

Sprawozdanie – stos, kalkulator RPN

Marta Piotrowska, gr. 2

Zadanie 1 - stos

Opis działania programu:

Zaimplementowano klasę **StackMethods**, która implementuje ideę stosów napisów i zawiera następujące metody publiczne:

- **push** – wkładanie jednego elementu na stos,
- **pop** – zdejmowanie jednego elementu ze stosu i oddawanie jego wartości,
- **peek** – oddawanie wartości elementu na szczycie stosu, lecz bez zdejmowania go ze stosu.

W przypadku metod **pop** oraz **peek** uwzględniono obsługę błędów, rzucając wyjątek `EmptyStackException` w sytuacji, gdy stos jest pusty.

Do przechowywania danych wykorzystano klasę `Stack` z biblioteki standardowej Javy. Stos nie posiada ograniczenia rozmiaru, a jego aktualną wielkość można sprawdzić za pomocą metody `getSize`.

Kod:

```
1 package main;
2
3 import java.util.Stack;
4 import java.util.EmptyStackException;
5
6 public class StackMethods { 3 usages
7     private final Stack<String> stack; 7 usages
8
9     public StackMethods() { 1 usage
10         this.stack = new Stack<>();
11     }
12
13     public void pushString(String value) { 1 usage
14         stack.push(value);
15     }
16
17     public String popString() { no usages
18         if (stack.isEmpty()) {
19             throw new EmptyStackException();
20         }
21         return stack.pop();
22     }
23
24     public String peekString() { no usages
25         if (stack.isEmpty()) {
26             throw new EmptyStackException();
27         }
28         return stack.peek();
29     }
}
```

```
public int getSize() { 1 usage
    return stack.size();
}
```

Zadanie 1 – testy

Analiza testów jednostkowych:

Testy sprawdzają poprawność działania metod klasy StackMethods:

- **testPush** – weryfikuje, czy metoda pushString poprawnie dodaje element do stosu i zwiększa jego rozmiar.
- **testPop** – sprawdza, czy metoda popString poprawnie zwraca ostatni dodany element i zmniejsza rozmiar stosu.
- **testPopEmptyStack** – testuje, czy metoda popString rzuca wyjątek EmptyStackException, gdy stos jest pusty.
- **testPeek** – weryfikuje, czy metoda peekString zwraca wartość elementu na szczycie stosu, nie usuwając go.
- **testPeekEmptyStack** – sprawdza, czy metoda peekString rzuca wyjątek EmptyStackException przy pustym stosie.
- **testGetSize** – testuje poprawność metody getSize, sprawdzając liczbę elementów po operacjach pushString i popString.

Testy zapewniają pełne pokrycie funkcjonalności stosu, w tym obsługę wyjątków dla operacji na pustym stosie.

