JUnit tests

Object Oriented Programming

http://softeng.polito.it/courses/09CBI

SoftEng
http://softeng.polito.it

Version 3.1.0 - April 2018

JUnit

- JUnit is a testing framework for Java programs
  - Written by Kent Beck and Erich Gamma
- It is a framework with unit-testing functionalities
- Integrated in Eclipse development Environment

http://www.junit.org

Unit Testing

- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

JUnit Framework

- JUnit helps the programmer:
  - Define and execute tests and test suites
  - Formalize requirements and clarify architecture
  - Write and debug code
  - Integrate code and always be ready to release a working version

History

- 1997 on the plane to OOPSLA97 Kent Beck and Erich Gamma wrote JUnit
- Junit.org – August 2000
- Junit 3.8.1 – September 2002
- Junit 4.0 – February 2006
  - Latest release: 4.12 – Dec 2012
- Junit 5.0 – September 2017

What JUnit does

- For each test (method) JUnit
  - calls pre-test fixture
    - Intended to acquire resources and create any objects that may be needed for testing
  - calls the test method
  - calls post-test fixtures
    - Intended to release resources and remove any objects you created

Test method

- A test method doesn't return a result
- The test method performs operations and checks the results
- Checks are performed using a set of **`assert*()`** method
- The JUnit framework detects the anomalies and deals with them

assert*() methods

```
assertTrue(boolean test)
assertFalse(boolean test)
assertEquals(expected, actual)
assertSame(Object expected,
           Object actual)
assertNotSame(Object expected,
              Object actual)
assertNull(Object object)
```


assert*() methods

- For a condition

```
assertTrue(condition)
```

  - If the tested condition is
    - **true** => proceed with execution
    - **false** => abort the test method execution, prints out the optional message

assert*() methods

```
assertNotNull(Object object)
```

```
fail()
```

- All the above may take an optional String message as the first argument, e.g.

```
static void assertTrue(
        String message,
        boolean test)
```


assert*()

- For objects, int, long, byte:

```
assertEquals( expected, actual)
```

  - Ex. `assertEquals( 2 , unoStack.size() );`

- For floating point values:

```
assertEquals(expected,actual,err)
```

  - Ex. `assertEquals(1.0, Math.cos(3.14), 0.01);`

JUnit 3

SYNTAX

Test a Stack

extends TestCase

```
public class StackTest extends TestCase {
  public void testStack() {
    Stack aStack = new Stack();
    assertTrue("Stack should be empty!",
        aStack.isEmpty());
    aStack.push(10);
    assertTrue("Stack should not be empty!",
                !aStack.isEmpty());
    aStack.push(-4);
    assertEquals(-4, aStack.pop());
    assertEquals(10, aStack.pop());
  }
}
```

Test method name: testSomething

One or more assertions to check results

Test a Stack

```
public void testStackEmpty() {
  Stack aStack = new Stack();
  assertTrue("Stack should be empty!",
              aStack.isEmpty());
  aStack.push(10);
  assertFalse("Stack should not be empty!",
               aStack.isEmpty());
}
public void testStackOperations() {
  Stack aStack = new Stack();
  aStack.push(10);
  aStack.push(-4);
  assertEquals(-4, aStack.pop());
  assertEquals(10, aStack.pop());
}
```


Running a test case

- Running a test case
  - Executes all methods
    - **public**
    - Returning **void**
    - With no arguments
    - Name starting with "**test**"
  - Ignores the rest
- The class can contain helper methods
  - That are not public
  - Or not starting with "test"

Creating a test class

- Define a subclass of TestCase
- Override the **`setUp()`** method to initialize object(s) under test.
- Override the **`tearDown()`** method to release object(s) under test.
- Define one or more public **`testXXX()`** methods that exercise the object(s) under test and assert expected results.

Implementing setUp() method

- Override **`setUp()`** to initialize the variables, and objects
  - Implements a fixture
- Since **`setUp()`** is your code, you can modify it any way you like (such as creating new objects in it)
- Reduces the duplication of code

The tearDown() method

- In most cases, the `tearDown()` method doesn't need to do anything
  - The next time you run **`setUp()`**, your objects will be replaced, and the old objects will be available for garbage collection
  - Like the **`finally`** clause in a try-catch-finally statement, **`tearDown()`** is where you would release system resources (such as streams)

Test suites

- Allow running a group of related tests
- To do so, group your test methods in a class which extends `TestSuite`

TestSuite

- Combine many test cases in a test suite:

```
public class AllTests extends TestSuite {
public static TestSuite suite() {
  TestSuite suite = new TestSuite();
  suite.addTestSuite(StackTester.class);
  suite.addTestSuite(AnotherTester.class);
}
```


