JUnit

Konrad Małek, Bartłomiej Mazgaj

Czym są testy jednostkowe i dlaczego warto je tworzyć?

Testy jednostkowe (ang. unit tests)

to rodzaj automatycznych testów w programowaniu, które mają na celu sprawdzenie poprawności działania **pojedynczych**, najmniejszych jednostek funkcjonalnych kodu, zwykle pojedynczych **metod,** sprawdzając, czy działa ona zgodnie z oczekiwaniami w określonych warunkach.

Poprawiają jakość kodu, zapewniają bezpieczeństwo przy zmianie programu, pełnią swego rodzaju dokumentacji

Jakie powinny być testy jednostkowe?

Testy powinny być pisane w myśl zasady FIRST:

- **FAST** — testy powinny być uruchamiane szybko i regularnie, najlepiej bezpośrednio po dokonaniu zmian w kodzie.
- **INDEPENDENT** — niezależność testów, pozwala na natychmiastowe zidentyfikowanie źródła błędów.
- **REPEATABLE** — testy powinny być powtarzalne i niezawodne, niezależnie od środowiska ich uruchamiania.
- **SELF-CHECKING** — testy powinny dawać automatycznie jednoznaczną informacja zwrotną o poprawności kodu.
- **TIMLY** — testy powinny być tworzone jednocześnie wraz z pisanym kodem (sprzyja to lepszej architekturze i bardziej przemyślanym rozwiązaniom).


