SENG 275

SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING

University of Victoria

TDD WORKED EXAMPLE

Specifications for a Stack

1. As a user I should be able to push into a stack, elements like 3,5,89,103, ‘t’, 4.9, ‘h’, 6.88, etc. however I should not overflow the stack.
2. As a user I should be able to pop elements from a stack but I should check for underflow.
3. As a user I should be able to check the topmost element of the stack, and also whether stack is empty or full.

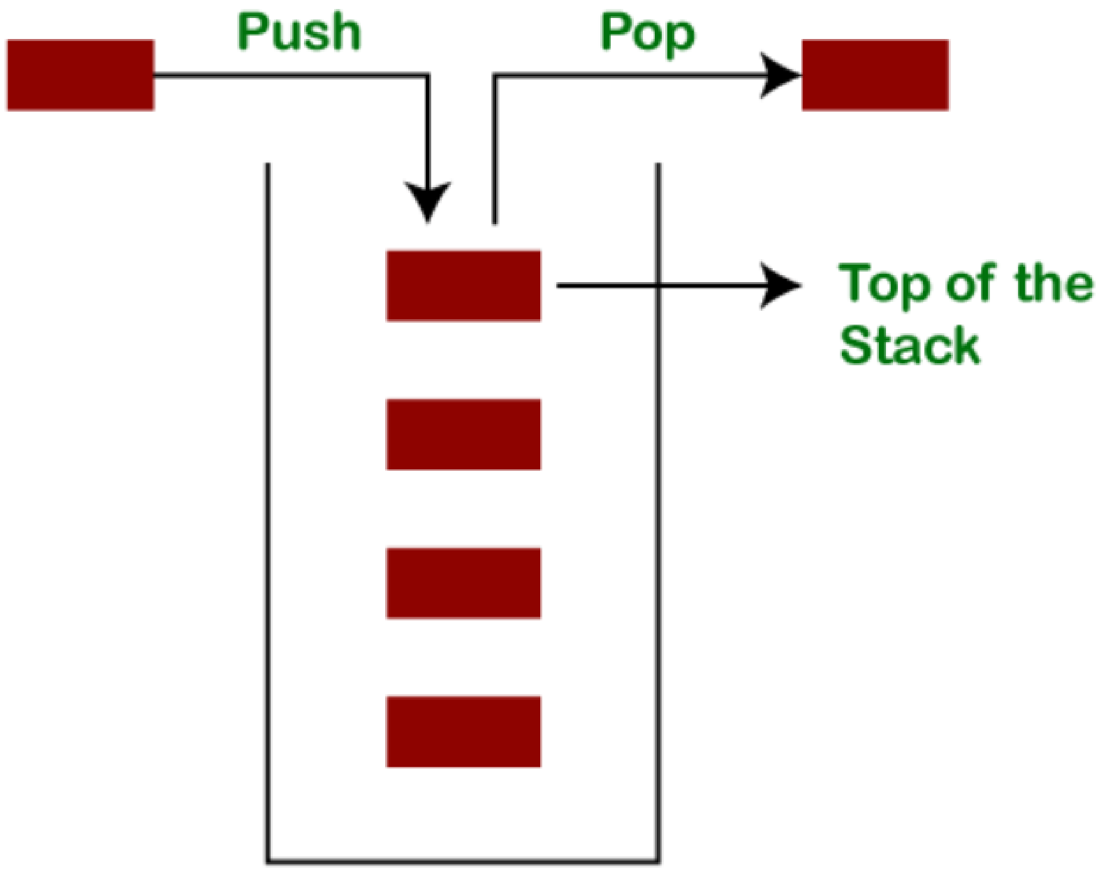
Task: approach the problem by following the TDD methodology. Identify the simplest possible junit test, write the test, fail it, write just enough java code to pass the test. Refactor. Repeat and build the application following this approach.

