МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра ИИТ

ЛАБОРАТОРНАЯ РАБОТА №11

по дисциплине СПП

Тема: «Тестирование кода с использованием библиотеки JUnit»

Выполнил: студент группы ПО-5

Корольчук А. С.

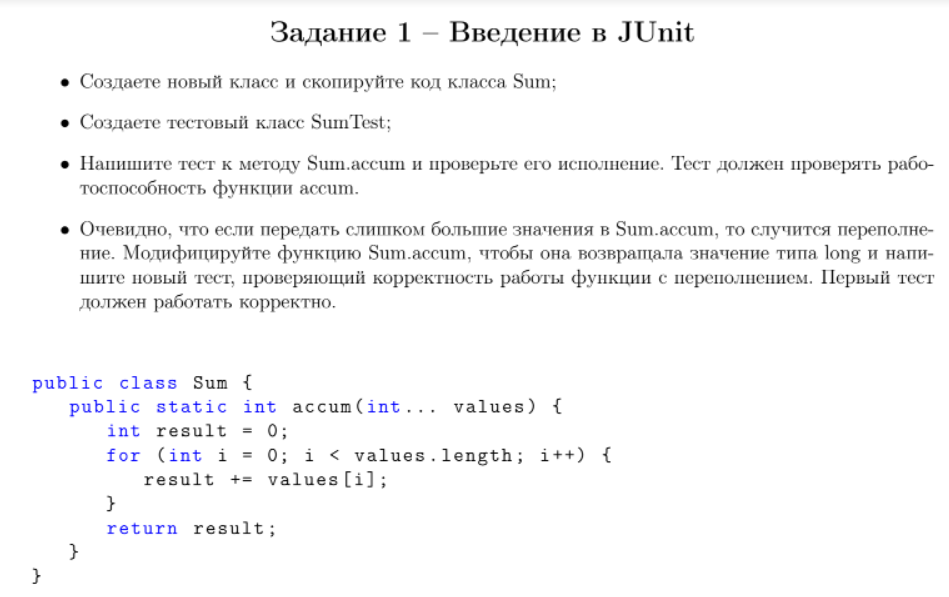
Проверил: преподаватель

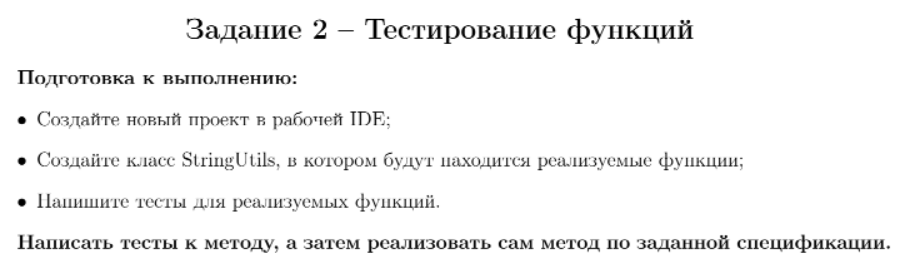
Крощенко А. А.

