przy uruchamianiu budowania przez Maven i Gradle. Możemy zatem uruchomić poniższą komendę (pamiętając, że musimy wcześniej skonfigurować [Maven Surefire plugin](#)):

```
mvn verify
```

I na ekranie zobaczymy taki wydruk:

```

[INFO]
[INFO] -----
[INFO]  T E S T S
[INFO] -----
[INFO] Running pl.zajavka.CalculatorTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.017 s - in
pl.zajavka.CalculatorTest
[INFO]
[INFO] Results:

```

```
[INFO]  
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

Możemy również przekazać Mavenowi, że chcemy wywołać build, ale ma nie uruchamiać testów, w tym celu wykonamy komendę:

```
mvn verify -DskipTests
```

Gradle

Korzystając z Gradle, wykonamy komendę:

```
gradle build
```

I na ekranie zobaczymy taki wydruk:

```
> Task :test  
  
CalculatorTest > testAdd() PASSED  
  
CalculatorTest > testSubtract() PASSED  
  
CalculatorTest > testDivide() PASSED  
  
CalculatorTest > testMultiply() PASSED  
  
BUILD SUCCESSFUL in 1s  
4 actionable tasks: 2 executed, 2 up-to-date
```

Możemy również przekazać Gradlowi, że chcemy wywołać build, ale ma nie uruchamiać testów, w tym celu wykonamy poniższą komendę. Skrót **-x** oznacza exclude task.

```
gradle build -x test
```

Konwencja given, when, then

O co chodzi z tym dodaniem komentarzy **given, when, then**? Po prostu - konwencja.



Obraz 2. Konwencja given, when, then

Jedną ze stosowanych konwencji przy pisaniu testów jest nazywanie klas testowych tak aby kończyły się na `*Test.java`. Jednocześnie wspomnieliśmy już, że narzędzia takie jak np. Maven rozpoznają w ten sposób, gdzie mogą znaleźć testy.

Kolejną konwencją przy pisaniu testów jest dokładne określenie co dana metoda testowa sprawdza. Możemy to określić nazywając metodę w odpowiedni sposób np. `shouldCalculateAdditionCorrectly()`. W praktyce jednak nie jest to tak często stosowane i każdy nazywa testy tak aby dało się zrozumieć co one sprawdzają. JUnit5 daje również możliwość dodania adnotacji `@DisplayName`, która może być umieszczona nad metodą testową i która pozwala opisać co dany test robi.

Często stosowaną konwencją jest `given, when, then`. Dzięki stosowaniu tej konwencji możemy podzielić naszą metodę testową na 3 logiczne fragmenty:

- **given** - w którym definiujemy co jest parametrem wejściowym naszego testu,
- **when** - w którym określamy jaka logika jest faktycznie testowana,
- **then** - w który definiujemy co jest spodziewanym wynikiem.

Stąd komentarze w kodzie i stąd nazwa `given, when, then`.

Wrócimy jeszcze do tej konwencji na etapie omawiania **Mockito** w przyszłych materiałach.

@DisplayName

Adnotacja `@DisplayName` może być zastosowana jak w przykładzie poniżej:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class CalculatorTest {
```



```

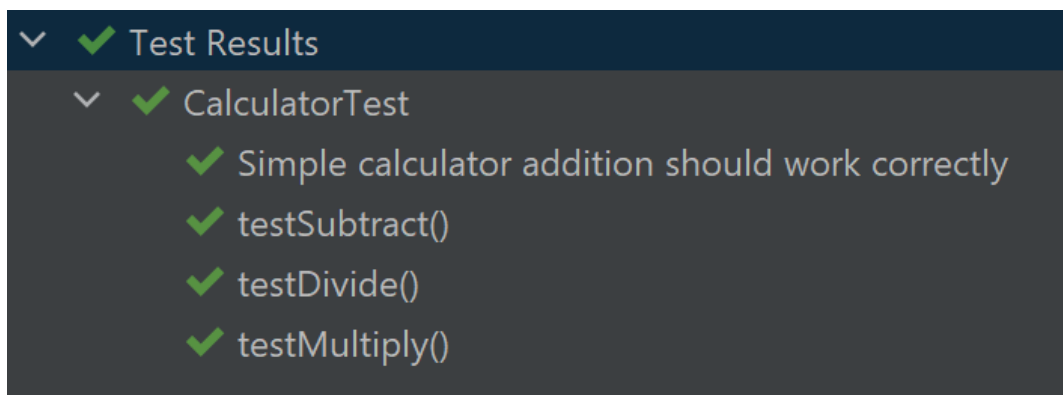
@Test
@DisplayName("Simple calculator addition should work correctly")
void testAdd() {
    // given
    int left = 5;
    int right = 7;
    Integer expected = 12;

    // when
    Integer result = Calculator.add(left, right);

    // then
    Assertions.assertEquals(expected, result);
}
}

