REFACTORING

What is refactoring?

- Improve the quality of code
  - Simple
  - Flexible
  - Easy to understand
- Does not add new functionality

"If it stinks, change it" -- Grandma Beck

Why refactoring?

- It is very difficult to design and write code correct the first time
  - Requirements change
  - Technology changes
  - A good design changes over time when the application is expanded

"Grow, don't build software" --Fred Brooks

How does refactoring work?

- Small steps
- Test after every step
  - Unit tests are a must
- Refactor tooling
- Refactor often

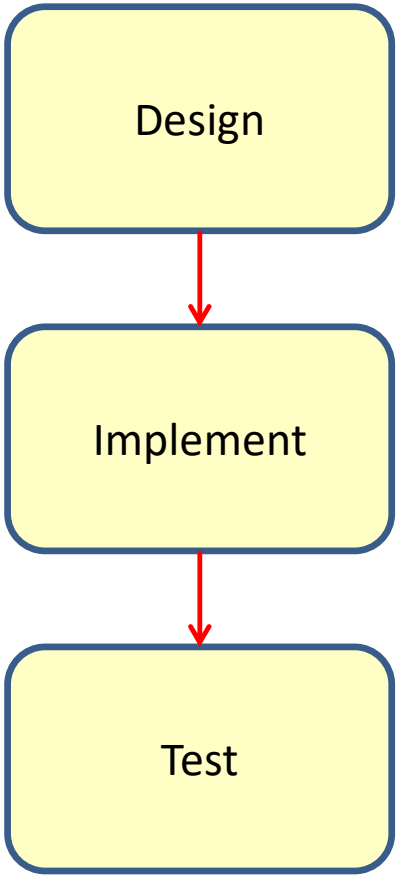
Refactoring helps you writing better code in a faster way

TEST DRIVEN DEVELOPMENT

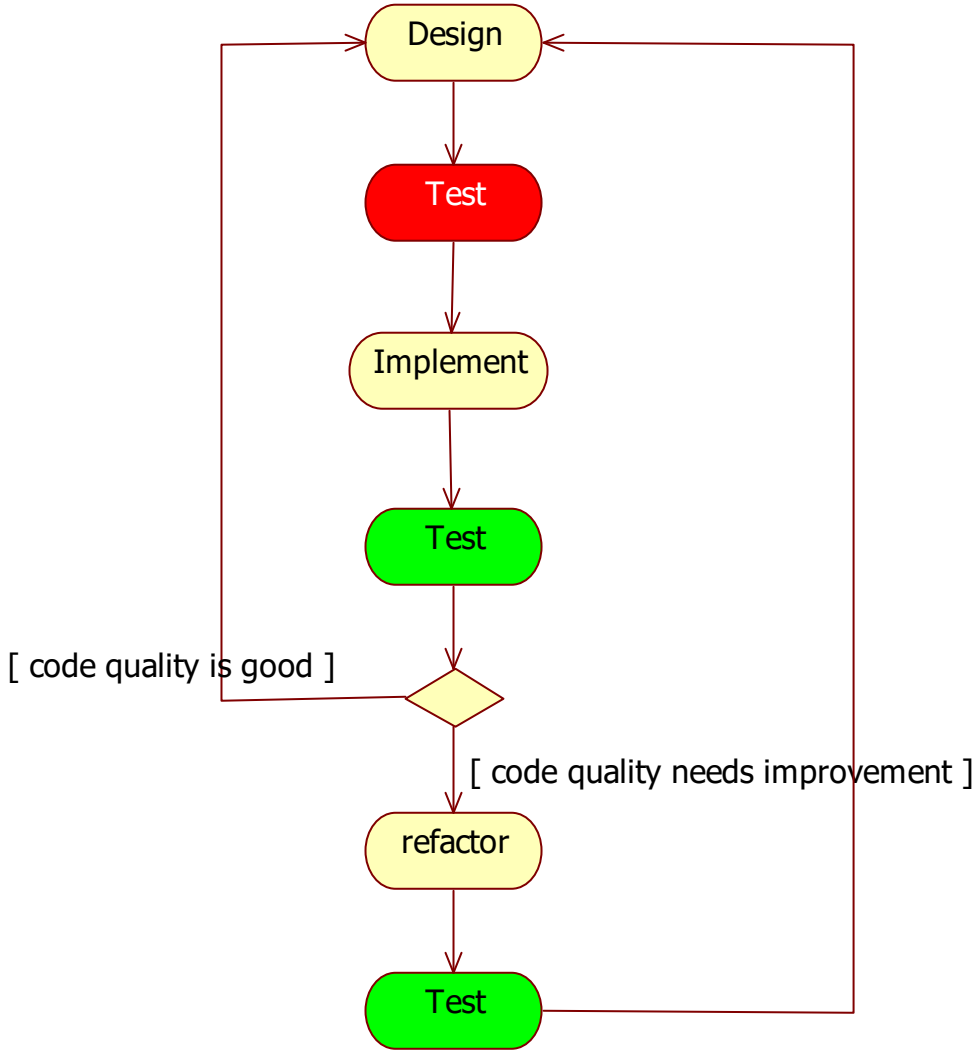
What is TDD?

