JUnit tests

Object-Oriented Programming

http://softeng.polito.it

Version 4.5.1

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 most IDEs

JUnit

http://www.junit.org

Unit Testing

- Unit tests are intended to verify the correctness of units of a program
  - Unit are considered in isolation
  - In Java, units are classes and packages
- Unit testing is particularly important when software requirements change frequently
  - Code often need 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: 4.13.2 – Feb 2021
- Junit 5.0 – September 2017
  - Latest release: 5.12.0 – Feb 2025

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
    - The actual test that checks the output of the element under test
  - calls post–test fixtures
    - Intended to release resources and remove any objects created that is no longer needed

Test method

- A test method returns no result
- It performs operations on the code under test and checks the results
- Checks are performed using a set of `assert*()` methods
- The JUnit framework detects the anomalies and reports them

Standard **`assert*()`** methods

```
assertTrue(boolean test)

assertFalse(boolean test)

assertEquals(expected, actual)

assertSame(Object exp, Object actual)

assertNotSame(Object exp, Object act)

assertNull(Object object)

assertNotNull(Object object)

fail()
```

Standard `assert*()` methods

- For a boolean condition

`assertTrue(condition)`

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

`assertFalse(condition)`

- Fails if condition is **`true`**

Standard `assert*()`

- Equality of primitives and objects, :

`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);`

Standard **`assert*`** w/message

- All the assertion methods may take an optional addition message argument, e.g.

JUnit 3 / 4

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

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

static void assertTrue(
    boolean test,
    Supplier<String> messageSupplier)
```

JUnit 5

Fixtures

- Portions of test cases that are cloned
  - For initializing or releasing resources
- They are called fixtures
  - Pre-test (set up)
  - Post-test (tear down)
- Can be collected in separate methods
  - Avoid duplication
  - Shorten test cases
  - Automatically executed before tests

Failures vs. Errors

- Failure
  - An **`assert*()`** method found the condition it checked is not verified
  - The program produced an output, but it is not the expected one
- Error
  - During the execution of the tests an error was found (e.g., **`NullPointerException`**)
  - The program could not produce any output due to an error

Skipped tests

- Under some conditions test may be skipped
  - Some resource is not available
  - Dependencies are not ready
  - Context does not match expected conditions

Sample class: Stack

```
public class Stack {
private int[] stack; private int next = 0;
public Stack(){ this(10);}
public Stack(int size){ stack = new int[size];}
public boolean isEmpty(){ return next==0; }
public boolean push(int i) {
   if(next==stack.length) return false;
   stack[next++] = i;
   return true;   }
public int pop() throws StackException {
   if(next==0) throw new StackException()
   return stack[--next]; }}
```

Testing exceptions

- In presence of exceptions two main cases shall be checked:
  - a normal behavior is expected, therefore no exception should be thrown
    - In this case the tests fails if that exception is raised
  - an anomaly is expected, therefore an exception should be thrown
    - In this case the tests fails if NO exception is detected

Junit 4

SYNTAX

JUnit 4

- Makes 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)

Any class

@Test annotation

```
public class TestStack {
   @Test
   public void testStack()
                          throws StackException {
    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());
   }
}
```

One or more assertions to check results

Running a test case

- The JUnit runner executes all methods
  - Annotated with "`@Test`"
  - `public`
  - Returning `void`
  - With no arguments `()`
  - Ignores the rest of the class
- The class may contain any helper methods provided they are
  - Not annotated
  - Not public

The pre-test fixture

- Annotate a method with `@Before` to make it a pre-test fixture:
  - It is executed before each test method is run
  - It is intended to initialize the objects that will be used by test methods
- There is no limit to the setup you can do in a pre-test method
- 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 intended to release system resources (such as streams)
- In most cases, a post-test fixture is not required
  - Before the next test is executed the pre-test fixture is run again so attribute will be re-initialized

TestingExceptions

- When an exception is expected from the test it must be declared