TDD approach to create a functionality

- **If you write your code after your test, that code is easier to test.** That’s because it’s made to be tested! Testable code is usually clearer code. Remember, try and keep your tests **focused on one thing**. As you write a focused test for each class, TDD encourages you to build software out of lots of little classes, each doing one thing well. This adds to the clarity of the code.
- This also shapes a minimal and **modular** design, meaning that your code isn’t clumped together in a few classes. Modular code is **flexible** and very **easy to change**. Why should you care if it’s easy to change? Any software you build at work will last for many years and continuously change as your **users’ needs also change.**

University of Victoria

Stack

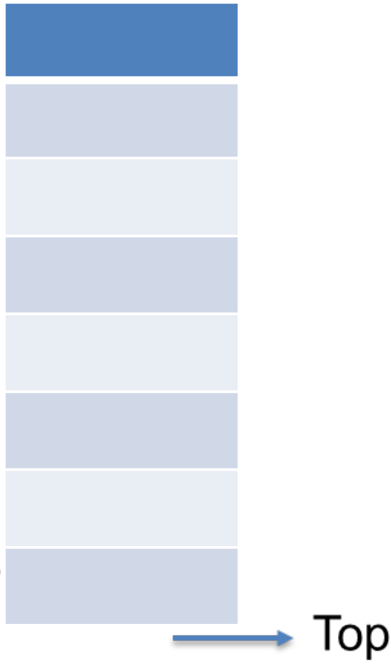


TDD approach demonstration using generic Stack to accept integer and String values

You need to write a code for stack:

1. Test for initial **integer** stack, **size 0.** Test fail. Code to pass the test. Execute the test and pass it.
2. Test for **pushing single item** on stack. Test fail. Code to pass the test. Execute the test and pass it.
3. **Refactor** without functionality change: **common object** created for all tests.
4. Pass the test.
5. **Refactor:** Make stack **generic** so that it can accept integers, strings and similar data types values.
6. Pass the test by passing a **string** to the function.

Create a test for an empty stack

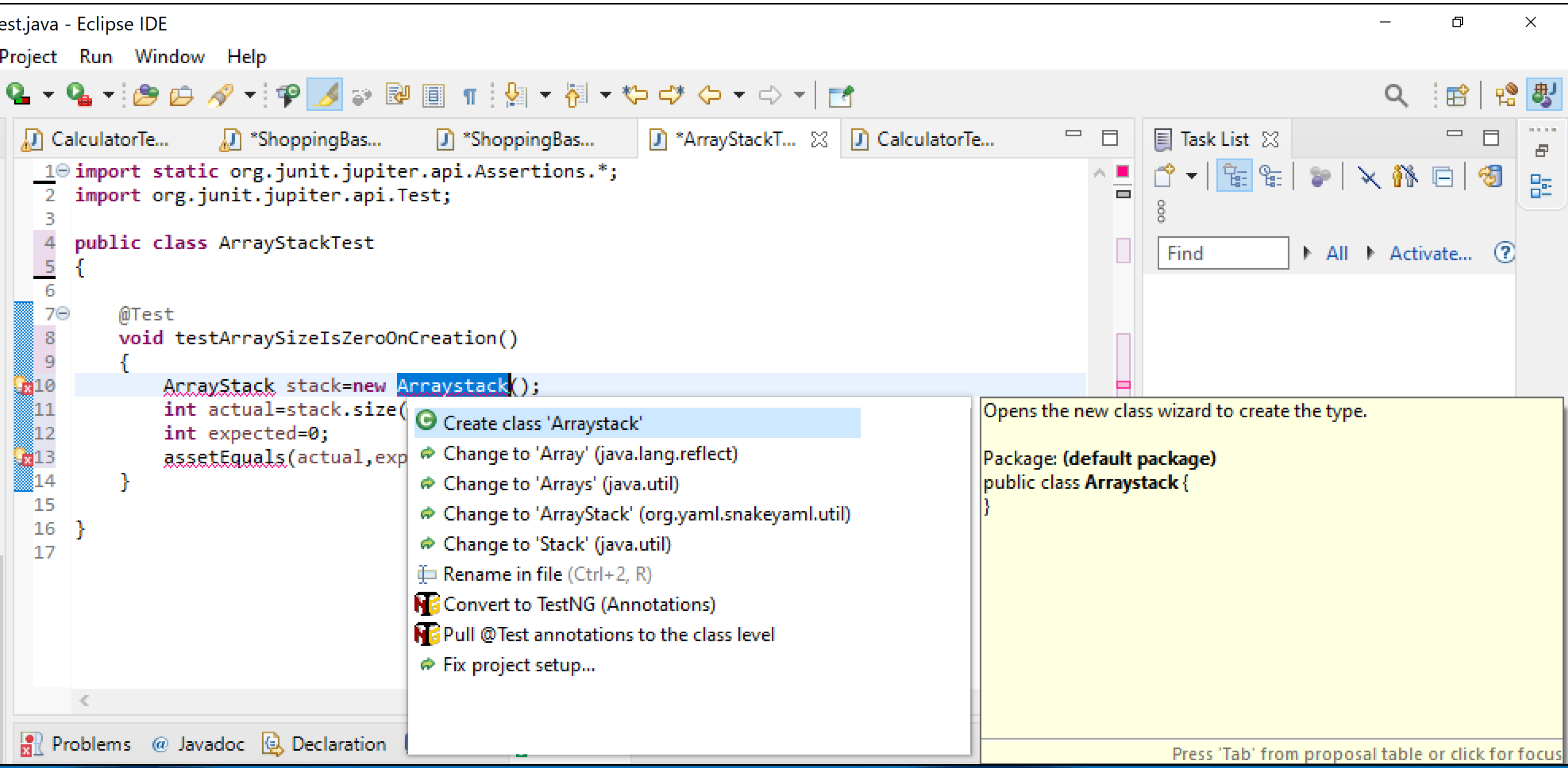


```
public class ArrayStackTest
{
@Test
Public void testArraySizeIsZeroOnCreation()
{
ArrayStack stack=new Arraystack();/*there is no class
called ArrayStack right now*/
int expected=stack.size(); /*there is no method
called size() right now*/
int actual=0;
assertEquals(expected,actual);
}}
```