Kod z testami:

```
1 package test;
2
3 import main.StackMethods;
4 import org.junit.Test;
5 import java.util.EmptyStackException;
6
7 import static org.junit.Assert.*;
8
9 public class StackMethodsTest {
10
11     @Test
12     public void testPush() {
13         StackMethods stackMethods = new StackMethods();
14         stackMethods.pushString( value: "miau");
15         assertEquals( expected: 1, stackMethods.getSize());
16     }
```

```

17
18     @Test
19     ▶ public void testPop() {
20         StackMethods stackMethods = new StackMethods();
21         stackMethods.pushString( value: "kot");
22         assertEquals( expected: "kot", stackMethods.popString());
23         assertEquals( expected: 0, stackMethods.getSize());
24     }
25
26     @Test(expected = EmptyStackException.class)
27     ▶ public void testPopEmptyStack() {
28         StackMethods stackMethods = new StackMethods();
29         stackMethods.popString();
30     }
31
32     @Test
33     ▶ public void testPeek() {
34         StackMethods stackMethods = new StackMethods();
35         stackMethods.pushString( value: "pies");
36         assertEquals( expected: "pies", stackMethods.peekString());
37         assertEquals( expected: 1, stackMethods.getSize());
38     }
39
40     @Test(expected = EmptyStackException.class)
41     ▶ public void testPeekEmptyStack() {
42         StackMethods stackMethods = new StackMethods();
43         stackMethods.peekString();
44     }

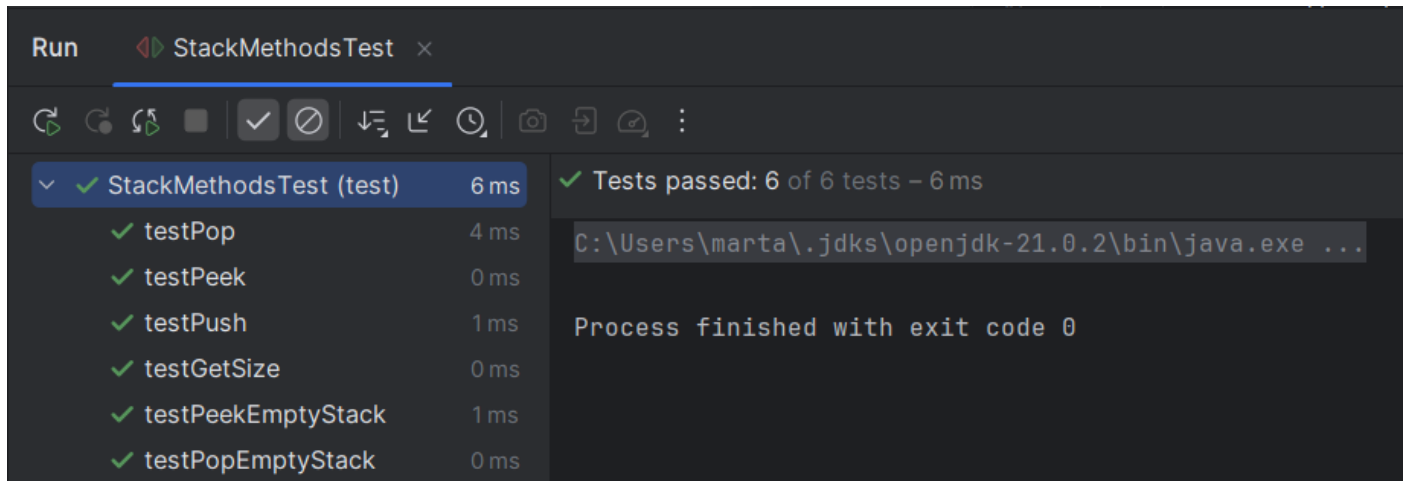
```

```

45
46     @Test
47     ▶ public void testGetSize() {
48         StackMethods stackMethods = new StackMethods();
49         assertEquals( expected: 0, stackMethods.getSize());
50         stackMethods.pushString( value: "a");
51         stackMethods.pushString( value: "b");
52         assertEquals( expected: 2, stackMethods.getSize());
53         stackMethods.popString();
54         assertEquals( expected: 1, stackMethods.getSize());
55     }
56 }
57

```

Output:



The screenshot shows the 'Run' window of an IDE. The title bar says 'Run StackMethodsTest x'. Below the title bar is a toolbar with icons for running, debugging, and other actions. The main area is divided into two panes. The left pane shows a list of tests for 'StackMethodsTest (test)' with a total time of 6 ms. The tests and their durations are: testPop (4 ms), testPeek (0 ms), testPush (1 ms), testGetSize (0 ms), testPeekEmptyStack (1 ms), and testPopEmptyStack (0 ms). The right pane shows the output of the tests, which is 'Tests passed: 6 of 6 tests - 6 ms'. Below this, the command line is shown: 'C:\Users\marta\.jdk\openjdk-21.0.2\bin\java.exe ...'. At the bottom, it says 'Process finished with exit code 0'.

```
Run StackMethodsTest x
```

Test	Duration
testPop	4 ms
testPeek	0 ms
testPush	1 ms
testGetSize	0 ms
testPeekEmptyStack	1 ms
testPopEmptyStack	0 ms

Tests passed: 6 of 6 tests - 6 ms

C:\Users\marta\.jdk\openjdk-21.0.2\bin\java.exe ...