- Is a Test-First approach
  - Write the test-code first and then write the dev-code

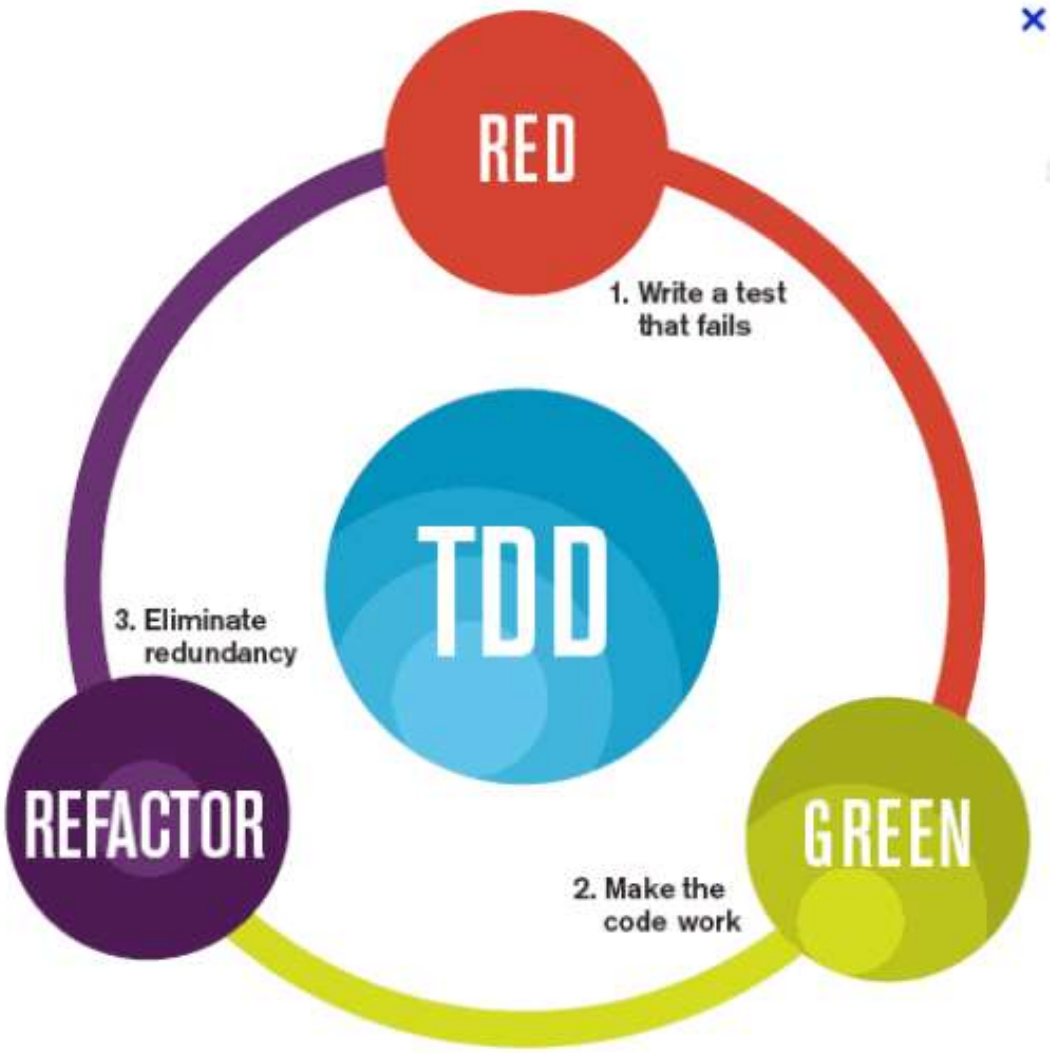
The old way



TDD



Red, green, refactor



Why TDD?

- It leads to think about ‘How to use a component’ first and then about ‘How to implement’.
- As much about design technique as testing technique.
- As much about (executable) documentation as testing.

TEST DRIVEN DEVELOPMENT IN ACTION

Test list for a Stack

- Create a *Stack* and verify that *IsEmpty* is true.
- *Push* a single object on the *Stack* and verify that *IsEmpty* is false.
- *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.
- *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.
- *Push* three objects, remembering what they are; *Pop* each one, and verify that they are removed in the correct order.
- *Pop* a *Stack* that has no elements.
- *Push* a single object and then call *Top*. Verify that *IsEmpty* is false.
- *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is the same as the one that was pushed.
- Call *Top* on a *Stack* with no elements.

1. Create a *Stack* and verify that *IsEmpty* is true

- Write the test first

```
package test;

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

public class StackTest {

  @Test
  public void testEmpty() {
    Stack stack = new Stack();
    assertTrue(stack.isEmpty());
  }
}
```

1. Create a *Stack* and verify that *IsEmpty* is true

- Write the Stack class
- Implement the smallest amount of work that needs to be done

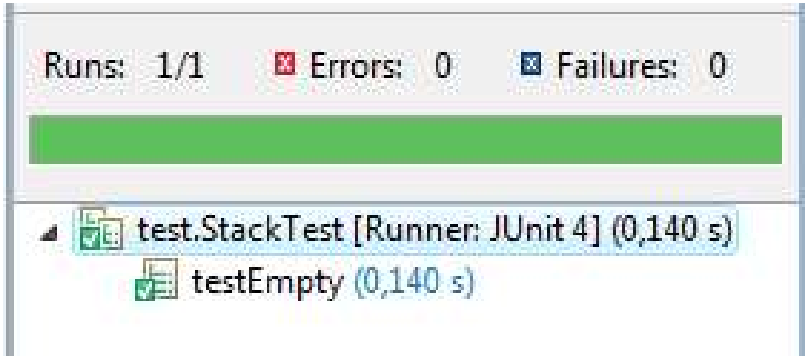
```
package domain;

public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }
}
```

1. Create a *Stack* and verify that *IsEmpty* is true

- Import the Stack in the test and run the test

```
package test;

import static org.junit.Assert.*;
import org.junit.Test;
import domain.Stack;

public class StackTest {

  @Test
  public void testEmpty() {
    Stack stack = new Stack();
    assertTrue(stack.isEmpty());
  }
}
```

```
package domain;

public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }
}
```



2. *Push* a single object on the *Stack* and verify that *IsEmpty* is false.