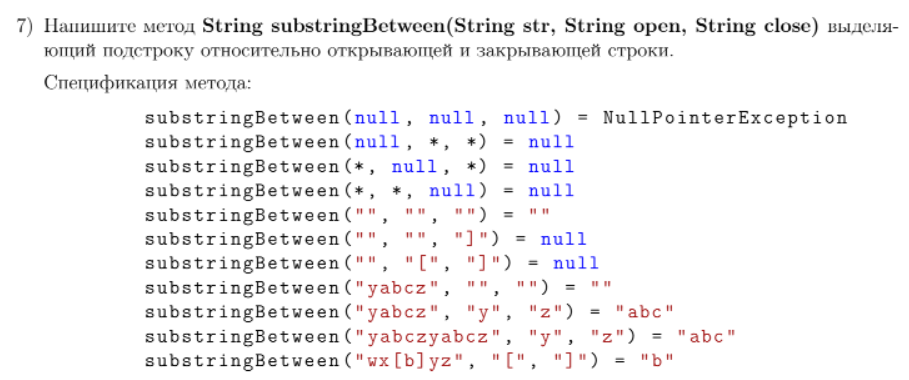
Брест, 2022

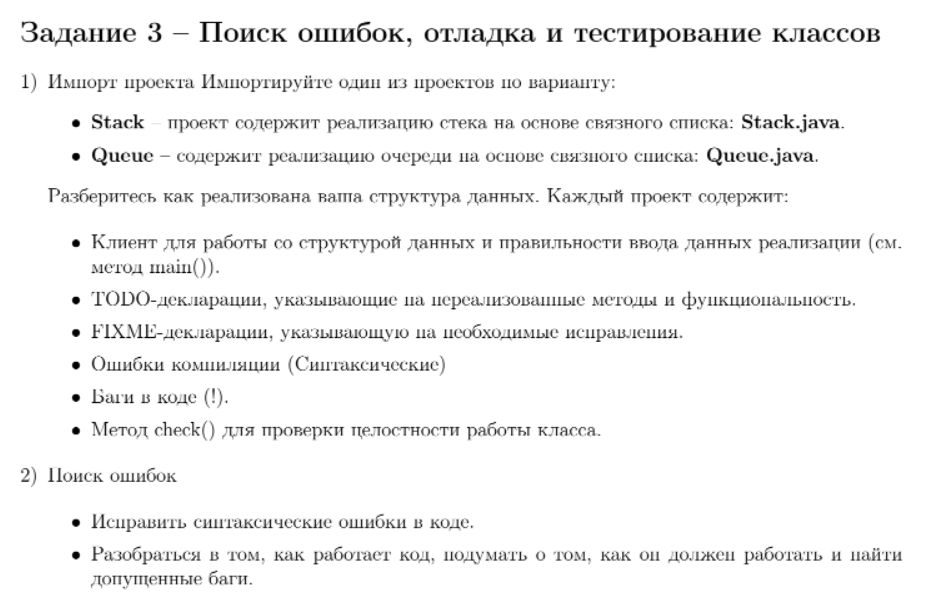
Цель: освоить приемы тестирования кода на примере использования библиотеки JUnit.

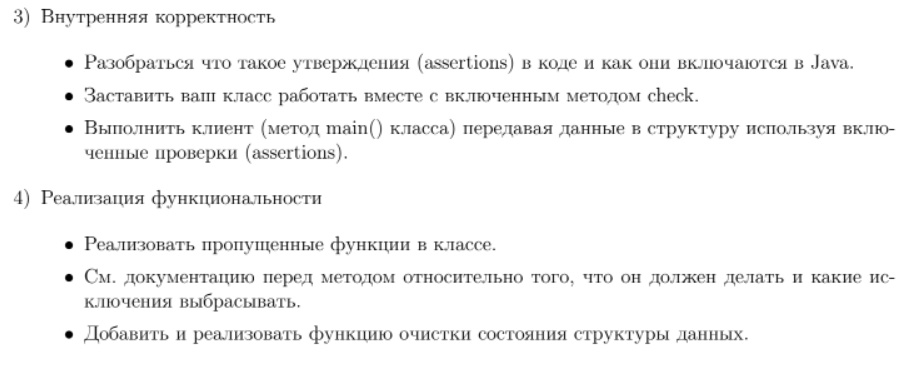
**Задание**

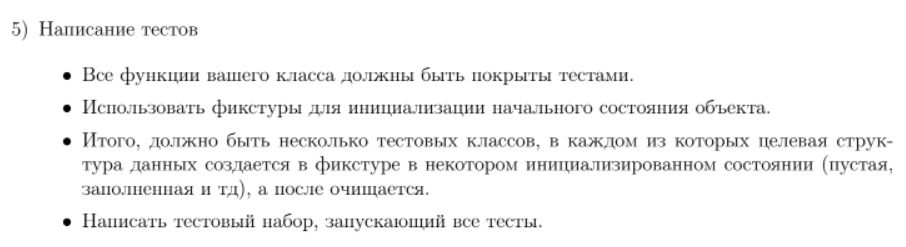












**Ход работы**

**Задание 1**

Реализация

*Тестируемые функции*

5 public static int ***accum*** ( int ... values ) {

6 int result = 0;

7 for ( int i = 0; i < values . length ; i ++) {

8 result += values [ i ];

9 }

10 return result ;

11 }

12

13 public static long ***accumLong*** ( int ... values ) {

14 long result = 0;

15 for ( int i = 0; i < values . length ; i ++) {

16 result += values [ i ];

17 }

18 return result ;

19 }

20 }

*Тесты*

32 @Test

33 public void **testAccum**() {

34 System.*out*.println("accum");

35 int[] values = {1, 2, 3};

36 int expResult = 6;

37 int result = Sum.*accum*(values);

38 *assertEquals*(expResult, result);

39 }

40

41 @Test

42 public void **testAccumLong**() {

43 System.*out*.println("accum");

44 int[] values = {1000000000,1000000000, 1000000000};

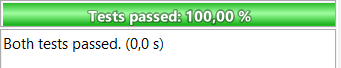
45 long expResult = 3000000000L;