CZĘŚĆ INSTALACYJNA

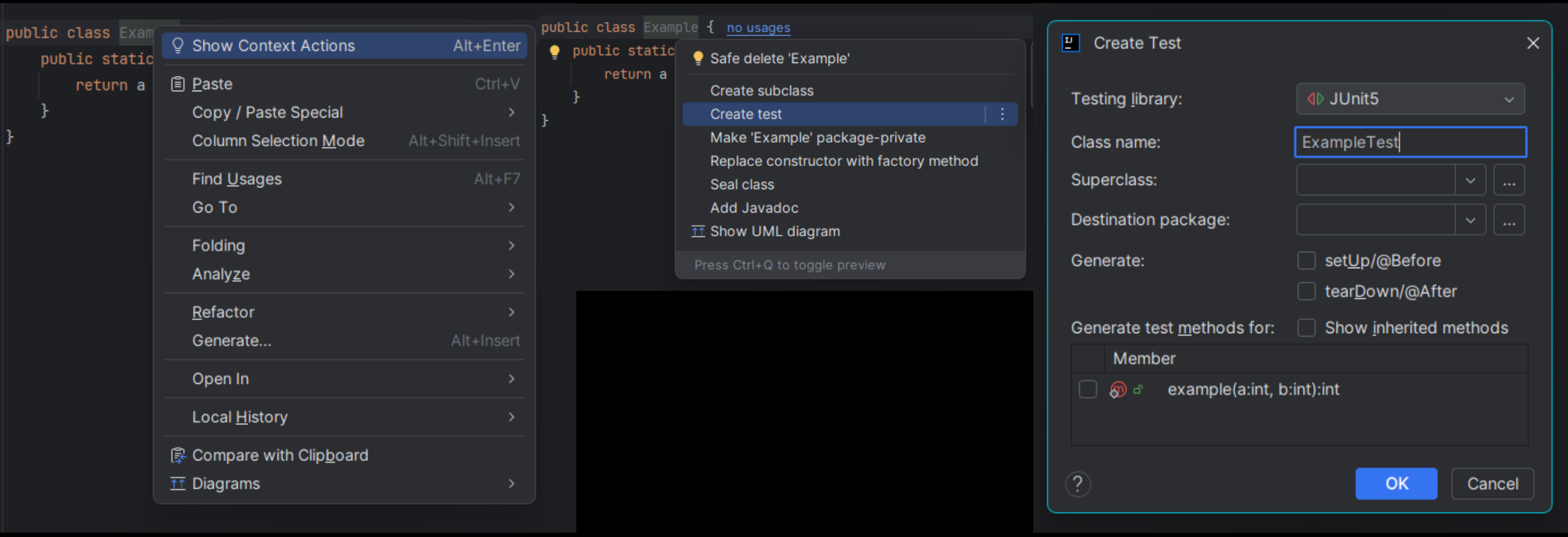
Jak dodać JUnit

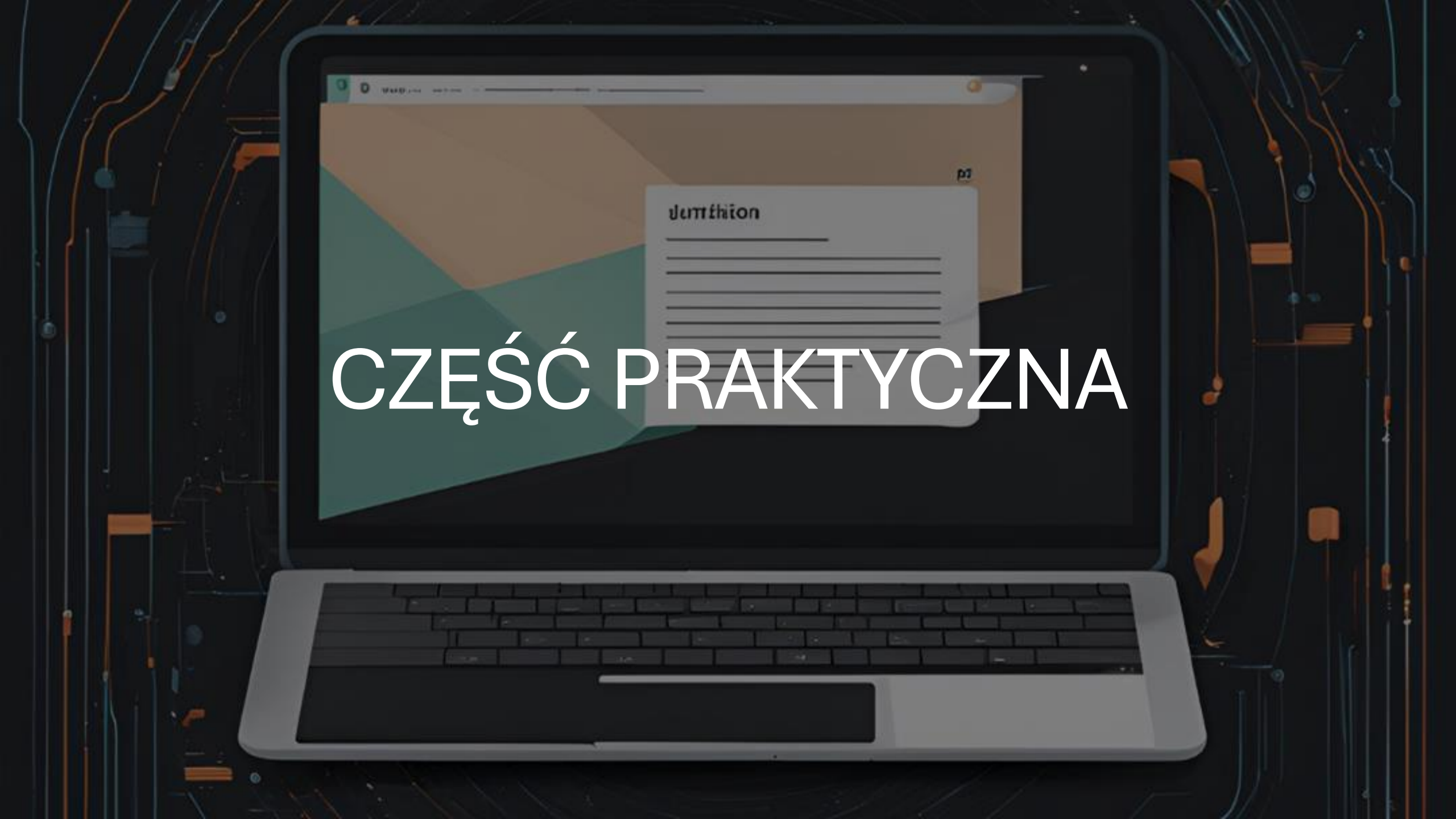
Żeby dodać Junit do projektu maven, trzeba w pliku pom.xml dodać zależność:

```
16      <dependencies>
17          <dependency>
18              <groupId>org.junit.jupiter</groupId>
19              <artifactId>junit-jupiter-engine</artifactId>
20              <version>5.11.3</version>
21              <scope>test</scope>
22          </dependency>
23      </dependencies>
```