```
@Test(expected=PossibleException.class)
```

  - If not declared an exception will be counted as an Error

- Or the asserted:

```
   assertThrows("message", Exception.class,
                  ()-> { … }
```

Expected exception test

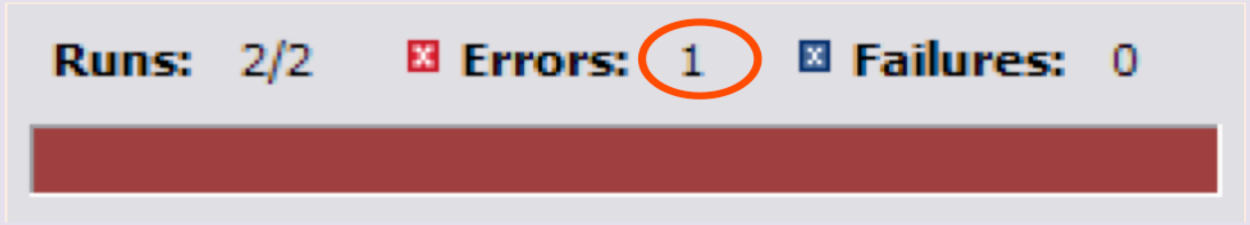
```
@Test(expected=StackException.class)
public void testEmptyPop()
                          throws StackException {
   Stack aStack = new Stack();
   aStack.pop();}


@Test
public void testEmptyErrorAssert() {
   Stack aStack = new Stack();
   assertThrows("Expected exc. on empty pop",
                   StackException.class,
                   ()-> aStack.pop());}
```

Unexpected exception test

```
@Test
public void testPop()
                throws StackException {
  Stack aStack = new Stack();
  aStack.push(1);
  aStack.pop();
}
```

Exception → Error


Runs: 2/2 Errors: 1 Failures: 0

TestSuite

- Allows running a group of related tests as a single batch:

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

Skipping tests

- Some tests can be intended to be skipped, e.g., not ready yet:

```
@Ignore("Incomplete test")
```

- Other tests can be executed only is some condition are met, e.g., a resource is available

```
assumeTrue("X not available",
           res.isAvailable())
```

JUnit 4 Annotations

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

JUnit 4 Imports

- All classes are in package **`org.junit`**
- Assertions are made available with
  - **`import static org.junit.Assert.*;`**
  - **`import static org.junit.Assume.*;`**
- Annotations must be imported as
  - **`import org.junit.After;`**
  - **`import org.junit.Before;`**
  - **`import org.junit.Test;`**
- Suites require:
  - **`import org.junit.runners.Suite;`**
  - **`import org.junit.runners.Suite. SuiteClasses;`**

Junit 5

SYNTAX

JUnit 5

- Uses Java annotations but different from 4:
  - `@Before/@After` → `@BeforeEach/@AfterEach`
  - `@BeforeClass/@AfterClass` → `@BeforeAll/@AfterAll`
- Test methods not necessarily public
- Java8 Lambda support, extensions, parameterized tests, etc.
- Suites
  - `@RunWith(JUnitPlatform.class)`
  - `@SelectClasses({ … })`
  - `@SelectPackages({ … })`

Test a Stack (JUnit5)

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

Any class

@Test annotation

Message as last argument

One or more assertions to check results

Expected exception test

```
@Test
public void testSomething(){
   // e.g. method invoked with "wrong" args
  assertThrows(PossibleException.class,()->{
     obj.method("Wrong Argument")
  });
}
```

```
class TheClassUnderTest {
  public void method(String p)
                 throws PossibleException
  { /*... */ }
}
```

Skipping tests

- Some tests can be intended to be skipped, e.g., not ready yet:

```
@Disabled("Incomplete test")
```

- Other tests can be executed only is some condition are met, e.g., a resource is available

```
assumeTrue("X not available",
           res.isAvailable())
```

TestSuite

- Indicate the `JUnitPlatform` runner
- Select classes with `SelectClasses`