46 long result = Sum.*accumLong*(values);

47 *assertEquals*(expResult, result);

48 }

Результаты



**Задание 2**

Реализация

*Тестируемая функция*

9 public static String ***substringBetween***(String str, String open, String close) {

10 String result = null;

11

12 if (str == null && open == null && close == null)

13 throw new NullPointerException();

14 else if (str == null || open == null || close == null)

15 return null;

16 else if (!Objects.*isNull*(str) && open.equals("") && close.equals(""))

17 return "";

18

19 int openIndex = str.indexOf(open);

20 int closeIndex = str.indexOf(close);

21

22 if (openIndex < 0 || closeIndex < 0)

23 return null;

24 else if (closeIndex < openIndex)

25 return null;

26 else if (openIndex == closeIndex)

27 return "";

28

29 result = str.substring(openIndex + 1, closeIndex);

30

31 return result;

32 }

*Тесты*

32 @Test

33 public void **testSubstringBetween**() {

34 System.*out*.println("substringBetween");

35

36 *assertNull*(StringUtils.*substringBetween* (null, "\*", "\*"));

37 *assertNull*(StringUtils.*substringBetween* ("\*", null, "\*"));

38 *assertNull*(StringUtils.*substringBetween* ("\*", "\*", null ));

39 *assertEquals*(StringUtils.*substringBetween* ("", "", ""), "");

40 *assertNull*(StringUtils.*substringBetween* ("", "", "]"));

41 *assertNull*(StringUtils.*substringBetween* ("", "[", "]"));

42 *assertEquals*(StringUtils.*substringBetween* ("yabcz", "", ""), "");

43 *assertEquals*(StringUtils.*substringBetween* ("yabcz", "y", "z"), "abc");

44 *assertEquals*(StringUtils.*substringBetween* ("yabczyabcz", "y", "z"), "abc");

45 *assertEquals*(StringUtils.*substringBetween* ("wx[b]yz", "[", "]"), "b");

46 }

47

48 @Test

49 public void **testSubstringBetweenNullPointerException**() throws NullPointerException {

50 System.*out*.println("substringBetweenNullPointerException");

51

52 try {

53 StringUtils.*substringBetween* (null, null, null);

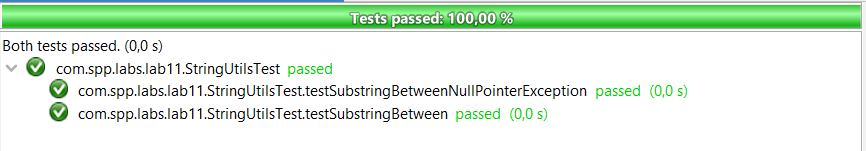
54 } catch (NullPointerException ex) {

55 System.*out*.println("NullPointerException thrown");

56 }

57 }

Результаты



**Задание 3**

Реализация

*Stack*

13 public class **Stack**<Item> {

14 private int N; // size of the stack

15 private Node first; // top of stack

16 private Node last;

17

18 // helper linked list class

19 private class **Node** {

20 private Item item;

21 private Node next;

22 }

23

24 /\*\*

25 \* **Create** **an** **empty** **stack.**

26 \*/

27 public **Stack**() {

28 clear();

29 }

30

31 public void **clear**() {

32 first = null;

33 last = null;

34 N = 0;

35 assert check();

36 }

37

38 /\*\*

39 \* Is the stack empty?

40 \*/

41 public boolean **isEmpty**() {

42 // TODO Implement method

43 return N == 0;

44 }

45

46 /\*\*

47 \* **Return** **the** **number** **of** **items** **in** **the** **stack.**

48 \*/

49 public int **size**() {

50 return N;

51 }

52

53 /\*\*

54 \* **Add** **the** **item** **to** **the** **stack.**

55 \*/

56 public void **push**(Item item) {

57 Node oldfirst = first;

58 first = new Node();

59 first.item = item;

60 first.next = oldfirst;

61 last = first;

62 N++;

63 assert check();

64 }

65

66 /\*\*

67 \* **Delete** **and** **return** **the** **item** **most** **recently** **added** **to** **the** **stack.**

68 \*

69 \* **@throws** java.util.NoSuchElementException if stack is empty.

70 \*/

71 public Item **pop**() {

72 // FIXME throw exception if stack is Empty.

73 if (isEmpty())

74 throw new NoSuchElementException();

75

76 Item item = first.item; // save item to return

77 first = first.next; // delete first node

78 N--;

79 assert check();

80 return item; // return the saved item

81 }

82

83 /\*\*

84 \* **Return** **the** **item** **most** **recently** **added** **to** **the** **stack** **without** **deletion.**

85 \*

86 \* **@throws** java.util.NoSuchElementException if stack is empty.