Example: Counter class

- For the sake of example, we will create and test a trivial “counter” class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value

Example: Counter class

- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)
- Don't be alarmed if, in this simple example, the JUnit tests are more code than the class itself

JUnit tests for Counter

```
public class CounterTest extends TestCase {
    Counter counter1;

    public void setUp() {
        // creates a (simple) test fixture
        counter1 = new Counter();
    }
    protected void tearDown() { }
        // no resources to release
```

JUnit tests for Counter…

```
    public void testIncrement() {
        assertTrue(counter1.increment()== 1);
        assertTrue(counter1.increment()== 2);
     }
    public void testDecrement() {
        assertTrue(counter1.decrement()==-1);
    }
}  // End from last slide
```


The Counter class itself

```
public class Counter {
 int count = 0;
 public int increment() {
     return ++count;
 }
 public int decrement() {
     return --count;
 }
        public int getCount() {
            return count;
        }  }
```

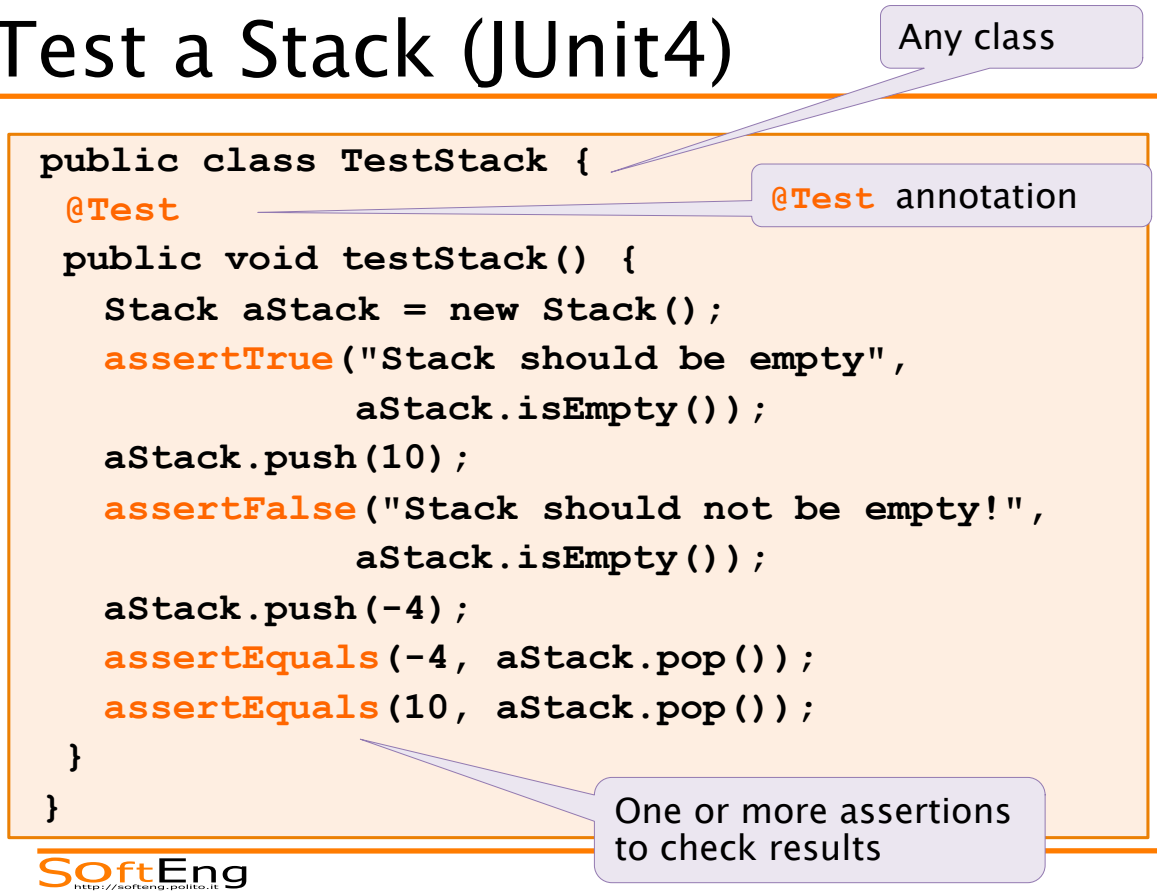
Junit 4

SYNTAX

JUnit 4

- Make use of java annotations
  - Less constraints on names
  - Easier to read/write
- Backward compatible with JUnit 3
- Assertions
  - `assert*()` methods
  - `assertThat()` method
    - To use the Hamcrest matchers