Run the test. It will fail without any implemented code.

University of Victoria

The IDE prompts you to create the ArrayStack class



```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class ArrayStackTest
{

    @Test
    void testArraySizeIsZeroOnCreation()
    {
        ArrayStack stack=new Arraystack();
        int actual=stack.size(
        int expected=0;
        assetEquals(actual,exp
    }

}
```

Create class 'Arraystack'
Change to 'Array' (java.lang.reflect)
Change to 'Arrays' (java.util)
Change to 'ArrayStack' (org.yaml.snakeyaml.util)
Change to 'Stack' (java.util)
Rename in file (Ctrl+2, R)
Convert to TestNG (Annotations)
Pull @Test annotations to the class level
Fix project setup...

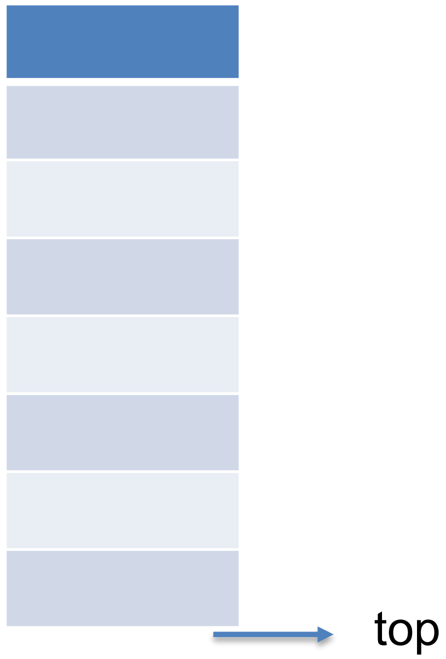
Opens the new class wizard to create the type.

Package: **(default package)**
public class **Arraystack** {
}

Press 'Tab' from proposal table or click for focus

ArrayStack.java

```
public class ArrayStack
{
int top=0;
public int size()
{
return top;
}


}
```

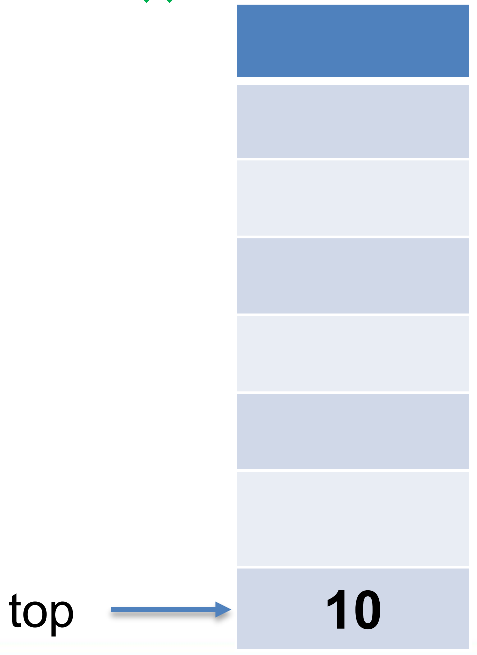


- TDD is not about taking teeny-tiny steps, it's about being able to take teeny-tiny steps.
- Would I code day-to-day with steps this small?
- No. But when things get the least bit weird, I'm glad I can.

