Assignment	Create JUnit tests of the Stack interface. Test two stack implementations that are provided by StackFactory.
Files	Add these files to your testing project: `ku/util/Stack.java` - interface for Stack. Write tests for this. `StackFactory.jar` - creates stack objects for testing.
Submit	Commit your project to Github. Please using the repo name **`stacktest`**. Write a README.md file containing a list of defects you found in each of the stack implementations. Use one section for each stack (stack type 0 and type 1). Create a table or numbered items to describe the errors. Describe each error in a way that would help a programmer fix his Stack code.

1. Stack Interface

The **`ku.util.Stack`** interface has the methods shown below. A Stack has a type parameter (**T**) which determines the type of objects you can put on the stack. There are several concrete implementations of this interface, which you will test in problem 3.

The Stack methods are

`int capacity( )`	return the maximum number of objects the stack can hold.
`boolean isEmpty( )`	true if the stack is empty, false otherwise.
`boolean isFull( )`	true if stack is full, false otherwise.
`T peek( )`	return the item on the top of the stack, without removing it. If the stack is empty, return **`null`**.
`T pop( )`	Return the item on the top of the stack, and remove it from the stack. Throws: **`java.util.EmptyStackException`** if stack is empty.
`void push( T obj )`	push a new item onto the top of the stack. The parameter (obj) must not be null. Throws: `InvalidArgumentException` if parameter is null. Throws: `IllegalStateException` if push is called when stack is full.
`int size( )`	return the number of items in the stack. Returns 0 if the stack is empty.

2. Write Test Cases on Paper and in Code

2.1 Design your test cases on paper. Test *all* the stack's methods and try to find cases that are likely to fail. You should include these kinds of test cases:
(a) normal cases
(b) borderline valid case, such as a stack of capacity 1 or stack of capacity n containing n-1 or n elements
(c) borderline invalid cases.
(d) weird cases. Things the programmer might not of thought of.

2.2 Create a JUnit test class names StackTest and write test methods. Use descriptive names for methods, such as testStackFullThrowsException, or testPopAllElements.

2.3 While you are writing tests, its helpful to have a "do nothing" implementation. Create a DummyStack class that implements Stack<String> but the methods don't do anything. Most of your unit tests will fail when using this stack, of course.

3. Test Actual Stacks using StackFactory

After you've written some tests, download StackFactory.jar and add it to your project.

StackFactory.jar contains a single class ku.util.StackFactory. This class has 2 methods.

Use makeStack(capacity) to create an untyped Stack (Stack containing any kind of Object).

```
Stack stack = StackFactory.makeStack( 5 );  // stack with capacity 5
```

To create a stack for a particular data type (like String), use a cast as in this example::

```
Stack<String> stack2 = (Stack<String>) StackFactory.makeStack( 10 );
```

Test Two Different Stack Implementations (type 0 and type 1)

StackFactory contains 2 different types of stacks. To specify which kind of stack to create, use:

```
StackFactory.setStackType( 0 ) ;   // make stacks of type 0
StackFactory.setStackType( 1 );    // make stacks of type 1
```

and then call makeStack( ) to create a Stack.

You should find *at least* one error in each type of Stack.

Writing Good Tests

Your objective is to find as many errors as possible. There are several cases or "zones" you should try to test:

- valid cases with no ambiguity
- borderline case that should be valid,: Stack of capacity 1 and push/peek/pop one element.
- borderline case that should be invalid, e.g. pushing 3 items onto a stack of capacity 2.
- "off by one" errors. Push 3 elements to a stack of capacity 2, or push 2 elements and then pop 3.
- weird cases. Things so weird or obviously invalid that the programmer maybe didn't think of them. Such as popping or peeking an empty stack.

Example Test Cases:

Test case	Action	Expected Result
new stack is empty	create stack of size 2 invoke capacity invoke size and isEmpty	 2 0 and true
pop an empty stack	create stack of size 1 invoke pop	 throws EmptyStackException
test peek	puch some elements onto stack. peek the same element 2 times	should always return the same element and not alter the stack
test size	push some elements and call size. peek some elements and call size. pop elements and call size.	size always returns correct number of elements in stack, peek and pop return the top element
test overflow	push element onto a full stack.	throws InvalidStateException

Example

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class StackTest {
     private Stack stack;
     /** "Before" method is run before each test. */
     @Before
     public void setUp( ) {
          stack = makeStack( 2 );
     }

     @Test
     public void newStackIsEmpty() {
          assertTrue( stack.isEmpty() );
          assertFalse( stack.isFull() );
          assertEquals( 0, stack.size() );
     }
     /** pop() should throw an exception if stack is empty */
     @Test( expected=java.util.EmptyStackException.class )
     public void testPopEmptyStack() {
          Assume.assumeTrue( stack.isEmpty() );
          stack.pop();
          // this is unnecessary. For documentation only.
          fail("Pop empty stack should throw exception");
     }

     /** A method to create stacks, so we can change implementation */
     private Stack makeStack(int capacity) {
          // a dummy stack
          return new DummyStack(capacity);
```

Common JUnit Assert methods

assertEquals(*expected, actual*)	For a primitive type, test that expected == actual. For object types, if the expected class has an equals(Object) method then it calls that method, otherwise ==
assertEquals("*message*", *expected*, *actual*)	Same as above. The message string is included in JUnit output if the test fails.
assertEquals(*expect*, *actual*, *tolerance*)	For float and double types, test that the values are equal to within a given tolerance.
assertSame(*expected*, *actual*)	Test if two object variables refer to the same object. This is like: assertTrue(expected == actual)
assertTrue(*boolean_expression*)	Test that the expression is true.
assertFalse(*boolean_expression*)	Test that the expression is false.
assertNull(*variable*)	Test that the value of a variable is null.
assertNotNull(*variable*)	Test that the value of a variable is not null.
assertThat(*expected*, *Matcher*)	Test that expected satisfies some condition, specified by a Matcher. This is more advanced -- study for yourself

References

`http://junit.org` JUnit home. Has many examples and how-to.

`http://junit.org/javadoc/latest/index.html` JUnit Javadoc.