Dodanie w Intellij

Intellij umożliwia automatyczne dodawanie testów do istniejących klas





CZĘŚĆ PRAKTYCZNA

Prosty Test - przykład

```java
public class Example {  2 usages
    public String example(String input) {  1 usage
        return "Example: " + input;
    }
}
```

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
class ExampleTest {
    private final Example example1 = new Example();  1 usage
    @Test
    void example() {
        assertEquals( expecte "Example: string", example1.example( input "string"), messag "Wrong example");
    }
}
```

Rodzaje assercji

Testy nie zwracają żadnej wartości - o powodzeniu testu decydują zawarte w nim asercje.

Przykładowe asercje zawarte w bibliotece JUnit 5:

- **`assertEquals(expected, actual)`** - sprawdza czy wartości są równe sobie
- **`assertAll(code)`** - pozwala na wykonanie wielu asercji w jednym teście, a wszystkie wyniki są raportowane razem.
- **`assertNotNull(value)`** - sprawdza, czy podana wartość jest null

```java
@Test
public void testAdd() {
    // Given
    Calculator calculator = new Calculator();

    // When
    Integer result = calculator.add(2, 3);

    // Then
    assertNotNull(result, "Result should not be null");
    assertEquals(5, result, "Sum should be equal to 5");
}
```

```java
@Test
public void testPersonProperties() {
    Person person = new Person("John Doe", 30, "123 Main St");

    // Sprawdzamy wiele asercji jednocześnie
    assertAll("person",
        () -> assertEquals("John Doe", person.getName(), "Name should be John Doe"),
        () -> assertEquals(30, person.getAge(), "Age should be 30"),
        () -> assertEquals("123 Main St", person.getAddress(), "Address should be 123
    );
}
```

Rodzaje assercji

Przykładowe asercje zawarte w bibliotece JUnit 5:

- **`assertThrows(exception.class, () -> {code})`** - sprawdza czy dany kod wyrzuca podany wyjątek
- **`assertTimeout()`** - czy dany fragment kodu wykonuje się w określonym czasie
- **`assertTrue(condition)` / `assertFalse(condition)`**

```java
@Test
public void testPerformLongTaskTimeout() {
    Task task = new Task();

    // Sprawdzamy, czy metoda wykonuje się w mniej niż 3 sekundy
    assertTimeout(Duration.ofSeconds(3), () -> {
        String result = task.performLongTask();
        assertEquals("Task Completed", result);
    });
}
```

```java
@Test
public void testDivideByZero() {
    // Given
    Calculator calculator = new Calculator();

