Software Process Quality

Testing a Stack data structure.

Module code: CA650
Name: Ruben Vasconcelos
Student number: 16212630
Language: Java
Testing framework: JUnit (version4.x)
Code coverage framework: JCov

Stack code:

Generic Bounded stack, taken from Joe Morris' ca213 notes on stacks

```
// http://computing.dcu.ie/~jmorris/ca213/indexLect.html
public class Stack <T> { // bounded
    private T[] seq; // the sequence
    private int size = 0; // size of sequence

    Stack(int n) { // n>0
        seq = (T[])(new Object[n]);
    }

    Stack() {
        seq = (T[])(new Object[10000]);  // or this(10000);
    }

    boolean isEmpty() { return size==0;}

    boolean push(T t) {
        if (size<seq.length) {
                seq[size] = t; size++; return true;
        }
        else return false;
    }

    T pop() {
        if (isEmpty()) return null;
        else {
            size--; return seq[size];
        }
    }
}
```

TestRunner code:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestStack.class);
      for (Failure failure : result.getFailures()) {
         System.out.println("Fail: " + failure.toString());
      }

      System.out.println();
      System.out.println("Number of tests run: " + result.getRunCount());
      System.out.println("Number of tests that failed: " + result.getFailureCount());
      System.out.println("Overall result: " + result.wasSuccessful());
   }
}
```

Tests

```java
import org.junit.Test;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import static org.junit.Assert.fail;

public class TestStack {

    @Test
    public void test_generic_stack() {
        Stack st_generic = new Stack();

        assertTrue(st_generic.isEmpty());

        int int_value = 1;
        String string_value = "1";

        assertNotEquals(int_value, string_value);

        st_generic.push(int_value);
        st_generic.push(string_value);
        assertEquals(st_generic.pop(), string_value);
        assertEquals(st_generic.pop(), int_value);
    }

    @Test
    public void test_type_stack() {
        Stack<Integer> st_int = new Stack<>();
        int num = 1;

        assertTrue(st_int instanceof Stack);
        assertTrue(st_int.isEmpty());
        assertTrue(st_int.push(num));
        assertFalse(st_int.isEmpty());
        // assertEquals(st_int.pop(), num);
        assertTrue(st_int.pop() == num);
        assertTrue(st_int.isEmpty());
    }

    @Test
    public void test_reassign_stack() {
        Stack<String> st_string = new Stack<String>();
        Stack st_generic = st_string;

        assertTrue(st_string.isEmpty());
        assertTrue(st_generic.isEmpty());

        st_string.push(“test”);
        assertFalse(st_string.isEmpty());
        assertFalse(st_generic.isEmpty());

        try {
            st_generic.push(1);
            fail("Expected incompatible types: int cannot be converted to String");
        } catch(Exception e) {
           assertTrue(st_string.isEmpty());
        }
    }

    @Test
    public void test_exceed_lower_bound() {
        Stack<String> st_string = new Stack<>();
        String str = "str";

        assertTrue(st_string.isEmpty());
        assertTrue(st_string.push(str));
        assertFalse(st_string.isEmpty());
        assertEquals(st_string.pop(), str);
        assertTrue(st_string.isEmpty());
        assertEquals(st_string.pop(), null);
    }

    @Test (expected = IndexOutOfBoundsException.class)
    public void test_exceed_upper_bound() {
        int size = 1;
        Stack<Integer> st_int = new Stack<>(size);
        assertTrue(st_int.push(size));
        assertFalse(st_int.push(size));
    }
}
```

Report

Expected behaviour:

1) `test_generic_stack`
2) `test_type_stack`

A great way to find some of the more obvious issues it to create tests that validate an expected output/behaviour. These prove that the code contains the features required, but are not so good for finding errors, faults and failures.

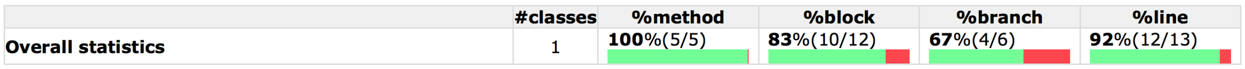
In this case, I actually found that the expected functionally was implemented and that there might be a possible issue with JUnits assertEquals when used with integers. The following worked fine for strings:

```
assertEquals(st_string.pop(), str);
```

But for Integers it did not due to some Object to Long conversion, so I had to use assertTrue:

```
assertEquals(st_int.pop(), num);
assertTrue(st_int.pop() == num);
```

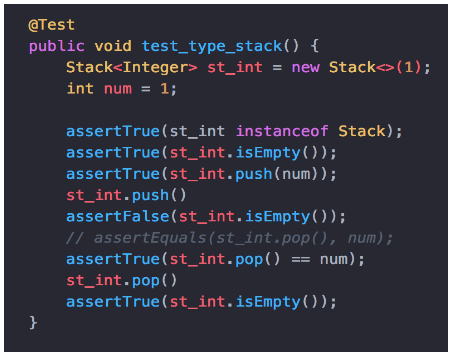
Code coverage usually goes up a lot after a few initial tests as it was the case this time.



	#classes	%method	%block	%branch	%line
Overall statistics	1	**100%**(5/5)	**83%**(10/12)	**67%**(4/6)	**92%**(12/13)

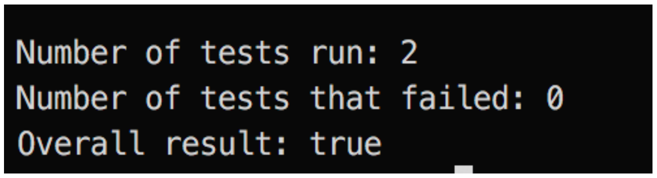
Full coverage fallacy

Code coverage stats have their benefits, but the problem is when tests are designed with the goal of increasing code coverage they often give this false sense of correctness. For example, a slight modification of the tests would archive 100 percent coverage but offer no extra benefits.

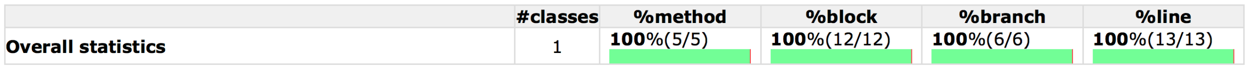