Process finished with exit code 0

Zadanie 2 – kalkulator RPN

Opis działania programu:

Zaimplementowano klasę **RPNCalculator** wyliczającą wyrażenia arytmetyczne zapisane w Odwrotnej Notacji Polskiej. Program umożliwia wyliczanie wyrażen złożonych z liczb całkowitych i operacji binarnych takich jak dodawanie, odejmowanie bądź mnożenie.

Do implementacji wykorzystano klasę **StackMethods** z zadania 1.

Kod:

```
1 package main;
2
3 public class RPNCalculator { 15 usages
4     private final StackMethods stack = new StackMethods(); 5 usages
5
6     @
7     public int evaluate(String expression) { 7 usages
8         for (String token : expression.split(regex: "\\s+")) {
9             if (token.matches(regex: "-?\\d+")) {
10                 stack.pushString(token);
11             } else {
12                 stack.pushString(String.valueOf(
13                     applyOperator(Integer.parseInt(stack.popString()), Integer.parseInt(stack.popString()), token)
14                 ));
15             }
16         }
17         return Integer.parseInt(stack.popString());
18     }
19
20     @
21     private int applyOperator(int b, int a, String operator) { 1 usage
22         return switch (operator) {
23             case "+" -> a + b;
24             case "-" -> a - b;
25             case "*" -> a * b;
26             default -> throw new IllegalArgumentException("Nieznany operator: " + operator);
27         };
28     }
29 }
```

Zadanie 2 – testy

Testy sprawdzają poprawność działania metod klasy `RPNCalculator`:

- **testSimpleAddition** – weryfikuje, czy kalkulator poprawnie wykonuje operację dodawania w notacji odwrotnej polskiej,
- **testSimpleSubtraction** – sprawdza poprawność operacji odejmowania,
- **testSimpleMultiplication** – testuje działanie mnożenia,
- **testComplexExpression** – sprawdza poprawność obliczania bardziej złożonego wyrażenia, składającego się z wielu operatorów i liczb,
- **testInvalidOperator** – weryfikuje, czy kalkulator rzuca wyjątek `IllegalArgumentException` w przypadku użycia nieznanego operatora.

Kod z testami:

```
1 package test;
2
3 import main.RPNCalculator;
4 import org.junit.Test;
5 import static org.junit.Assert.*;
6
7 public class RPNCalculatorTest {
8
9     @Test
10    public void testSimpleAddition() {
11        assertEquals("expected: 5, new RPNCalculator().evaluate( expression: \"2 3 +\")");
12    }
13
14    @Test
15    public void testSimpleSubtraction() {
16        assertEquals("expected: 1, new RPNCalculator().evaluate( expression: \"4 3 -\")");
17    }
18
19    @Test
20    public void testSimpleMultiplication() {
21        assertEquals("expected: 12, new RPNCalculator().evaluate( expression: \"4 3 *\")");
22    }
23
24    @Test
25    public void testComplexExpression() {
26        assertEquals("expected: 14, new RPNCalculator().evaluate( expression: \"5 1 2 + 4 * + 3 -\")");
27    }
28
29    @Test(expected = IllegalArgumentException.class)
30    public void testInvalidOperator() {
31        new RPNCalculator().evaluate("2 3 &"); // Niepoprawny operator
32    }
33
34 }
```

```
28
29
30    @Test(expected = IllegalArgumentException.class)
31    public void testInvalidOperator() {
32        new RPNCalculator().evaluate("2 3 &"); // Niepoprawny operator
33    }
34 }
```

Output:

✓ RPNCalculatorTest (test)	5 ms	✓ Tests passed: 5 of 5 tests – 5 ms
✓ testInvalidOperator	4 ms	C:\Users\marta\.jdk\openjdk-21.0.2\bin\java.exe ...
✓ testSimpleMultiplication	0 ms	
✓ testSimpleAddition	0 ms	Process finished with exit code 0
✓ testSimpleSubtraction	1 ms	
✓ testComplexExpression	0 ms	