    // When & Then
    assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0),
                 "Expected ArithmeticException when dividing by zero");
}
```

Cykl życia testów

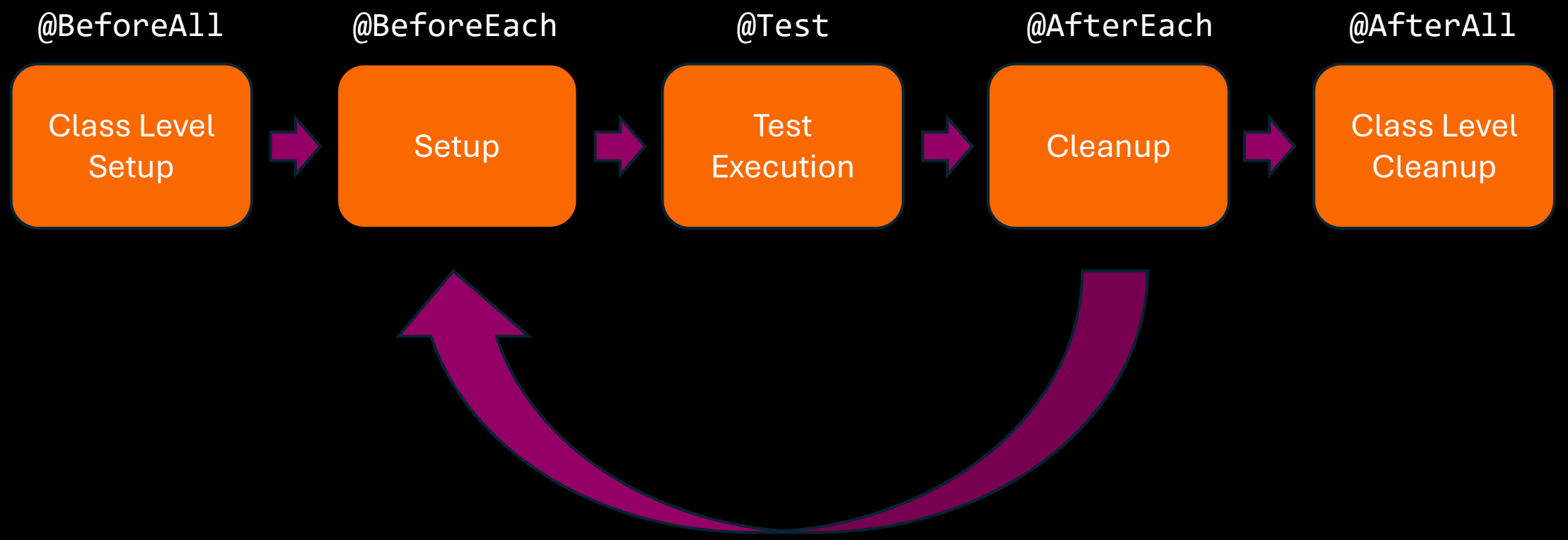
Żeby pozwolić na osobną egzekucję indywidualnych testów jednostkowych, JUnit tworzy nową instancję klasy testowej przed wykonaniem każdej osobnej metody. Jest to podstawowe zachowanie dla JUnit Jupitera.

Możemy to zmienić korzystając z adnotacji:

```
@TestInstance(Lifecycle.PER_CLASS)
```

Która spowoduje tworzenie nowych instancji raz na klasę testów. Może to spowodować konieczność sformułowania metod przygotowującej oraz sprzątającej po testach jednostkowych za pomocą adnotacji odpowiednio `@BeforeEach, @AfterEach`

Dodatkowo możemy zdefiniować metody wykonujące się przed i po wszystkimi testami jednostkowymi danej klasy testowej za pomocą adnotacji **`@BeforeAll`** oraz **`@AfterAll`**



Tagi do testów

- W JUnit 5, tagi (ang. **tags**) są używane do kategoryzowania testów, co ułatwia ich grupowanie, uruchamianie lub filtrowanie. Dzięki tagom można łatwo zarządzać dużymi zestawami testów, wybierając do uruchomienia tylko te, które należą do określonej kategorii. Tagi mogą być używane do oznaczania testów jako np. "powolne", "szybkie", "regresyjne", "integracyjne" itp.

```java
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```

```java
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

public class SampleTest {

    @Tag("fast")
    @Test
    void fastTest() {
        // Test, który jest szybki
    }

    @Tag("slow")
    @Test
    void slowTest() {
        // Test, który jest wolny
    }

    @Tag("integration")
    @Test
    void integrationTest() {
        // Test integracyjny
    }
}
```

Tagi w połączeniu z Maven

Uruchamianie metody fastTest()

- `mvn -Dtest=SampleTest#fastTest test`

Uruchomienie wszystkich metod w klasie

- `mvn -Dtest=SampleTest test`

Uruchomienie wszystkich metod o tagu "fast" lub "slow"