```

Jeżeli teraz uruchomimy ten test, to na ekranie zobaczymy taki zapis:



Obraz 3. Rezultat zastosowania adnotacji @DisplayName

Widać, że jest to czytelniejsze niż określanie co dany test sprawdza poprzez nazywanie w ten sposób metody. Jeżeli nazywamy w metodzie co dany test robi to dostaniemy szlaczek pokroju `shouldCalculateAdditionCorrectly()`, które w praktyce potrafią być jeszcze dłuższe. Stosując `@DisplayName` łatwiej jest odczytać intencję autora.

Metody before i after

JUnit daje nam możliwość uruchomienia kodu jednokrotnie przed wszystkimi metodami testowymi w klasie lub za każdym razem przed każdą z nich. Jednocześnie możemy zrobić to samo po każdej metodzie testowej lub jednokrotnie po wszystkich.

Zdefiniujmy klasę testową w ten sposób:

```

package pl.zajavka;

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.DisplayName;
import org.junit.jupiter.api.Test;

class CalculatorTest {

```

```

@BeforeEach
void methodSetup() {
    System.out.println("Calling @BeforeEach");
}

@AfterEach
void methodTearDown() {
    System.out.println("Calling @AfterEach");
}

@BeforeAll
static void classSetup() {
    System.out.println("Calling @BeforeAll");
}

@AfterAll
static void classTearDown() {
    System.out.println("Calling @AfterAll");
}

@Test
@DisplayName("Simple calculator addition should work correctly")
void testAdd() {
    System.out.println("Calling testAdd");
}

@Test
void testSubtract() {
    System.out.println("Calling testSubtract");
}

@Test
void testMultiply() {
    System.out.println("Calling testMultiply");
}

@Test
void testDivide() {
    System.out.println("Calling testDivide");
}
}

```

Jeżeli teraz uruchomimy wszystkie testy z poziomu klasy, na ekranie wydrukuje się przebieg wykonania testów z całej klasy:

```

Calling @BeforeAll
Calling @BeforeEach
Calling testAdd
Calling @AfterEach
Calling @BeforeEach
Calling testSubtract
Calling @AfterEach
Calling @BeforeEach
Calling testDivide
Calling @AfterEach
Calling @BeforeEach
Calling testMultiply
Calling @AfterEach

```

Zwróć uwagę, że zachowanie jest takie jak napisałem. `@BeforeAll` i `@AfterAll` uruchamia się tylko na początku i na końcu klasy testowej. Natomiast `@BeforeEach` i `@AfterEach` uruchamiają się za każdym razem przed i po uruchomieniu metody testowej.



Zwróć uwagę, że metody oznaczone adnotacjami `@BeforeAll` i `@AfterAll` są statyczne!

Po co to? W naszym przykładzie testujemy metody statyczne (metody w klasie `Calculator` są statyczne). Ale jeżeli musielibyśmy testować metody na obiekcie, który trzeba utworzyć, to po co za każdym razem tworzyć obiekt w każdej metodzie testowej na nowo, jeżeli można zapisać to tak:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void methodSetup() {
        calculator = new Calculator();
    }