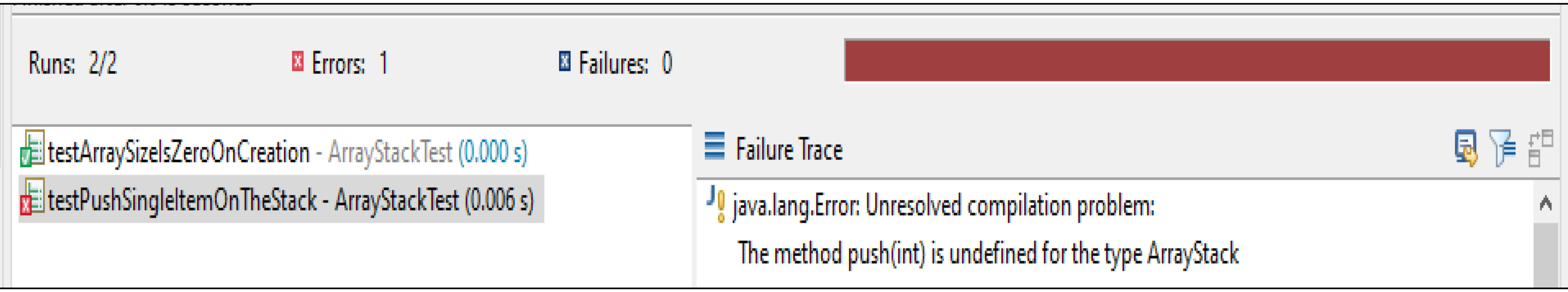
University of Victoria

Now push an integer element 10 in the stack. The size should now become 1.

```
@Test
public void testPushSingleItemOnTheStack()
{
ArrayStack stack=new ArrayStack();
stack.push(10); /* There is no push() function right
now so the test fails*/
int actual=1;
int expected=stack.size();
assertEquals(expected,actual);
}
```



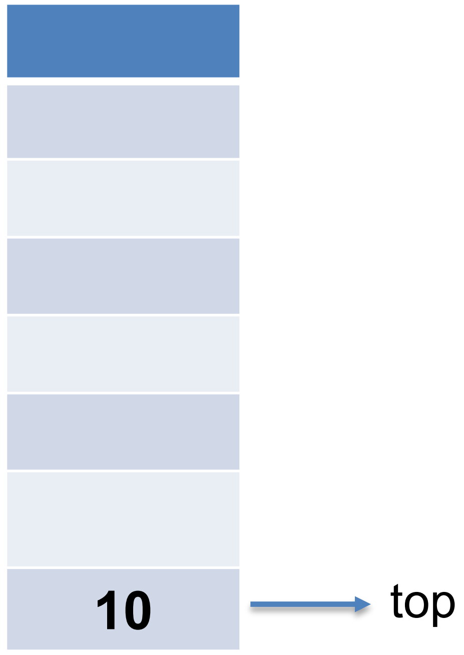
The test fails



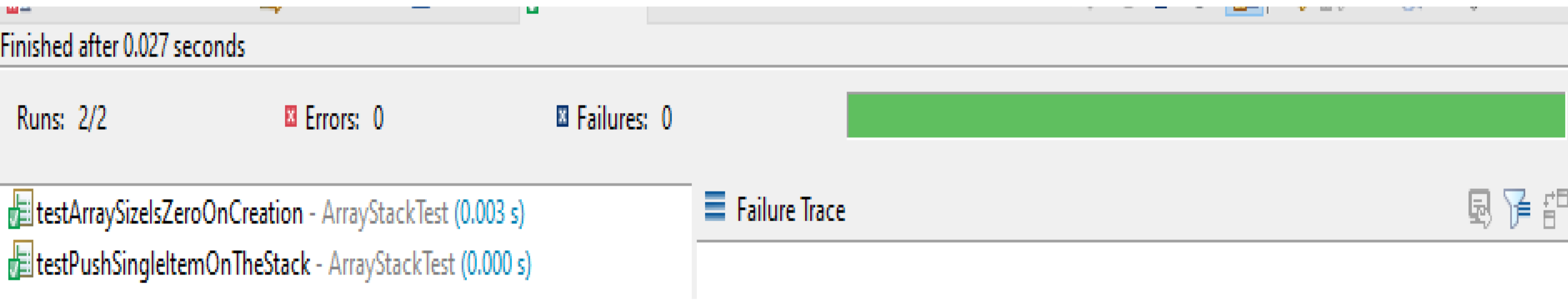
```
Runs: 2/2          Errors: 1          Failures: 0

testArraySizeIsZeroOnCreation - ArrayStackTest (0.000 s)
testPushSingleItemOnTheStack - ArrayStackTest (0.006 s)

Failure Trace
java.lang.Error: Unresolved compilation problem:
  The method push(int) is undefined for the type ArrayStack
```