87 \*/

88 public Item **peek**() {

89 // TODO implement function

90 // FIXME throw exception if stack is Empty.

91 if (isEmpty())

92 throw new NoSuchElementException();

93

94 return first.item;

95 }

96

97 public int **search**(Item searchItem) {

98 if (isEmpty())

99 return -1;

100

101 int counter = 0;

102 Node item = first;

103

104 while (item != null) {

105 if (item.item.equals(searchItem)) {

106 return counter;

107 }

108

109 counter++;

110 item = item.next;

111 }

112

113 return -1;

114 }

115

116 /\*\*

117 \* **Return** **string** **representation.**

118 \*/

119 public String **toString**() {

120 StringBuilder s = new StringBuilder();

121 for (Node current = first; current != null; current = current.next) {

122 Item item = current.item;

123 s.append(item).append(" ");

124 }

125 return s.toString();

126 }

127

128 // check internal invariants

129 private boolean **check**() {

130 if (N == 0) {

131 if (first != null || last != null) {

132 return false;

133 }

134 }

135 else if (N == 1) {

136 if (first == null || last == null) {

137 return false;

138 }

139 if (first.next != null) {

140 return false;

141 }

142 }

143 else {

144 if (first.next == null) {

145 return false;

146 }

147 if (first == null || last == null) {

148 return false;

149 }

150 }

151

152 // check internal consistency of instance variable N

153 int numberOfNodes = 0;

154

155 for (Node x = first; x != null; x = x.next) {

156 numberOfNodes++;

157 }

158

159 if (numberOfNodes != N) {

160 return false;

161 }

162

163 return true;

164 }

165 }

*StackTest*

9 @RunWith(Suite.class)

10 @Suite.*SuiteClasses*({

11 EmptyStackTest.class,

12 NotEmptyStackTest.class

13 })

14 public class **StackTest** {

15

16 }

*EmptyStackTest*

8 public class **EmptyStackTest** {

9 Stack<String> s = new Stack<String>();

10

11 @Before

12 public void **initBefore**() {

13 s.clear();

14 }

15

16 @After

17 public void **initAfter**() {

18

19 }

20

21 /\*\*

22 \* **Test** **of** **clear** **method, of** **class** **Stack.**

23 \*/

24 @Test

25 public void **testClear**() {

26 System.*out*.println("clear");

27

28 s.clear();

29

30 *assertTrue*(s.isEmpty());

31 *assertEquals*(s.size(), 0);

32 }

33

34 /\*\*

35 \* **Test** **of** **isEmpty** **method, of** **class** **Stack.**

36 \*/

37 @Test

38 public void **testIsEmpty**() {

39 System.*out*.println("isEmpty");

40

41 *assertTrue*(s.isEmpty());

42 }

43

44 /\*\*

45 \* **Test** **of** **size** **method, of** **class** **Stack.**

46 \*/

47 @Test

48 public void **testSize**() {

49 System.*out*.println("size");

50

51 *assertEquals*(s.size(), 0);

52 }

53

54 /\*\*

55 \* **Test** **of** **push** **method, of** **class** **Stack.**

56 \*/

57 @Test

58 public void **testPush**() {

59 System.*out*.println("push");

60

61 s.push("1");

62

63 *assertEquals*(s.size(), 1);

64 *assertEquals*(s.search("1"), 0);

65 *assertEquals*(s.peek(), "1");

66 }

67

68 /\*\*

69 \* **Test** **of** **pop** **method, of** **class** **Stack.**

70 \*/

71 @Test

72 public void **testPop**() throws NoSuchElementException {

73 System.*out*.println("pop");

74

75 try {

76 s.pop();

77 } catch (NoSuchElementException ex) {

78 System.*out*.println("NoSuchElementException");

79 }

80 }

81

82 /\*\*

83 \* **Test** **of** **peek** **method, of** **class** **Stack.**

84 \*/

85 @Test

86 public void **testPeek**() throws NoSuchElementException {

87 System.*out*.println("peek");

88

89 try {

90 s.peek();

91 } catch (NoSuchElementException ex) {

92 System.*out*.println("NoSuchElementException");

93 }

94 }

95

96 /\*\*

97 \* **Test** **of** **search** **method, of** **class** **Stack.**

98 \*/

99 @Test

100 public void **testSearch**() {

101 System.*out*.println("search");

102

103 *assertEquals*(s.search("1"), -1);