Test a Stack (JUnit4)




```
public class TestStack {
 @Test
 public void testStack() {
   Stack aStack = new Stack();
   assertTrue("Stack should be empty",
              aStack.isEmpty());
   aStack.push(10);
   assertFalse("Stack should not be empty!",
              aStack.isEmpty());
   aStack.push(-4);
   assertEquals(-4, aStack.pop());
   assertEquals(10, aStack.pop());
 }
}
```


Running a test case

- The JUnit runner
  - Executes all methods
    - Annotated with "`@Test`"
    - `public`
    - Returning `void`
    - With no arguments
  - Ignores the rest
- The class can contain helper methods provided they are not annotated
  - Not public

The pre-test fixture

- Annotate a method with `@Before` to make it a post-test fixture:
  - It is executed before each test method is run
  - It is the place to initialize attributes that will be used by tests
- There no limit to the setup you can do in a pre-test method: it is a general method
- It helps reducing duplication of code

The post-test fixture

- Annotate a method with `@After` to make it a post-test fixture
  - It is executed after each test method is run
  - It is where you would release system resources (such as streams)
- In most cases, a post-test fixture is not required
  - Before the next test is executed the setup fixture is run so attribute will be re-initialized

TestSuite

- Combines many test cases in a test suite:

```
@RunWith(Suite.class)
@SuiteClasses({
   TestStack.class, AnotherTest.class
})
public class AllTests { }
```


JUnit 4 Annotations

- **`@Test`**
  - Marks test methods
- **`@Before`** and **`@After`**
  - Mark pre and post fixtures
- Test suites require:
- **`@RunWith(Suite.class)`**
- **`@Suite.SuiteClasses({ … })`**

JUnit 4 Packages and classes

- All classes are in packages **`org.junit`**
- Assertions are made available with
  - **`import static org.junit.Assert.*;`**
- Annotations have to be imported as
  - **`import org.junit.After;`**
  - **`import org.junit.Before;`**
  - **`import org.junit.Test;`**

Counter test with Junit 4

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class CounterTests {
   private Counter counter;
   @Before
   public void setUp() throws Exception {
      counter = new Counter(); }
   @After
   public void tearDown() throws Exception {}
```

Counter test with Junit 4