ArrayStack class

```
public class ArrayStack {
int[] itemsArray;
int top=0;
public ArrayStack()
{
itemsArray=new int[8];
}
public int size()
{
return top;
}
public void push(int item)
{
itemsArray[top]=item;
top++;
}}
```



Test passes



```
Finished after 0.027 seconds

Runs: 2/2          Errors: 0          Failures: 0

testArraySizeIsZeroOnCreation - ArrayStackTest (0.003 s)
testPushSingleItemOnTheStack - ArrayStackTest (0.000 s)
```

Let’s refactor the code now. We see new ArrayStack object created for each test duplicate code.

```
public class ArrayStackTest {
ArrayStack stack;
@Before
public void setupBeforeEachTestCase(){
stack=new ArrayStack();}
@Test
public void testArraySizeIsZeroOnCreation() {
int actual=0;
int expected=stack.size();
assertEquals(expected,actual);}

@Test
public void testPushSingleItemOnTheStack()
{
stack.push(10);
int actual=1;
int expected=stack.size();
assertEquals(expected,actual);}}
```

University of Victoria

Another refactoring- make the stack generic and now push a string ‘a’ to check whether everything works fine after refactoring

```
public class ArrayStack<T> {
T[] itemsArray;
int top=0;
public ArrayStack()
{
itemsArray=(T[])new Object[8];
}
public int size()
{
return top;
}
public void push(T item)
{
itemsArray[top]=item;
top++;
}}
```

University of Victoria

ArrayStackTest- the test passes after refactoring

```
public class ArrayStackTest {
ArrayStack <String> stack;
@Before
public void setupBeforeEachTestCase()
{
stack=new ArrayStack<String>();
}
@Test
public void testArraySizeIsZeroOnCreation() {
int actual=0;
int expected=stack.size();
assertEquals(expected,actual);
}
@Test
public void testPushSingleItemOnTheStack()
{
stack.push("a");
int actual=1;
int expected=stack.size();
assertEquals(expected,actual);
}}
```

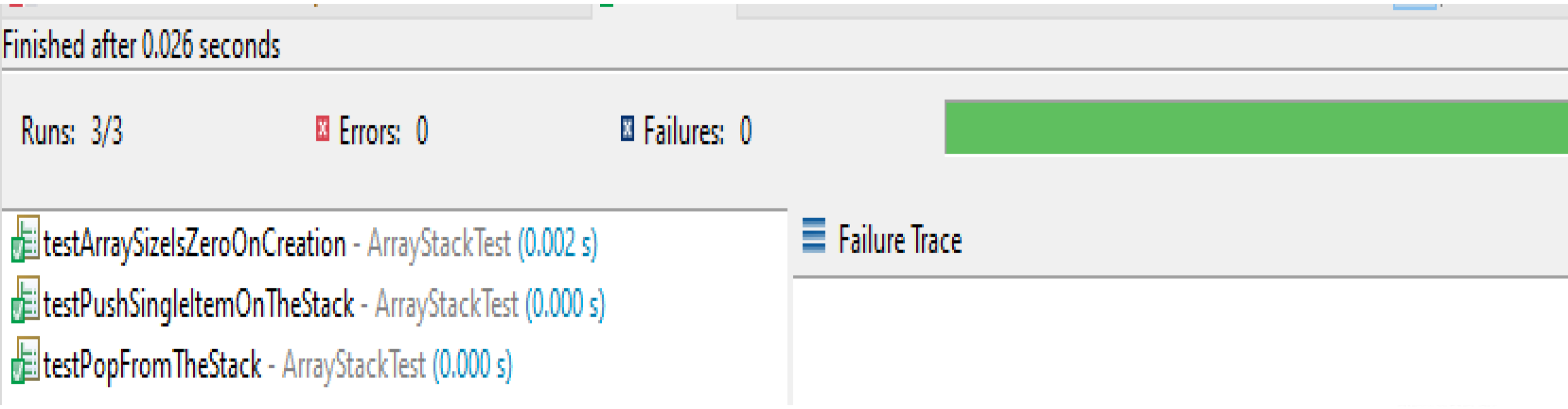
Create a test case for pop() which removes the top element from the stack

```
@Test
public void testPopFromTheStack()
{
int actual=0;
int expected=stack.size();
assertEquals(expected,actual);
}
```