    @Test
    @DisplayName("Simple calculator addition should work correctly")
    void testAdd() {
        calculator.calculate();
        // reszta testu
    }
}
```

W ten sposób, możemy przy każdym teście operować na nowym obiekcie, ale nie musimy za każdym razem zapisywać jego utworzenia na nowo. `@BeforeAll` i `@AfterAll` mogą być natomiast użyte jeżeli chcemy w ramach całego testu zainicjować np. połączenie do bazy danych. Przecież nie będziemy go inicjować za każdym razem, przed uruchomieniem metody testowej.

Asercje

Teraz możemy przejść do zapisu na końcu całego testu, czyli `Assertions.assertEquals(expected, result);`. Asercje są sposobem na określenie, co jest spodziewanym wynikiem naszego testu. Asercja sprawdza wtedy, czy wynik działania testu zgadza się ze stanem oczekiwanym. W poprzednim przypadku została użyta metoda `assertEquals()`, która przyjmuje parametry: spodziewany i faktyczny wynik. Następnie sprawdza czy faktycznie jest tak jak oczekiwaliśmy. Jeżeli wartość oczekiwana i faktyczny rezultat są zgodne, test zostaje zaliczony (`passed`). Jeżeli nie, asercja przerywa wykonanie testu i oznacza go jako nieudany (`failed`).

Wspomniałem, że asercja `failed` przerywa wykonanie testu. Możemy zdefiniować kilka asercji pod sobą,

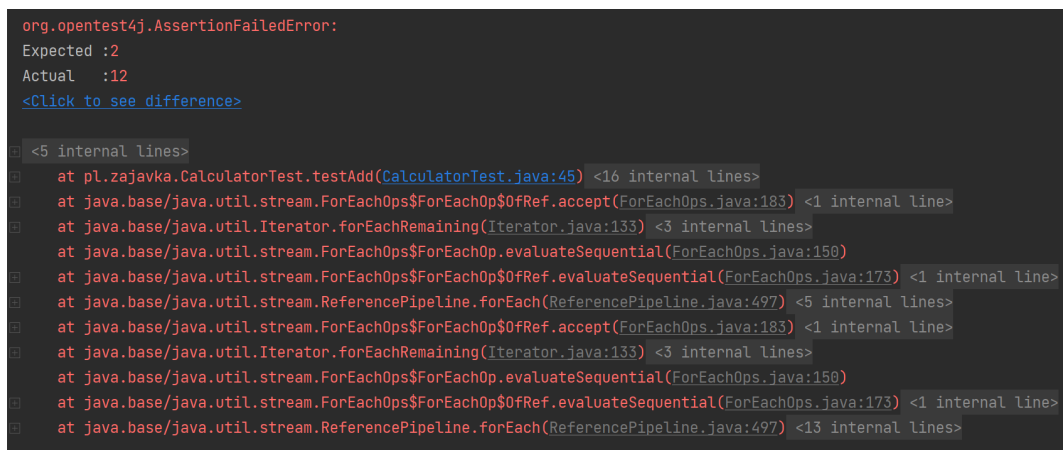
w zależności od tego jakie są nasze potrzeby. Jeżeli tylko pierwsza z nich failuje (w sumie to na co dzień tak się to określa), pozostałe nawet nie są sprawdzane. Przykład failującego testu:

```
@Test
@DisplayName("Simple calculator addition should work correctly")
void testAdd() {
    // given
    int left = 5;
    int right = 7;
    Integer expected = 2;

    // when
    Integer result = Calculator.add(left, right);

    // then
    Assertions.assertEquals(expected, result);
}
```

Wynik $5 + 7 \neq 2$, zatem test failuje. Na ekranie zobaczymy wtedy coś takiego:



Obraz 4. Wynik testu, który zakończył się niepowodzeniem

Jakie mamy dostępne podstawowe asercje?

Przykład podstawowych asercji umieszczam w tabeli poniżej:

Tabela 1. Podstawowe asercje

Metoda	Przykład
<code>assertEquals</code>	<code>assertEquals(2, 12, "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertTrue</code>	<code>assertTrue(5 != 7, 0 → "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertFalse</code>	<code>assertFalse(2 == 4, "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertNotNull</code>	<code>assertNotNull(jakaśReferencja, "Opcjonalna wiadomość przy niepowodzeniu");</code>
<code>assertNull</code>	<code>assertNull(jakaśReferencja, "Opcjonalna wiadomość przy niepowodzeniu");</code>

Zauważ, że w jednym przykładzie, wiadomość została przekazana przez **lambdę**. Napiszmy taki przykład:

```
class CalculatorTest {

    @Test
    void someTest() {
        Assertions.assertEquals(1, 1, createMessage(1));
        Assertions.assertEquals(1, 1, () -> createMessage(2));
        Assertions.assertEquals(1, 2, () -> createMessage(3));
    }

    private String createMessage(int i) {
        System.out.println("Evaluating failure message: " + i);
        return "failure message";
    }
}
```

Po wywołaniu metody testowej `someTest()`, na ekranie zostanie wydrukowane:

```
Evaluating failure message: 1
Evaluating failure message: 3

org.opentest4j.AssertionFailedError: failure message ==>
Expected :1
Actual   :2
```

Zauważ, że jeżeli przekazaliśmy wiadomość bezpośrednio (oczywiście w formie metody, aby widzieć, poprzez `System.out.println()`, że metoda została wykonana), to nawet jeżeli asercja nie failuje, to i tak na ekranie wydrukowała się wiadomość `Evaluating failure message: 1`.

W przypadku wiadomości `Evaluating failure message: 2`, została ona przekazana w formie lambdy, zatem w przypadku gdy asercja przechodzi poprawnie, wiadomość ta nie jest drukowana na ekranie. Zostanie ona wydrukowana dopiero gdy asercja failuje, dlatego na ekranie widzimy wiadomość `Evaluating failure message: 3`.

Powyższy przykład pokazuje, że jeżeli chcemy przekazywać wiadomość jako parametr asercji, należy to robić przy wykorzystaniu **Supplier**, wtedy implementujemy lambdę jako argument wywołania. Oszczędzamy dzięki temu zasoby, jeżeli określenie wiadomości, która jest przekazana byłoby kosztowne.

AssertAll

Wspomnieliśmy, że jeżeli mamy kilka asercji po sobie i którakolwiek z nich zakończy się niepowodzeniem, to pozostałe nie zostaną sprawdzone. Jeżeli natomiast chcemy w takim przypadku wymusić sprawdzenie wszystkich, należy zastosować metodę `assertAll()`. Porównaj ze sobą 2 poniższe wywołania:

Wywołanie bez `assertAll()`

```
Assertions.assertEquals("test1", "test1", createMessage(1));
Assertions.assertEquals("test1", "test2", () -> createMessage(2));
Assertions.assertEquals("test1", "test4", () -> createMessage(3));
```

I jego wynik na ekranie:

```
org.opentest4j.AssertionFailedError: failure message ==>
Expected :test1
Actual   :test2
```

Wywołanie z `assertAll()`

```
Assertions.assertAll(
    () -> Assertions.assertEquals("test1", "test1", createMessage(1)),
    () -> Assertions.assertEquals("test1", "test2", () -> createMessage(2)),
    () -> Assertions.assertEquals("test1", "test4", () -> createMessage(3))
);
```

I jego wynik na ekranie:

```
failure message ==> expected: <test1> but was: <test2>
Comparison Failure:
Expected :test1
Actual   :test2

failure message ==> expected: <test1> but was: <test4>
Comparison Failure:
Expected :test1
Actual   :test4
```

Widać, że jeżeli stosujemy `assertAll()`, to wywołanie nie zostaje przerwane i wykonywane są wszystkie asercje.

Sprawdzanie wyjątków

Możemy oczekiwać, że kod, który wywołujemy w teście zakończy się wyrzuceniem wyjątku. Jak zapisać taką asercję?