- Write the test first

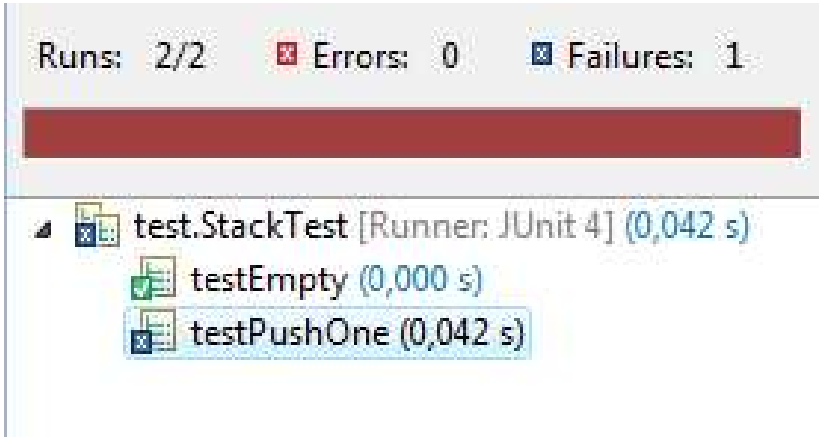
```
@Test
public void testPushOne() {
  Stack stack = new Stack();
  stack.push("first element");
  assertFalse("After push, isEmpty should be false",stack.isEmpty());
}
```

2. *Push* a single object on the *Stack* and verify that *IsEmpty* is false.

- Get it to compile

```
public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }
  public void push (Object object){
  }
}
```

- Run the test
  - It should fail



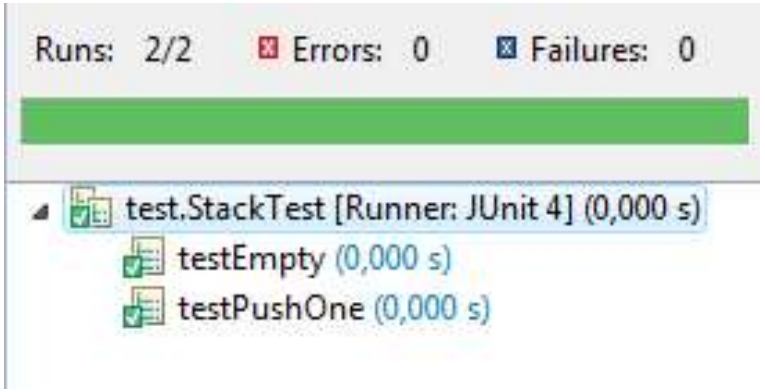
2. *Push* a single object on the *Stack* and verify that *IsEmpty* is false.

- Do the smallest amount of work to succeed

```
public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }
  public void push (Object object){
    empty=false;
  }
}
```

We know we have to store the elements in a collection, but we have no tests for this, so we will wait with this

- Run the test
  - It should succeed



Refactor the test code

- The test code is just as important as the production code

```
public class StackTest {
  private Stack stack=null;

  @Before
  public void Init(){
    stack = new Stack();
  }

  @Test
  public void testEmpty() {
    assertTrue(stack.isEmpty());
  }

  @Test
  public void testPushOne() {
    stack.push("first element");
    assertFalse("After push, isEmpty should be false",stack.isEmpty());
  }
}
```

Eliminate duplication

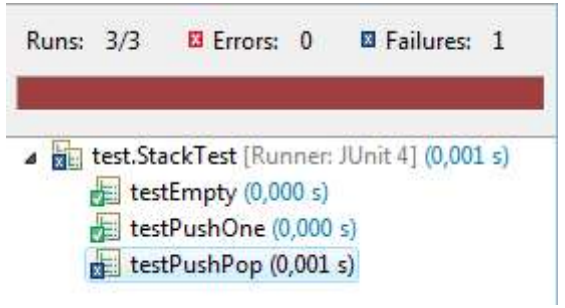
3. *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.

- Write the test first

```
@Test
public void testPushPop() {
  stack.push("first element");
  stack.pop();
  assertTrue("After push-pop, isEmpty should be true",stack.isEmpty());
}
```

- Make it compile and fail the test

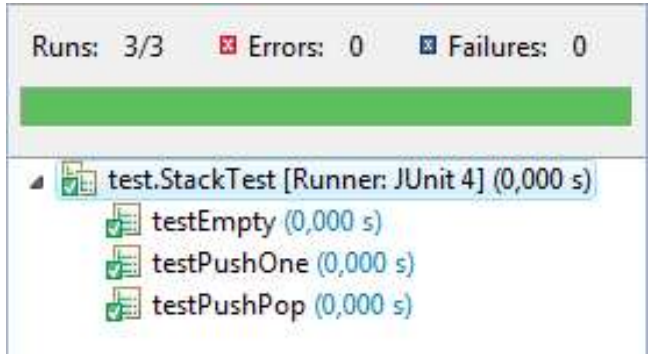
```
public void pop (){

}
```



3. *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true.