104 }

105

106 /\*\*

107 \* **Test** **of** **toString** **method, of** **class** **Stack.**

108 \*/

109 @Test

110 public void **testToString**() {

111 System.*out*.println("toString");

112

113 *assertEquals*(s.toString(), "");

114 }

115 }

*NotEmptyStackTest*

7 public class **NotEmptyStackTest** {

8 Stack<String> s = new Stack<String>();

9

10 @Before

11 public void **initBefore**() {

12 s.clear();

13 s.push("3");

14 s.push("2");

15 s.push("1");

16 }

17

18 @After

19 public void **initAfter**() {

20

21 }

22

23 /\*\*

24 \* **Test** **of** **clear** **method, of** **class** **Stack.**

25 \*/

26 @Test

27 public void **testClear**() {

28 System.*out*.println("clear");

29

30 s.clear();

31

32 *assertTrue*(s.isEmpty());

33 *assertEquals*(s.size(), 0);

34 }

35

36 /\*\*

37 \* **Test** **of** **isEmpty** **method, of** **class** **Stack.**

38 \*/

39 @Test

40 public void **testIsEmpty**() {

41 System.*out*.println("isEmpty");

42

43 *assertFalse*(s.isEmpty());

44 }

45

46 /\*\*

47 \* **Test** **of** **size** **method, of** **class** **Stack.**

48 \*/

49 @Test

50 public void **testSize**() {

51 System.*out*.println("size");

52

53 *assertEquals*(s.size(), 3);

54 }

55

56 /\*\*

57 \* **Test** **of** **push** **method, of** **class** **Stack.**

58 \*/

59 @Test

60 public void **testPush**() {

61 System.*out*.println("push");

62

63 s.push("0");

64

65 *assertEquals*(s.size(), 4);

66 *assertEquals*(s.search("3"), 3);

67 *assertEquals*(s.peek(), "0");

68 }

69

70 /\*\*

71 \* **Test** **of** **pop** **method, of** **class** **Stack.**

72 \*/

73 @Test

74 public void **testPop**() {

75 System.*out*.println("pop");

76

77 *assertEquals*(s.pop(), "1");

78 *assertEquals*(s.size(), 2);

79 }

80

81 /\*\*

82 \* **Test** **of** **peek** **method, of** **class** **Stack.**

83 \*/

84 @Test

85 public void **testPeek**() {

86 System.*out*.println("peek");

87

88 *assertEquals*(s.peek(), "1");

89 }

90

91 /\*\*

92 \* **Test** **of** **search** **method, of** **class** **Stack.**

93 \*/

94 @Test

95 public void **testSearch**() {

96 System.*out*.println("search");

97

98 *assertEquals*(s.search("1"), 0);

99 *assertEquals*(s.search("2"), 1);

100 *assertEquals*(s.search("3"), 2);

101 }

102

103 /\*\*

104 \* **Test** **of** **toString** **method, of** **class** **Stack.**

105 \*/

106 @Test

107 public void **testToString**() {

108 System.*out*.println("toString");

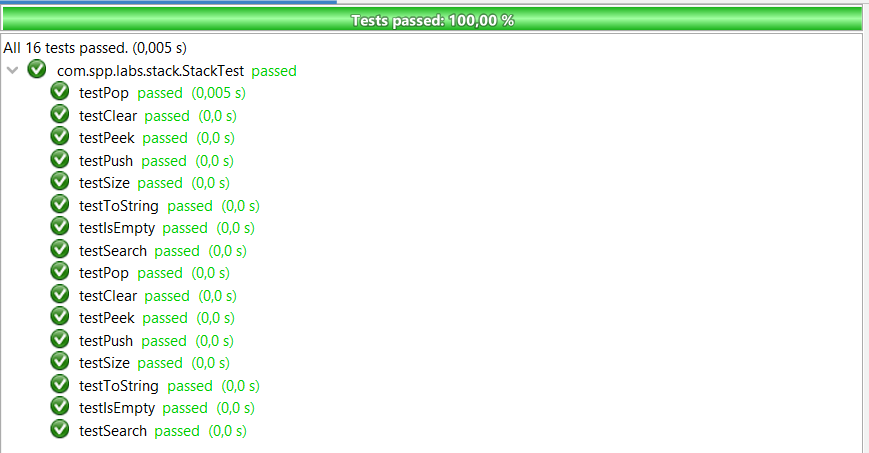
109

110 *assertEquals*(s.toString(), "1 2 3 ");

111 }

112 }

Результаты



Вывод: освоил приемы тестирования кода на примере использования библиотеки JUnit.