```
Throwable exception =
    Assertions.assertThrows(NumberFormatException.class, () -> Integer.parseInt("MyInput"));
Assertions.assertEquals("For input string: \"MyInput\"", exception.getMessage());
```

W pierwszej linijce określamy, że oczekujemy, że fragment kodu `Integer.parseInt("MyInput")` wyrzuci wyjątek `NumberFormatException`. W drugiej linijce natomiast określamy, że wiadomość jaka ma być zawarta w tym wyjątku to: `For input string: "MyInput"`.

AssertJ

Jeżeli powyżej pokazane asercje uznamy za niewystarczające, możliwe jest korzystanie z bibliotek, które dają nam więcej możliwości. Przykładem, w którym takie asercje okażą się niewystarczające będzie sytuacja, gdy chcemy porównać ze sobą 2 obiekty, które składają się z 15 pól, ale chcemy pominąć przy tym porównaniu 2 wybrane. Wtedy sięga się np. po **AssertJ**, którego dokumentację zamieszczam [tutaj](#). Nie chcę natomiast w tym momencie poruszać kwestii wykorzystania tych asercji. Dodam tylko, że są one intuicyjne i w miarę analogiczne do tych domyślnie dostępnych w JUnit.

Wyłączenie testów

Poszczególne testy można wyłączyć przy wykorzystaniu adnotacji. W poprzednich wersjach Junit, było to **@Ignore**, która mogła zostać umieszczona nad klasą z testami lub nad poszczególnymi metodami. W JUnit5 adnotacja ta została zastąpiona adnotacją **@Disabled**. Adnotacja ta daje również możliwość opisanie jako parametr, dlaczego dany test jest wyłączony.

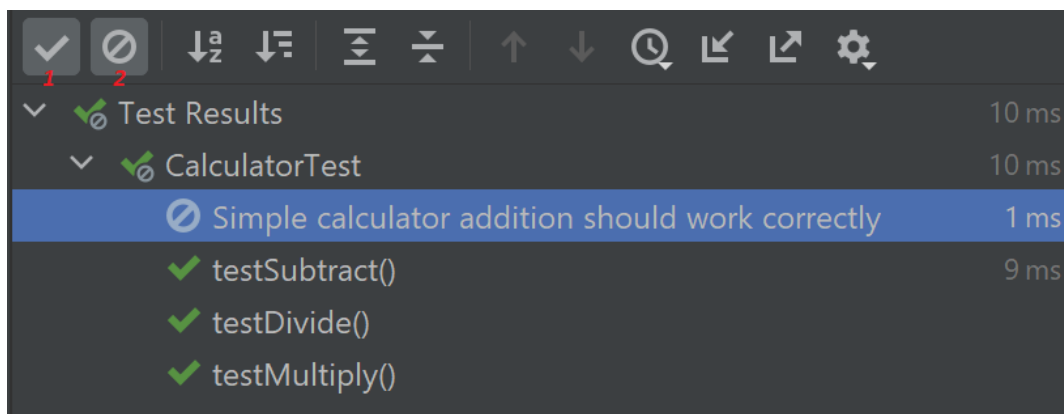


Czy powinno się oznaczać testy jako **@Disabled**? Moim osobistym zdaniem **NIE**. Jeżeli test failuje to jest to oznaka, że albo mamy błąd w kodzie, albo mamy błąd w testach. Niezależnie od przyczyny błędu trzeba poprawić jedno albo drugie, a nie to ignorować 😊.

Przykład oznaczenia metody testowej jako **@Disabled**:

```
@Test
@DisplayName("Simple calculator addition should work correctly")
@Disabled("Example why this test is disabled")
void testAdd() {
    // ...
}
```

Jeżeli teraz uruchomimy całą klasę testową, na ekranie będzie to wyglądało w ten sposób:



Obraz 5. Test oznaczony jako **@Disabled**

Co natomiast oznaczają ikonki określone jako 1 i 2?

- 1 - włącz/wyłącz pokazywanie testów, które zostały zaliczone
- 2 - włącz/wyłącz pokazywanie testów, które są ignorowane

Testy parametrized

Wyobraźmy sobie teraz sytuację, w której mamy kilka testów, które testują ten sam fragment kodu, ale chcemy wywołać je z innymi parametrami i spodziewamy się wtedy innego wyniku. ~~W takiej sytuacji należałoby napisać dla każdego wariantu oddzielną metodę testową.~~ Oczywiście, że nie! Po to są testy parametrized (parametryzowane).

W tym celu musimy dodać zależność do naszego projektu:

Maven

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-params</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Gradle

```
dependencies {
    // .. pozostałe dependencje
    testImplementation 'org.junit.jupiter:junit-jupiter-params:5.8.2'
}
```

Przykład kodu, który obrazuje w jaki sposób można napisać test **parametrized**:

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class CalculatorTest {

    @ParameterizedTest
    @MethodSource(value = "testData")
    void testParametrizedAdding(int[] testData) {
        // given
        int left = testData[0];
        int right = testData[1];
        Integer expected = testData[2];

        // when
        Integer result = Calculator.add(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }

    @SuppressWarnings("unused")
    public static int[][] testData() {
        return new int[][]{{1, 1, 2}, {2, 3, 5}, {9, 8, 17}, {2, 19, 21}};
    }
}
```