- `mvn test –Dgroups="fast | slow"`

```java
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

public class SampleTest {

    @Tag("fast")
    @Test
    void fastTest() {
        // Test, który jest szybki
    }

    @Tag("slow")
    @Test
    void slowTest() {
        // Test, który jest wolny
    }

    @Tag("integration")
    @Test
    void integrationTest() {
        // Test integracyjny
    }
}
```

Parametryzacja

Parametryzacja testów w JUnit pozwala na uruchamianie, tego samego testu z różnymi zestawami danych wejściowych. Dzięki temu można łatwo sprawdzić, jak metoda zachowuje się w różnych warunkach, co prowadzi do bardziej zwięzłego i efektywnego kodu testowego

Adnotacja **`@ParameterizedTest`** wskazuje, że metoda testowa jest testem parametryzowanym.

```java
@ParameterizedTest
@ValueSource(ints = {2, 4, 6, 8, 10})
void isEven(int number) {
    assertTrue(number % 2 == 0, number + " should be even");
}
```

```java
@ParameterizedTest
@CsvSource({
    "hello, 5",
    "world, 5",
    "JUnit, 5",
    "test, 4"
})
void stringLength(String input, int expectedLength) {
    assertEquals(expectedLength, input.length());
}
```

Parametryzacja

Dostarczanie danych za pomocą źródeł danych: Można używać różnych dostawców danych, takich jak:

- `@ValueSource` - Dostarcza pojedyncze wartości (np. liczby, łańcuchy).
- `@MethodSource` - Pozwala na użycie metody, która zwraca strumień danych do testów.

```java
@ParameterizedTest
@ValueSource(ints = {2, 4, 6, 8, 10})
void isEven(int number) {
    assertTrue(number % 2 == 0, number + " should be even");
}

@ParameterizedTest
@MethodSource("provideNumbers")
void isPrime(int number) {
    assertTrue(isPrimeNumber(number));
}

static Stream<Integer> provideNumbers() {
    return Stream.of(2, 3, 5, 7, 11, 13);
}
```

```java
package example;

import java.util.stream.Stream;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class ExternalMethodSourceDemo {

    @ParameterizedTest
    @MethodSource("example.StringsProviders#tinyStrings")
    void testWithExternalMethodSource(String tinyString) {
        // test with tiny string
    }
}

class StringsProviders {

    static Stream<String> tinyStrings() {
        return Stream.of(".", "oo", "OOO");
    }
}
```

Parametryzacja

Dostarczanie danych za pomocą źródeł danych: Można używać różnych dostawców danych, takich jak:

- `@CsvSource` - Umożliwia dostarczanie wielu wartości w formacie CSV.
- `@CsvFileSource(files = "src/test/resources/two-column.csv")` - Umożliwia dostarczenie danych z pliku

```
@ParameterizedTest(name = "[{index}] {arguments}")
@CsvFileSource(resources = "/two-column.csv", useHeadersInDisplayName = true)
void testWithCsvFileSourceAndHeaders(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

```
@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

```
@ParameterizedTest
@CsvSource({
    "hello, 5",
    "world, 5",
    "JUnit, 5",
    "test, 4"
})
void stringLength(String input, int expectedLength) {
    assertEquals(expectedLength, input.length());
}
```

Repeated tests

JUnit umożliwia powtórzenie testu określoną liczbę razy poprzez adnotację metody `@RepeatedTest` i określenie całkowitej liczby żądanych powtórzeń. Każde wywołanie powtórzonego testu zachowuje się jak wykonanie zwykłej metody `@Test` z pełnym wsparciem dla tych samych wywołań zwrotnych i rozszerzeń cyklu życia.