Create a method pop() which removes the top element from the stack

```
public T pop()
{
T x=itemsArray[top-1];
top--;
return x;
}
```

Before creation of the method the test fails then it passes after creation.



```
Finished after 0.026 seconds

Runs: 3/3        Errors: 0        Failures: 0

testArraySizeIsZeroOnCreation - ArrayStackTest (0.002 s)
testPushSingleItemOnTheStack - ArrayStackTest (0.000 s)
testPopFromTheStack - ArrayStackTest (0.000 s)
```

Write other test cases for stack (15M)

There are other cases that you should test:

1. Before executing pop(), check for empty stack. (5M)
2. Set a maximum size for the stack and you should not be able to push an element onto the stack if the stack is full. (5M)
3. Test a complete stack implementation. (10M)

Tennis game-scoring application

1. Each player can have either of these points in one game 0 15 30 40
2. If you have 40 and you win the ball you win the game, however there are special rules.
3. If both have 40 the players are deuce.
4. If the game is in deuce, the winner of a ball will have advantage and game ball.
5. If the player with advantage wins the ball, he wins the game
6. If the player without advantage wins, they are back at deuce.
7. A game is won by the first player to have won at least four points in total and at least two points more than the opponent.
8. The running score of each game is described in a manner peculiar to tennis: scores from zero to three points are described as “love”, “fifteen”, “thirty”, and “forty” respectively.
9. If at least three points have been scored by each player, and the scores are equal, the score is “deuce”.
10. If at least three points have been scored by each side and a player has one more point than his opponent, the score of the game is “advantage” for the player in the lead.

Test cases

playerOneScore	playerTwoScore	expectedPrettyScore
0	0	"Love - All"
1	1	"Fifteen - All"
2	2	"Thirty - All"
3	3	"Deuce"
4	4	"Deuce"
1	0	"Fifteen - Love"
0	1	"Love - Fifteen"
2	0	"Thirty - Love"
0	2	"Love - Thirty"
3	0	"Forty - Love"
0	3	"Love - Forty"
4	0	"Win for Joan"
0	4	"Win for Finner"
2	1	"Thirty - Fifteen"
1	2	"Fifteen - Thirty"
3	1	"Forty - Fifteen"
1	3	"Fifteen - Forty"
4	1	"Win for Joan"
1	4	"Win for Finner"
3	2	"Forty - Thirty"
2	3	"Thirty - Forty"
4	2	"Win for Joan"
2	4	"Win for Finner"
4	3	"Advantage Joan"
3	4	"Advantage Finner"
5	4	"Advantage Joan"
4	5	"Advantage Finner"
6	4	"Win for Joan"
4	6	"Win for Finner"

University of Victoria

Tennis game-scoring application

- Create given, when, then acceptance scenarios for the application using these test cases.
- Use the TDD approach to develop this application.
- Show all steps.

City search

Implement a city search functionality using TDD. The function takes a string (search text) as input and returns the found cities which corresponds to the search text.

Prerequisites

Create a collection of strings that will act as a database for the city names.

City names: Paris, Budapest, Skopje, Rotterdam, Valencia, Vancouver, Amsterdam, Vienna, Sydney, New York City, London, Bangkok, Hong Kong, Dubai, Rome, Istanbul

Requirements

1. If the search text is fewer than 2 characters, then should return no results. (It is an optimization feature of the search functionality.)
2. If the search text is equal to or more than 2 characters, then it should return all the city names starting with the exact search text. For example, for search text "Va", the function should return Valencia and Vancouver
3. The search functionality should be case insensitive
4. The search functionality should work also when the search text is just a part of a city name. For example, "ape" should return "Budapest" city
5. If the search text is a "*" (asterisk), then it should return all the city names.

University of Victoria

Practice TDD (30M)

- Create a single link list using the TDD approach. Create functions for insertion, deletion and checking size of the list. (10M)
- Divide the input domain in equivalence partitions to create a testing domain. (5M)
- Create edge test cases using boundary value analysis. (5M)
- Create and execute both valid and invalid test cases.(10)

University of Victoria