- Make the test succeed

```
public class Stack {
  private boolean empty = true;

  public boolean isEmpty() {
    return empty;
  }

  public void push (Object object){
    empty=false;
  }

  public void pop (){
    empty=true;
  }
}
```



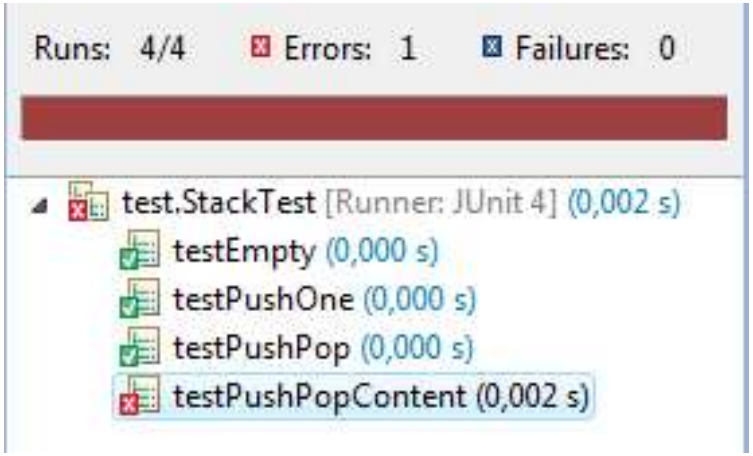
4. *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.

- Write the test first

```
@Test
public void testPushPopContent() {
  int expected = 1234;
  stack.push(expected);
  int actual = (Integer) stack.pop();
  assertEquals(expected, actual);
}
```

- Make it compile and fail the test

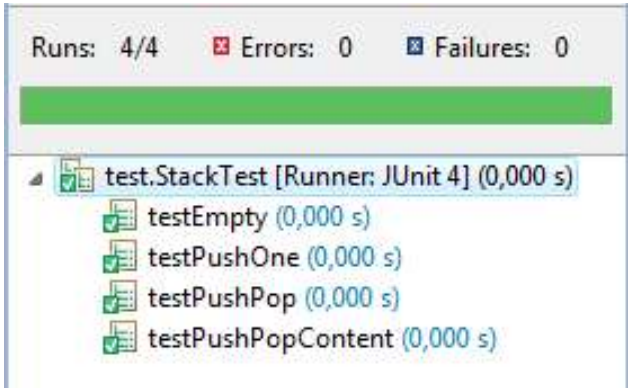
```
public Object pop (){
  empty=true;
  return null;
}
```



4. *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal.

- Make the test succeed

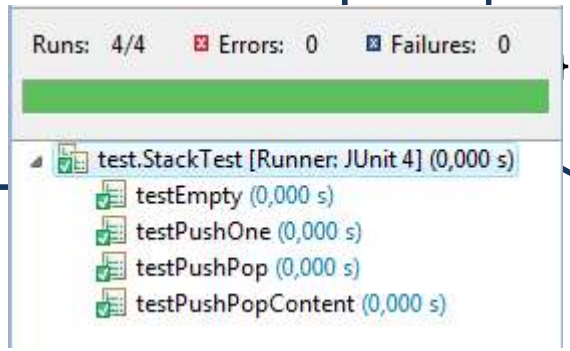
```
public class Stack {
  private boolean empty = true;
  private Object element = null;

  public boolean isEmpty() {
    return empty;
  }

  public void push (Object object){
    empty=false;
    element =object;
  }

  public Object pop (){
    empty=true;
    Object top = element;
    element=null;

    return top;
  }
}
```



Refactor the Stack

```
public class Stack {
  private boolean empty = true;
  private Object element = null;

  public boolean isEmpty() {
    return empty;
  }

  public void push (Object object){
    empty=false;
    element =object;
  }

  public Object pop (){
    empty=true;
    Object top = element;
    element=null;

    return top;
  }
}
```

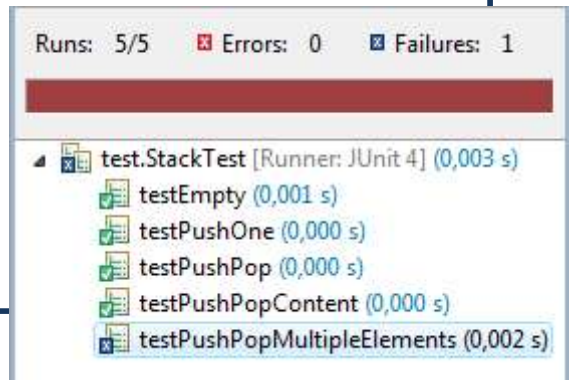
```
public class Stack {
  private Object element = null;

  public boolean isEmpty() {
    return (element == null);
  }

  public void push (Object object){
    element =object;
  }

  public Object pop (){
    Object top = element;
    element=null;

    return top;
  }
```



5. *Push* three objects, remembering what they are; *Pop* each one, and verify that they are correct.

- Write the test first and run the test

```
@Test
public void testPushPopMultipleElements() {
  String element1="1";
  String element2="2";
  String element3="3";
  stack.push(element1);
  stack.push(element2);
  stack.push(element3);

  String popped = (String) stack.pop();
  assertEquals(element3, popped);
  popped = (String) stack.pop();
  assertEquals(element2, popped);
  popped = (String) stack.pop();
  assertEquals(element1, popped);
}
```

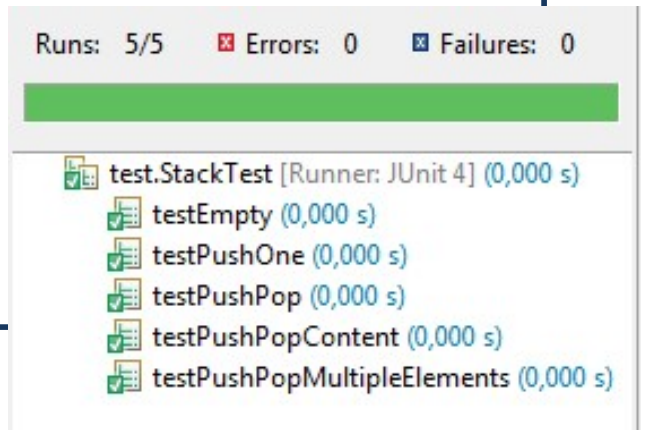


5. *Push* three objects, remembering what they are; *Pop* each one, and verify that they are correct.