`@RepeatedTest` przyjmuje następujące parametry:

- `value` – liczba powtórzeń testu
- `failureThreshold` – maksymalna liczba niezaliczonych testów po których reszta nie jest wykonywana; bazowo wartość ta jest ustawiona na `Integer.MAX_VALUE` sygnalizująca brak limitu błędów
- `name` – wyświetlana nazwa powtarzanych testów, można ją modyfikować po przez użycie symboli zastępczych, a jej bazowa forma to **`"repetition {currentRepetition} of {totalRepetitions}"`**:
  - `{displayName}` - wyświetla nazwę metody `@RepeatedTest`
  - `{currentRepetition}` - obecny licznik powtórzeń
  - `{totalRepetitions}` - liczba wszystkich powtórzeń

```java
class RepeatedTests{
    @RepeatedTest(10)
    void repeatedTest() {
        //...
    }

    @RepeatedTest(5)
    void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
        assertEquals( expected: 5, repetitionInfo.getTotalRepetitions());
    }

    @RepeatedTest(value = 8, failureThreshold = 1)
    void repeatedTestWithFailureThreshold(RepetitionInfo repetitionInfo) {
        // Simulate unexpected failure every second repetition
        if (repetitionInfo.getCurrentRepetition() % 2 == 0) {
            fail("Boom!");
        }
    }

    @RepeatedTest(value = 1, name = "{displayName} {currentRepetition}/{totalRepetitions}")
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        assertEquals( expected: "Repeat! 1/1", testInfo.getDisplayName());
    }
```

Dodatkowo, do metody `@RepeatedTests` można przekazać instancję klasy **RepetitionInfo,** która umożliwia dostęp do inforamcji o obecnym powtórzeniu testu. Jej metody to:

- `getCurrentRepetition()`
- `getFailureCount()`
- `getFailureThreshold()`
- `getTotalRepetitions()`