```
    @Test
    public void testGetCounterInitial() {
       assertEquals(0,counter.getCount()); }
    @Test
    public void testIncrement() {
       assertEquals(1,counter.increment());
       assertEquals(2,counter.increment()); }
    @Test
    public void testDecrement() {
       assertEquals(-1,counter.decrement()); }
}
```


Junit 4 test suite

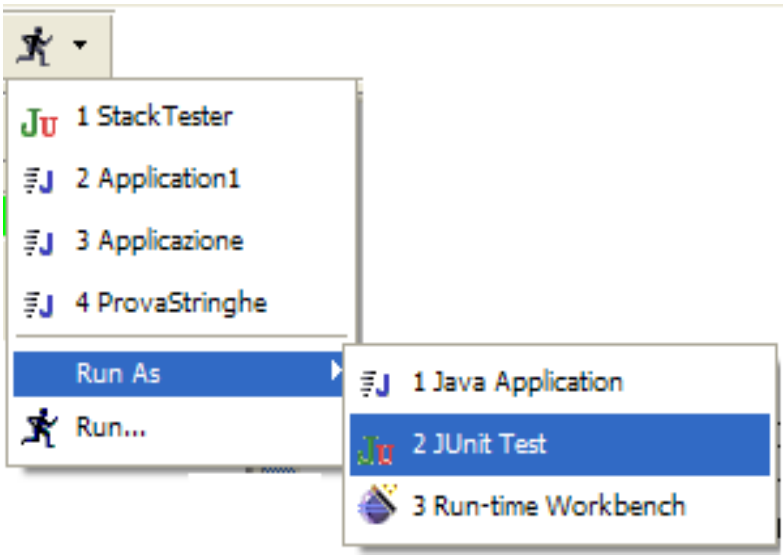
```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ CounterTests.class })
public class AllTests { }
```

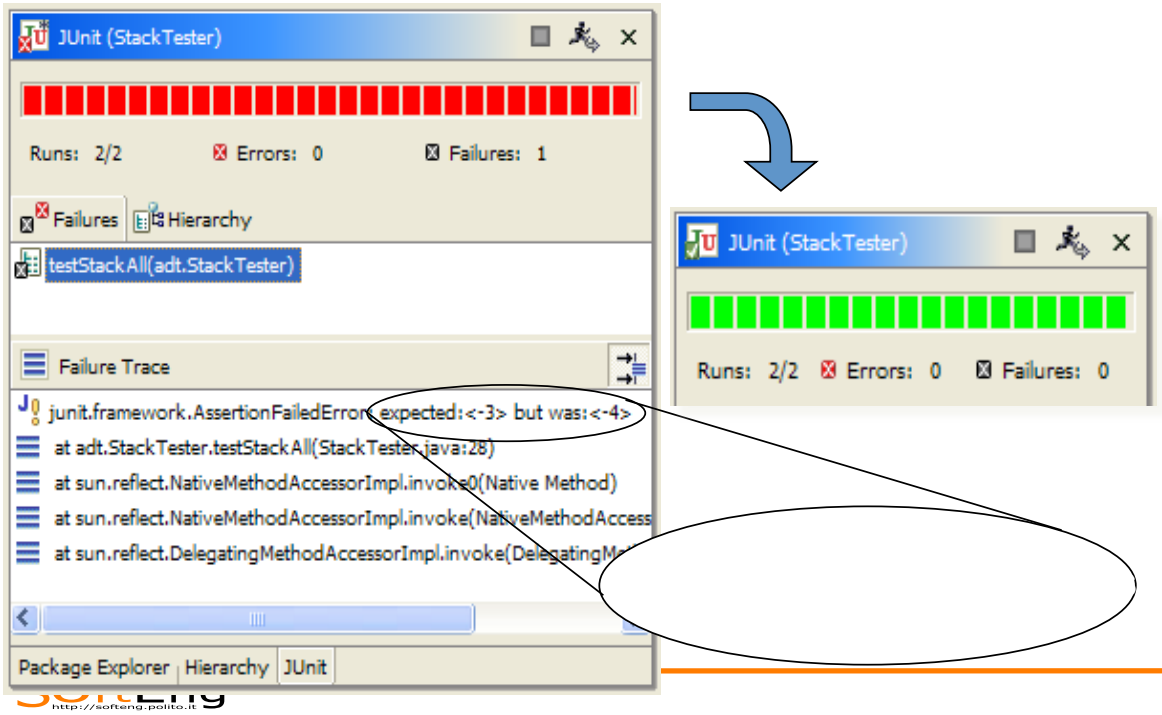
ECLIPSE JUNIT PLUG-IN

Junit in Eclipse – Run as JUnit Test

- Run
- Run As..
- Junit Test



Red / Green Bar



...use JUnit

Keep the bar green to keep the code clean...

Organizing Tests in Eclipse

- Second source folder
  - Place tests within a second source folder
  - Allows clear separation
  - Add JUnit library to the project
- Separate project
  - Place tests inside a separate project
  - No unit test libraries are added to your primary project
  - Refer to the primary project

JUnit in Eclipse – Path Setup

- When creating a new test case
  - Eclipse suggests adding the JUnit library
- When importing a test, the library must be added explicitly
  - open project's property window
  - java build path
  - libraries
  - JUnit

USING JUNIT

Test-Driven Development

- Specify a portion of the feature yet to be coded
- Run the test and see it fail (red bar)
- Write code until the tests pass (green bar)
- Repeat until whole feature implemented
- Refactor
  - Keeping the bar green

Bug reproduction

- When a bug is reported
- Specify the expected correct outcome
- See the test fail
  - Reproduce the bug
- Modify the code until the bug-fix tests pass.
- Check for regressions

Guidelines

- Test should be written before code
- Test everything that can break
- Run tests as often as possible

- Whenever you are tempted to type something into a print statement or a debugger expression write it as a test

  – M.Fowler

Limitations of unit testing

- JUnit is designed to call methods and compare the results they return against expected results
  - This ignores:
    - Programs that do work in response to GUI commands
    - Methods that are used primary to produce output

Limitations of unit testing…

- Heavy use of JUnit encourages a "functional" style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse

Summary: elements of JUnit

- assert*()
  - Comparison functions
- Test cases
  - Are implemented by methods in test classes
- TestSuite
  - Class containing a sequence of TestCase

Why JUnit

- Allow you to write code faster while increasing quality
- Elegantly simple
- Check their own results and provide immediate feedback
- Tests is inexpensive
- Increase the stability of software
- Developer tests
- Written in Java
- Free
- Gives proper uniderstanding of unit testing

References

- K.Beck, E.Gamma. **Test Infected: Programmers Love Writing Tests**
  - http://members.pingnet.ch/gamma/junit.htm
- Junit home page
  - https://junit.org
- Hamcrest matchers
  - http://hamcrest.org/JavaHamcrest/