Co zostało zmienione? Zamiast adnotacji **@Test** mamy teraz **@ParameterizedTest**. Dodana została adnotacja **@MethodSource(value = "testData")**, która określa nam nazwę metody, która zwraca źródło danych do testu. W tym źródle danych mamy nie tylko parametry wejściowe, ale też spodziewany wynik. Metoda **testData()** zwraca tablicę dwuwymiarową, w której wymiar "zewnętrzny" określa kolejne grupy danych testowych, czyli możemy rozumieć to w ten sposób:

```
public static int[][] testData() {
```



```

return new int[][]{
    {1, 1, 2}, // pierwsza grupa parametrów
    {2, 3, 5}, // druga grupa parametrów
    {9, 8, 17}, // trzecia grupa parametrów
    {2, 19, 21} // czwarta grupa parametrów
};
}

```

Na tej podstawie widać, że test uruchomi się 4 razy, z 4 różnymi grupami parametrów.

W metodzie testowej `testParametrizedAdding(int[] testData)` umieszczona już jest tablica jednowymiarowa. Wynika to z tego, że każda z grup wspomnianych wcześniej reprezentuje już tablicę jednowymiarową i to z nią dany test jest wykonywany. Czyli mamy czterokrotne wykonanie testu z parametrami, które są przekazane w postaci tablicy jednowymiarowej.

W teście następnie określamy, że indeksy 0 i 1 tablicy są naszymi danymi wejściowymi, natomiast indeks 2 oznacza wartość oczekiwaną.

Dodałem też adnotację `@SuppressWarnings("unused")` nad metodą `testData()`, gdyż IntelliJ może pokazywać warning, że metoda ta nie jest używana. W ten sposób oznaczam, że jest i wiem o tym - jest to moje świadome działanie.

Zapis `@MethodSource(value = "testData")` może zostać uproszczony. Zamiast wpisywać ręcznie nazwę metody, która zawiera dane testowe możemy nazwać metodę z danymi testowymi identycznie jak nazywa się nasza metoda testowa. W taki przypadku testy zostaną uruchomione z prawidłowymi danymi testowymi. Przykład:

```

package pl.zajavka;

// importy

class CalculatorTest {

    @ParameterizedTest
    @MethodSource
    void testParametrizedAdding(int[] testData) {
        // ciało testu
    }

    @SuppressWarnings("unused")
    public static int[][] testParametrizedAdding() {
        return new int[][]{{1, 1, 2}, {2, 3, 5}, {9, 8, 17}, {2, 19, 21}};
    }
}

```

Jeżeli teraz uruchomimy test `testParametrizedAdding()`, to na ekranie zobaczymy jego czterokrotne wywołanie z różnymi parametrami.

✓	Test Results	61 ms
✓	CalculatorTest	61 ms
✓	testParametrizedAdding(int[])	61 ms
✓	[1] [1, 1, 2]	58 ms
✓	[2] [2, 3, 5]	1 ms
✓	[3] [9, 8, 17]	1 ms
✓	[4] [2, 19, 21]	1 ms

Obraz 6. Wynik wywołania testu parametrized

Arguments

Przykład poniżej zostanie pokazany dopiero w ćwiczeniu praktycznym, ale w kontekście notatek chciałbym umieścić to tutaj.