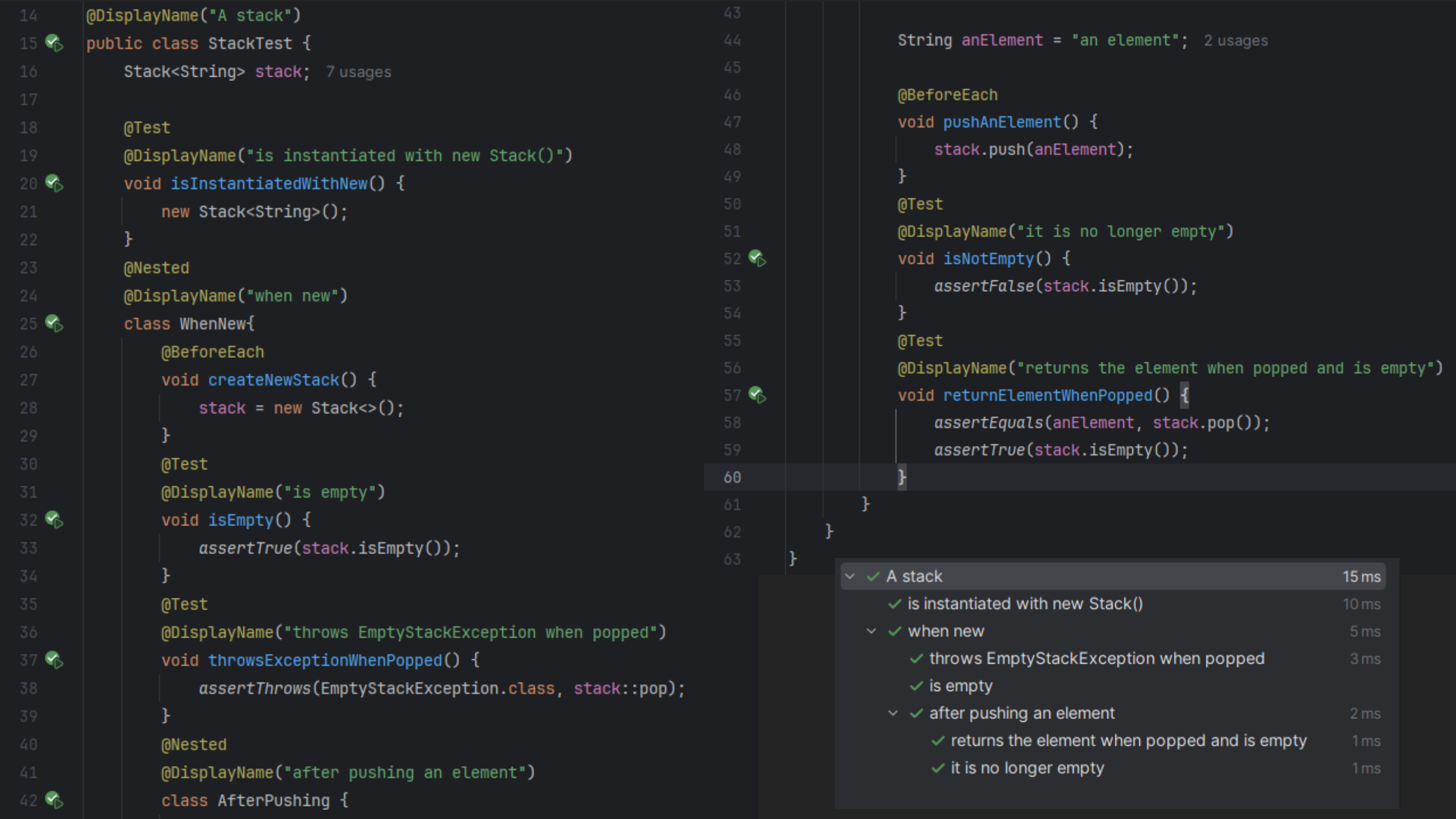
Nested tests

Zagnieżdżone testy dają autorowi testów więcej możliwości wyrażania relacji między kilkoma grupami testów. Takie zagnieżdżone testy wykorzystują zagnieżdżone klasy Javy i ułatwiają hierarchiczne myślenie o strukturze testu.

Adnotacja `@Nested` wskazuje, że dana klasa testów jest klasą zagnieżdżoną.

Warto wiedzieć:

- Metoda `@BeforEach` klasy wyższej wykona się przed odpowiadającą jej metodą klasy zagnieżdżonej
- Metoda `@AfterEach` klasy wyższej wykona się po odpowiadającej jej metodzie klasy zagnieżdżonej


@DisplayName("A stack")
public class StackTest {
Stack<String> stack; 7 usages
@Test
@DisplayName("is instantiated with new Stack()")
void isInstantiatedWithNew() {
new Stack<String>();
}
@Nested
@DisplayName("when new")
class WhenNew{
@BeforeEach
void createNewStack() {
stack = new Stack<>();
}
@Test
@DisplayName("is empty")
void isEmpty() {
assertTrue(stack.isEmpty());
}
@Test
@DisplayName("throws EmptyStackException when popped")
void throwsExceptionWhenPopped() {
assertThrows(EmptyStackException.class, stack::pop);
}
@Nested
@DisplayName("after pushing an element")
class AfterPushing {
String anElement = "an element"; 2 usages
@BeforeEach
void pushAnElement() {
stack.push(anElement);
}
@Test
@DisplayName("it is no longer empty")
void isNotEmpty() {
assertFalse(stack.isEmpty());
}
@Test
@DisplayName("returns the element when popped and is empty")
void returnElementWhenPopped() {
assertEquals(anElement, stack.pop());
assertTrue(stack.isEmpty());
}
}
}
}
A stack 15 ms
is instantiated with new Stack() 10 ms
when new 5 ms
throws EmptyStackException when popped 3 ms
is empty
after pushing an element 2 ms
returns the element when popped and is empty 1 ms
it is no longer empty 1 ms

Disabled

Za pomocą adnotacji `@Disabled` możne wyłączać całe klasy testów bądź poszczególne metody.

Jeżeli jest zaaplikowana do klasy wtedy wyłącza wszystkie zawarte w niej metody

Wyłączenie tylko poszczególnej metody, uniemożliwia również wykonanie na niej metod `@BeforEach` i `@AfterEach`, ale nie wpływa na powstanie instancji klasy oraz wykonanie metod `@BeforAll` oraz `@AfterAll`

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled("Disabled until bug #21 has been fixed")
class DisabledTestDemo {

    @Test
    void testWillBeSkipped() {
    }
}
```

```java
class DisabledTestsDemo {

    @Disabled("Disabled until bug #NUH_UH has been resolved")
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }

}
```

bibliografia

- https://junit.org/junit5/docs/current/user-guide/