- Make it compile and test successfully

```
public class Stack {
  private List<Object> elements = new LinkedList<Object>();

  public boolean isEmpty() {
    return (elements.size() == 0);
  }

  public void push (Object object){
    elements.add(0,object);
  }

  public Object pop (){
    Object top = elements.get(0);
    elements.remove(0);
    return top;
  }
}
```

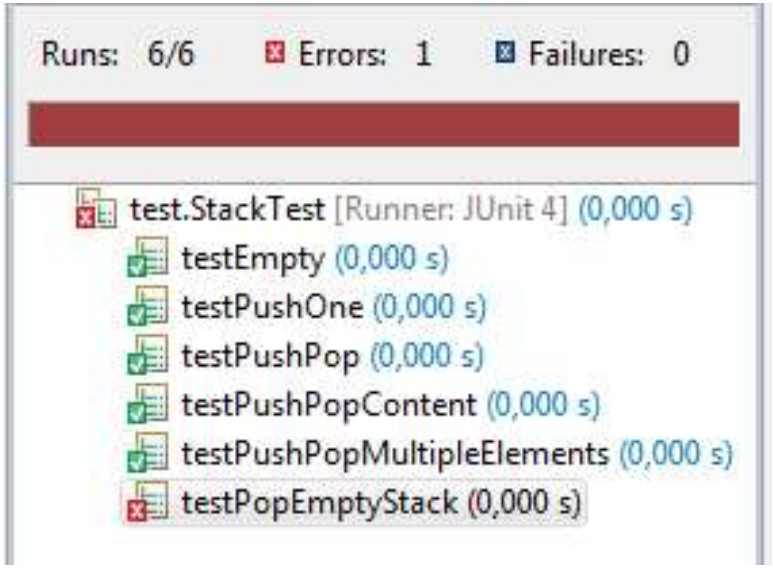
Big changes, and all previous tests are still running



6. *Pop* a *Stack* that has no elements.

- Write the test first and run the test

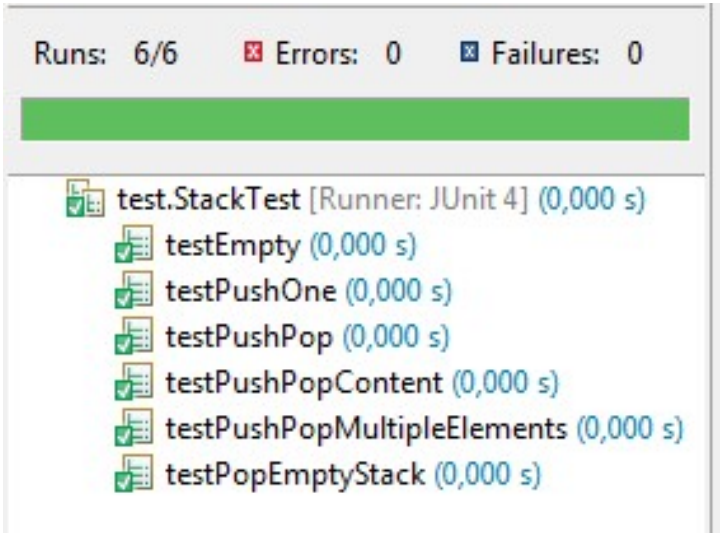
```
@Test(expected=IllegalStateException.class)
    public void testPopEmptyStack(){
        stack.pop();
}
```



6. *Pop* a *Stack* that has no elements.

- Make it compile and test successfully

```
public Object pop (){
  if (isEmpty())
    throw new IllegalStateException("cannot pop an empty stack");
  Object top = elements.get(0);
  elements.remove(0);
  return top;
}
```



New tests

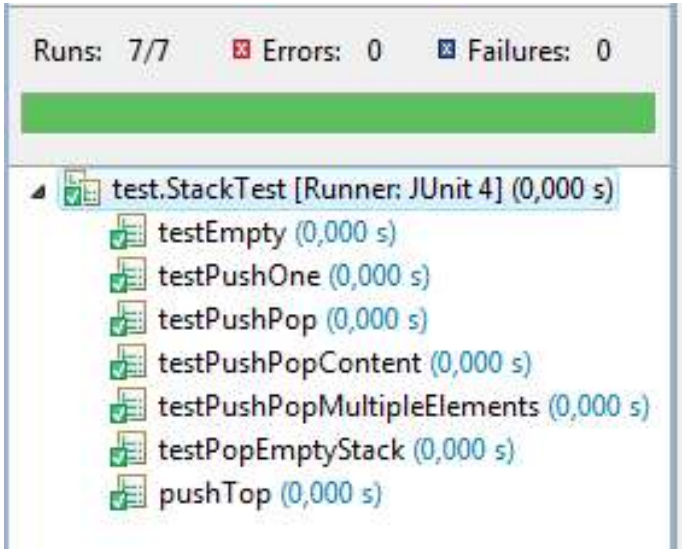
- As we were implementing this test, a few additional tests came to mind
  - *Push null* onto the *Stack* and verify that *IsEmpty* returns false.
  - *Push null* onto the *Stack*, *Pop* the *Stack*, and verify that the value returned is *null*.
  - *Push null* onto the *Stack*, call *Top*, and verify that the value returned is *null*.

7. *Push* a single object and then call *Top*. Verify that *IsEmpty* returns false.