```
@Test
public void test_type_stack() {
    Stack<Integer> st_int = new Stack<>(1);
    int num = 1;

    assertTrue(st_int instanceof Stack);
    assertTrue(st_int.isEmpty());
    assertTrue(st_int.push(num));
    st_int.push()
    assertFalse(st_int.isEmpty());
    // assertEquals(st_int.pop(), num);
    assertTrue(st_int.pop() == num);
    st_int.pop()
    assertTrue(st_int.isEmpty());
}
```



```
Number of tests run: 2
Number of tests that failed: 0
Overall result: true
```



	#classes	%method	%block	%branch	%line
Overall statistics	1	**100%**(5/5)	**100%**(12/12)	**100%**(6/6)	**100%**(13/13)

Experience and understand the tooling:

I realise that I could have modified Joe's stack to use only integers, that way it would require less testing but I wanted to make a point about the test Designer's experience and understanding the tooling. Good test designers and testers are people that often have extensive experience in a particular area; things such as floating point errors or incorrect data type assignments can often be the cause of many failures, which in many cases are very difficult to test for unless you had some previous exposure to these kinds of issues.

```
3) test_reassign_stack
```

With this test case we proved that even though Java is a strongly typed language it is still possible to encounter these kinds of problems; as both the generic stack and the strings stack reference the same object, and we were able to push an integer onto it. In this case that is what makes generic objects possible, nonetheless, it could still be the cause of some failures.

Imagine a scenario where an application used many stacks of different types and there was a function of the following form:

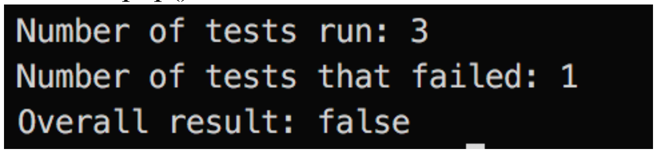
```
some_func(Stack st)
        st.push("1")
```

If that function was actually passed in an integer stack, then the function executing "pop" may throw an error or not, depending on what it does with the data.

Fault: The stack implementation of the push function allows for different types to be pushed onto a specific typed stack.

Error: If the function that does the pop is simply printing out the objects, then it should still work for most cases regardless of the type. E.g println(st.pop())

Failure: If the function that does the pop tries to assign it to a different type E.g: int a = st.pop() **or** 1 + st.pop()



```
Number of tests run: 3
Number of tests that failed: 1
Overall result: false
```

Understand requirements

Testing every possible outcome is usually impossible, test designers must understand the requirements in order to design good tests. A typical approach for testing data structures is testing their bounds, even more so for bounded data structure like the one we're testing.

```
4) test_exceed_lower_bound
5) test_exceed_upper_bound
```

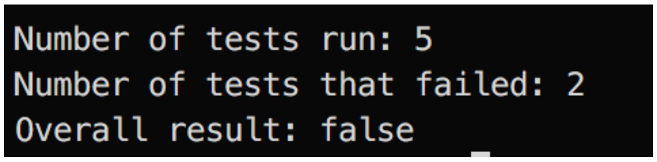
For testing the lower bound I believe that it's enough to show that the stack was populated, then emptied before exceeding the bound and that the correct null value was returned; because any extra calls of "pop" on an empty stack should not have any impact on other parts of the code.

Testing the upper bound it's a bit more tricky. Obviously, Joe M. meant to design the stack in a simple way to teach students. But let's suppose this was a stack to be used by some production software with a producer and a consumer, where the producer keeps pushing integers onto the stack and a consumer that, when given the chance, it pops these times and adds them to a total variable.

Fault: When the upper bound is exceeded no exception is thrown, allowing the code to fail silently. (We would like things to fail early and make as much noise as possible)

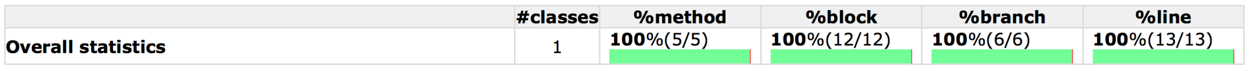
Error: If the producer pushes the number 0 but the stack is already full, that 0 will get lost. Then eventually the consumer wakes up to pop and sum the items in the stack it will never try to add the lost value but it will still get the correct result because it was a zero. (same would happen if numbers that cancel each other out, were pushed after the stack was full E.g 5 and -5)

Failure: Similar to the scenario above but different integers. E.g if 1 was pushed after the stack was full, then the consumer woke up and done the addition, the result would be off by one.



```
Number of tests run: 5
Number of tests that failed: 2
Overall result: false
```

In the end, I still got 100 percent coverage because this application is so small and despite the fact that I was not directly aiming for it. In a real-life scenario, this would not be a feasible goal.



	#classes	%method	%block	%branch	%line
Overall statistics	1	**100%**(5/5)	**100%**(12/12)	**100%**(6/6)	**100%**(13/13)