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 instances. See #3 below.

Submit	Commit your project with StackTest.java as "Lab10" in your Bitbucket account. Before you commit anything, create a .gitignore file to ignore class files, Eclipse files, and JAR files:
	`# .gitignore`
	`*.class`
	`*.jar`
	`bin`
	`.settings`
	`.project`
	`.classpath`

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 Write your test cases on paper. Test *all* the stack's methods and try to find cases that are likely to fail. You should include 3 kinds of test cases: (a) normal case, (b) borderline valid case, (c) borderline invalid case. See below for example.
For example:

2.2 Create a JUnit test class names StackTest and write test methods. Use descriptive names for methods.

Since you don't have any real stack implementation yet, create a Dummy stack class that implements the Stack interface. Eclipse will auto-generate "do nothing" method code.

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 "factory" class ku.util.StackFactory. Use makeStack(capacity) to create an untyped Stack (Stack of Object). If you want a stack for a particular data type, use a cast as in the second example. Examples:

```
Stack stack = StackFactory.makeStack( 5 );  // stack with capacity 5
Stack<String> stack2 = (Stack<String>) StackFactory.makeStack( 3 );
```

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

You must find at least one bug in each stack type. In fact, there is more than one bug.

Writing Good Tests

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

- valid case with no ambiguity
- borderline case that should be valid,: Stack of capacity 1 and pushing/popping one element.
- borderline case that should be invalid, e.g. pushing 3 items onto a stack of capacity 2.
- "off by one" errors. See if you can push 3 elements to a stack of capacity 2, or push elements and then pop 3. Another "off by one" might occur if you push 3, pop 2, peek, then push something.
- 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 1 invoke capacity invoke size and isEmpty	 1 0 and true
pop an empty stack	create stack of size 1 invoke pop	throws EmptyStackException
test peek	such some elements onto stack. peek the same element 2 times	should always return same element and not change stack size

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 = new DummyStack( 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");
     }
```

References

`http://junit.org` - JUnit home. Has many links to wiki.

`http://junit.org/javadoc/latest/index.html` - JUnit Javadoc. The most useful class is Assert.