- Write the test first and run the test

```
@Test
public void pushTop() {
  stack.push("83");
  stack.top();
  assertFalse(stack.isEmpty());
}
```

```
public Object top (){
  return null;
}
```



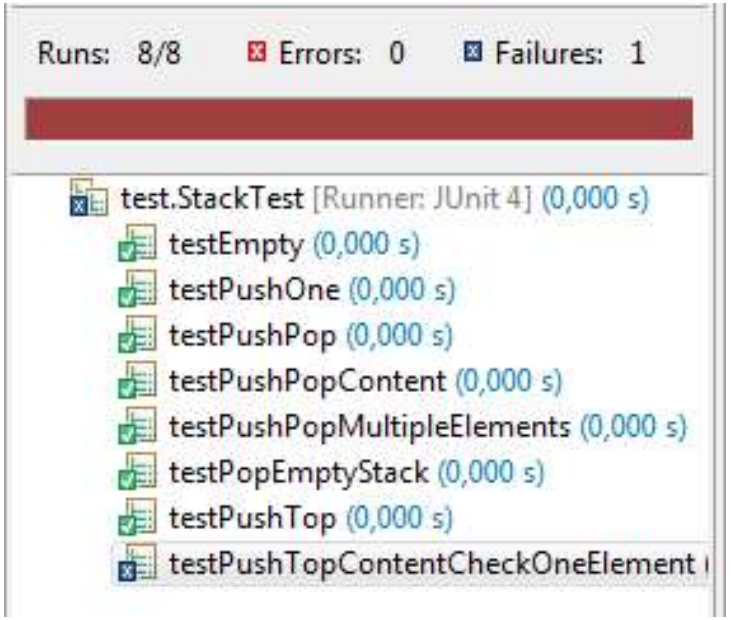
New tests

- As we were implementing this test, a few additional tests came to mind
  - *Push* multiple items onto the *Stack* and verify that calling *Top* returns the correct object.
  - *Push* an item on the *Stack*, call *Top* repeatedly, and verify that the object returned each time is equal to the object that was pushed onto the *Stack*.

8. *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is equal to the one that was pushed.

- Write the test first and run the test

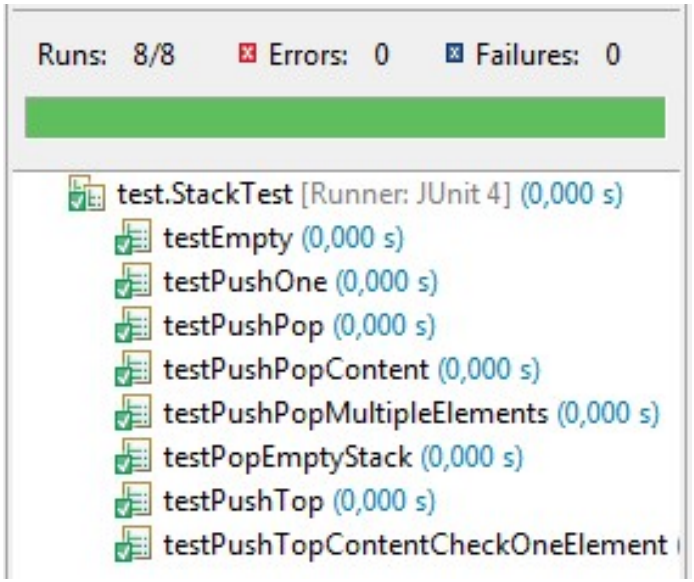
```
@Test
public void
testPushTopContentCheckOneElement() {
  String pushed ="96";
  stack.push(pushed);
  String topped = (String)stack.top();
  assertEquals(pushed, topped);
}
```



8. *Push* a single object, remembering what it is; and then call *Top*. Verify that the object that is returned is equal to the one that was pushed.

- Make it compile and test successfully

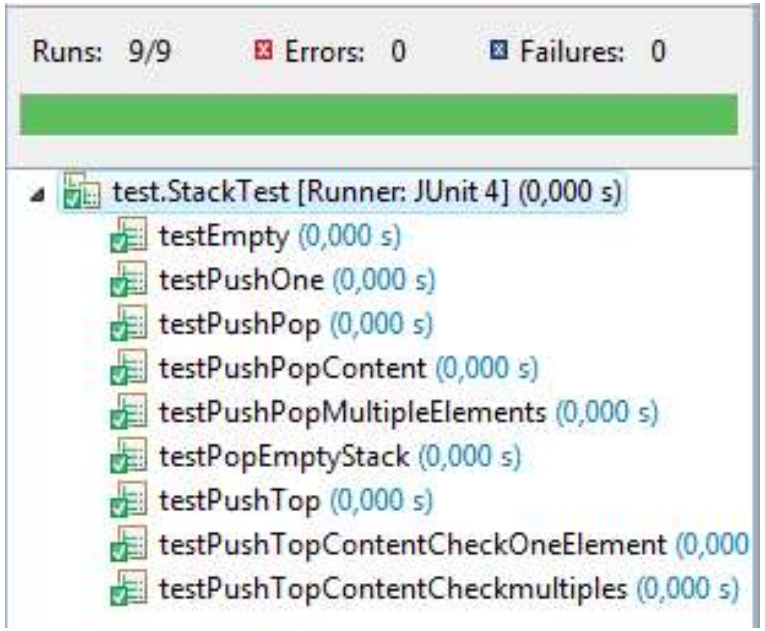
```
public Object top (){
  return elements.get(0);
}
```



9. *Push* multiple objects, remembering what they are; call *Top*, and verify that the last item pushed is equal to the one returned by *Top*.