W testach parametryzowanych mogą wystąpić sytuacje, gdzie dostarczenie danych testowych w postaci dwuwymiarowej tablicy typu `Object` nie zadziała i będziemy dostawali błędy. W takiej sytuacji należy wykorzystać inną konstrukcję. Przykład zapisu poniżej.

```
package pl.zajavka;

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

class CalculatorTest {

    @ParameterizedTest
    @MethodSource ①
    void testParametrizedAdding(int left, int right, int expected) { ②
        // given, when
        Integer result = Calculator.add(left, right);

        // then
        Assertions.assertEquals(expected, result);
    }

    @SuppressWarnings("unused")
    public static Stream<Arguments> testParametrizedAdding() { ③
        return Stream.of(
            Arguments.of(1, 1, 2),
            Arguments.of(2, 3, 5),
            Arguments.of(9, 8, 17),
            Arguments.of(2, 19, 21)
        );
    }
}
```

- ① Nadal nie podajemy nazwy metody z danymi testowymi jawnie, tylko musimy pamiętać, że nazwa metody z danymi musi być identyczna jak nazwa metody testowej.
- ② Każdy z argumentów, który parametryzujemy został tutaj rozdzielony jako parametry wywołania testu i będą one przypisywane zgodnie z kolejnością argumentów określoną w metodzie z danymi.
- ③ Wykorzystujemy `Stream`, gdzie typem jest interfejs `Arguments`. Możemy wtedy wykorzystać albo metodę `Arguments.of()` albo `Arguments.arguments()`.

W zapisie powyżej kod nadal zachowa się tak samo, czyli uruchomią nam się testy z parametrami. Który zapis jest czytelniejszy - zostawiam to do Twojej oceny 😊. Ja osobiście najczęściej korzystam ze `Stream<Arguments>`.

Tak jak zostało wspomniane wcześniej, ten przykład zostanie pokazany w rozwiązaniu ćwiczenia praktycznego.



Jeżeli potrzebujesz lepiej zobaczyć jak się to zachowa, pamiętaj zawsze możesz taki Debugować.

Błędy



Jeżeli ktoś dostaje błędy w stylu:

```
java.lang.NoSuchMethodError:
org.junit.jupiter.api.extension.ExtensionContext
    .getConfigurationParameter(java.lang.String, java.util.function.Function)
```

Może to wynikać z niezgodności wersji bibliotek. Niestety, biblioteki mają do siebie, że jeżeli używamy kilku to muszą być ze sobą kompatybilne. Jeżeli natomiast otrzymujesz takie błędy to jak je rozwiązać? Googlować, w praktyce zawsze się to tak kończy.

Logowanie

Wróćmy jeszcze na moment do tematu logowania w zestawieniu z uruchamianiem testów. Stwórz katalog `src/test/resources` i dodaj w nim plik `logback-test.xml`, w którym określisz następującą konfigurację:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="WARN">
        <appender-ref ref="CONSOLE"/>
    </root>
```

```
</configuration>
```

Do tego stwórzmy plik `logback.xml` w katalogu `src/main/resources` i dodajmy w nim identyczną konfigurację jak powyższa, tyle, że w `root` zamiast `WARN` napiszmy `INFO`.

Następnie w klasie `Calculator` dodaj logowanie (np. w metodzie `add()`), żebyśmy mieli tam przykładowo taki kod:

```
package pl.zajavka;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class Calculator {

    public static Integer add(int left, int right) {
        log.debug("DEBUG message");
        log.info("INFO message");
        log.warn("WARN message");
        log.error("ERROR message");
        return left + right;
    }
}
```

Teraz uruchom test `testAdd()` a następnie uruchom metodę `Calculator.add()` normalnie, np. z metody `main()`. Zobacz jaka będzie różnica w logach pokazywanych na ekranie.

Jeżeli przy powyższej konfiguracji uruchomimy kod normalnie (nie z testu), to na ekranie logują nam się wiadomości `INFO`, `WARN` i `ERROR`. Jeżeli uruchomimy natomiast metodę testową to na ekranie logują się tylko wiadomości `WARN` i `ERROR`. Wynika to z tego, że rozdzieliliśmy konfigurację logowania na dwie różne - jedna ma być używana normalnie w kodzie, a druga tylko w testach. Możemy to rozumieć jak 2 classpathy, jeden classpath jest używany tylko w kodzie, a drugi w testach. Tak samo zachowywałyby się inne pliki, które umieścimy w katalogu `src/test/resources` - będą one dostępne w kontekście testów.

Taki podział często jest przydatny w praktyce w sytuacjach, gdy chcemy ustawić inną konfigurację (czy to logowania czy czego innego) podczas normalnego wykonania programu, a inną w przypadku uruchamiania tego samego kodu aplikacji w testach.