- Write the test first and run the test

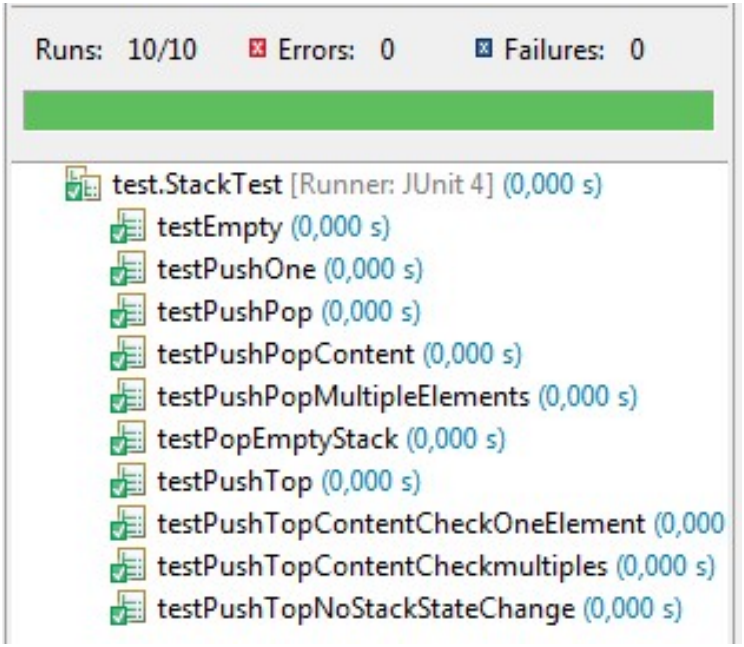
```
 @Test
public void
testPushTopContentCheckmultiples() {
  String pushed1 ="1";
  String pushed2 ="2";
  String pushed3 ="3";

  stack.push(pushed1);
  stack.push(pushed2);
  stack.push(pushed3);

  String topped = (String)stack.top();
  assertEquals(pushed3, topped);
}
```



10. *Push* one object and call *Top* repeatedly, comparing what is returned to what was pushed.

- Write the test first and run the test

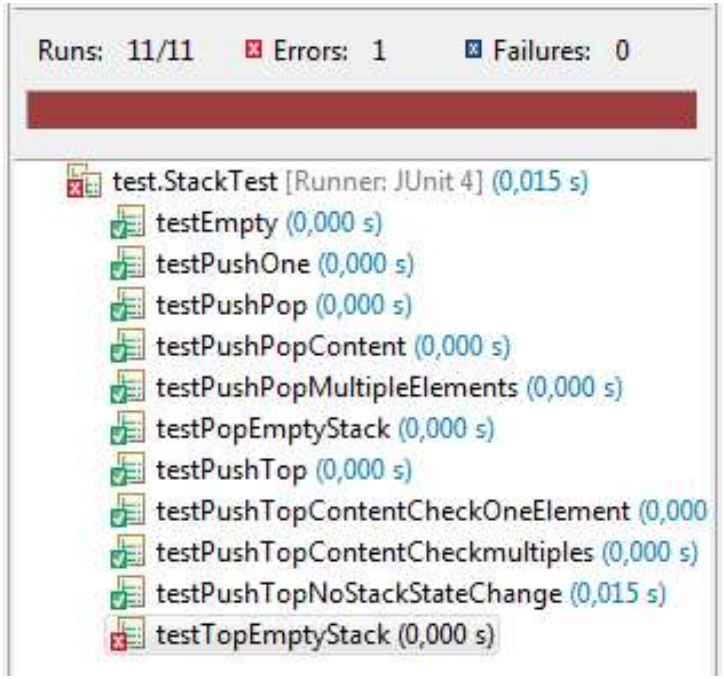
```
@Test
public void testPushTopNoStackStateChange() {
String pushed ="45";
    stack.push(pushed);

    for (int x=0; x<5; x++){
    String topped = (String)stack.top();
    assertEquals(pushed, topped);
    }
}
```



11. Call *Top* on a *Stack* that has no elements.

- Write the test first and run the test

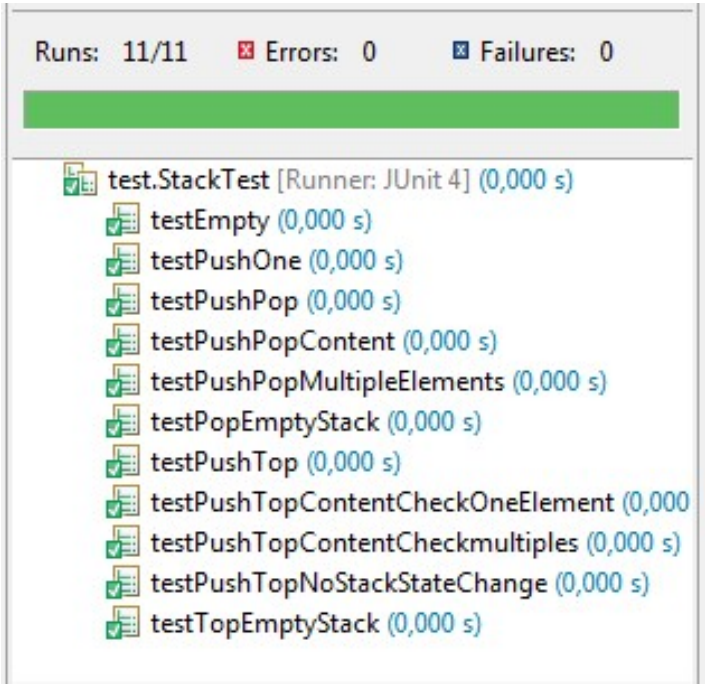
```
@Test(expected=IllegalStateException.class)
public void testTopEmptyStack(){
  stack.top();
}
```



11. Call *Top* on a *Stack* that has no elements.

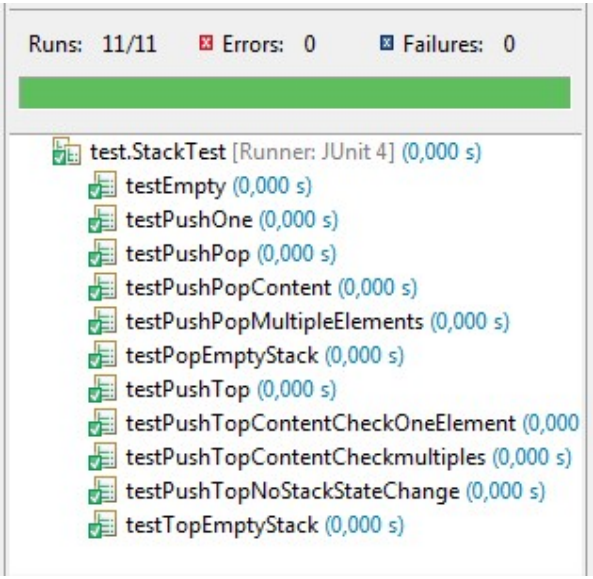
- Make the test succeed

```
public Object top (){
  if (isEmpty())
    throw new IllegalStateException("cannot top an empty stack");
  return elements.get(0);
}
```



Refactor

```
public Object pop (){
  Object top = top();
  elements.remove(0);
  return top;
}

public Object top (){
  if (isEmpty())
    throw new IllegalStateException("stack is empty");
  return elements.get(0);
}
```



Last 3 tests

- 12: *Push null* onto the *Stack* and verify that *IsEmpty* is false

```
@Test
public void testPushNull() {
    stack.push(null);
    assertFalse(stack.isEmpty());
}
```

- 13: *Push null* onto the *Stack*, *Pop* the *Stack*, and verify that the value returned is *null*

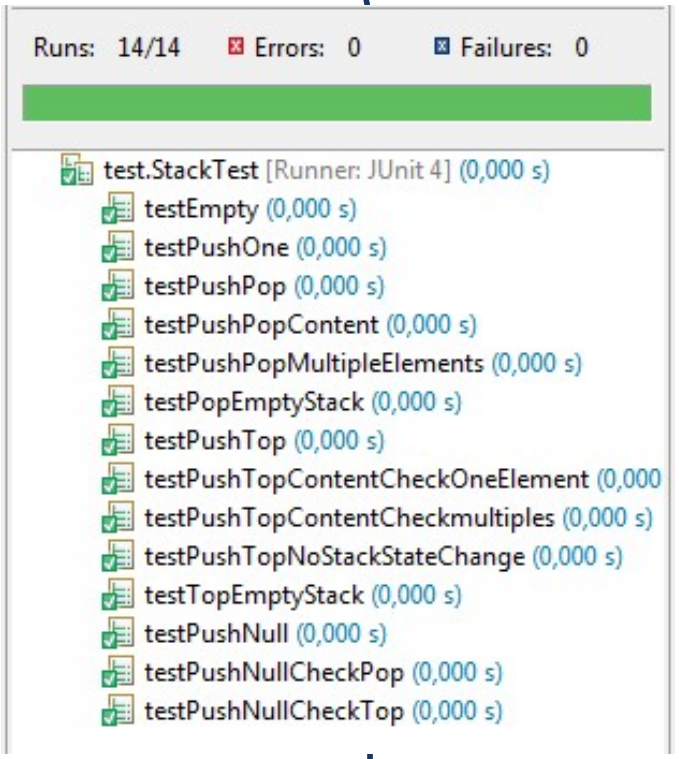
```
@Test
public void testPushNullCheckPop() {
    stack.push(null);
    assertNull(stack.pop());
    assertTrue(stack.isEmpty());
}
```

- Push *null* onto the *Stack*, call *Top*, and verify that the value returned is *null*

```
@Test
public void testPushNullCheckTop() {
    stack.push(null);
    assertNull(stack.top());
    assertFalse(stack.isEmpty());
}
```

Summary

```
public class Stack {
  private List<Object> elements = new LinkedList<Object>();

  public boolean isEmpty() {
    return (elements.size() == 0);
  }

  public void push (Object object){
    elements.add(0,object);
  }

  public Object pop (){
    Object top = top();
    elements.remove(0);
    return top;
  }

  public Object top (){
    if (isEmpty())
      throw new IllegalStateException("stack is empty");
    return elements.get(0);
  }
}
```



Summary

- There is more test code than actual code
- Most time went in writing tests
- Our focus was on how the stack is used
  - Instead of how it is implemented
- The stack implementation is clear and concise
  - Due to refactoring

TDD Best practices

- The name of the test should describe a feature or specification
  - The name should clearly describe the purpose of the test
- Each test should be for a single concept
- Strive for one assertion per test
  - Or as few as possible
  - More than one may indicate more than one concept is being tested

TDD Best practices

- Start with the requirements
- Even seemingly trivial tests are important
- Treat your test code like production code
  - Maintain
  - Refactor
- Just making it work isn’t good enough
- Tests should never depend on each other

TDD challenges

- Discipline is required
- Developers are often stubborn and lazy
  - I have no time for testing
  - QA will do the testing
    - Only blackbox!
- It is hard for developers to drop bad habits
- Management doesn't often understand "Internal Quality"

Main point

- Testing is an important aspect of software development that increases the quality of the code.
- The daily connection with our divine source increases all important aspects of life.