```
@RunWith(JUnitPlatform.class)
@SelectClasses({
   TestStack.class, AnotherTest.class
})
public class AllTests { }
```

JUnit 5 Imports

- All in package **`org.junit.jupiter.api`**
- Assertions are made available with
  - **`import static org.junit.jupiter.api.Assertions.*;`**
- Annotations have to be imported as
  - **`import org.junit.jupiter.api.AfterEach;`**
  - **`import org.junit.jupiter.api.BeforeEach;`**
  - **`import org.junit.jupiter.api.Test;`**
- Suites require:
  - **`import org.junit.platform.runner.JUnitPlatform;`**
  - **`import org.junit.platform.suite.api.SelectClasses;`**

Using JUnit

Test-Driven Development

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

Keep your code clean:
keep the bar green

Bug reproduction

- When a bug is reported
- Specify the expected correct outcome
- See the test fail
  - I.e. reproduce the bug
- Modify the code and adjust it until the bug-reproducing tests pass.
- Check for regressions
  - With the existing test suites

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

References

- K.Beck, E.Gamma. Test
  Infected: Programmers Love Writing Tests
  - http://members.pingnet.ch/gamma/junit.htm
- Junit 5 home page
  - https://junit.org
- Junit 4 home page
  - https://junit.org/junit4/
- Hamcrest matchers
  - http://hamcrest.org/JavaHamcrest/
- AssertJ – Fluent assertions
  - http://joel-costigliola.github.io/assertj/index.html

Additional materials

APPENDIX

JUnit 3

SYNTAX

Test a Stack

extends TestCase

```
public class StackTest extends TestCase {
   public void testStack() {
      Stack aStack = new Stack();
      assertTrue("Should be empty at first!",
                 aStack.isEmpty());
      aStack.push(10);
      assertTrue("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());
}
```

Test case execution

- Running a test case
  - Executes all methods
    - `public`
    - Returning `void`
    - Name starting with "`test`"
    - With no arguments `()`
  - Ignores the rest of methods
- The class can contain helper methods provided they
  - are not public or
  - do not start 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.

Pre-test fixture

- Override `setUp()` to initialize the variables, and objects
  - Implements a initialization fixture
- Since `setUp()` is your code, you can modify it any way you like (such as creating new objects in it)
  - Typically it initializes instance attributes that are later used by test methods

Post-test fixture

- 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)

Expected exception test

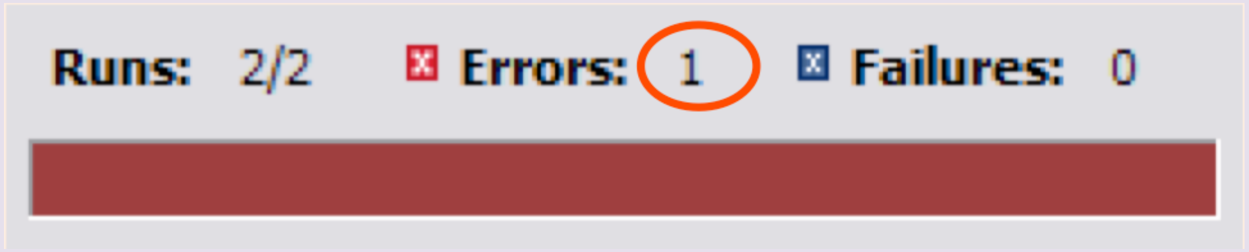
```
try{
   //  e.g. method invoked with "wrong" args
   obj.method(null);
   fail("Method didn't detected anomaly");
}catch(PossibleException e){
   assertTrue(true); // OK
}
```

```
class TheClassUnderTest {
 public void method(String p)
              throws PossibleException
   { /*... */ }
}
```

Unexpected exception test

```
public void testSomething()
     throws PossibleException {
  // e.g. method invoked with right args
  obj.method(“Right Argument");
}
```

Exception → Error

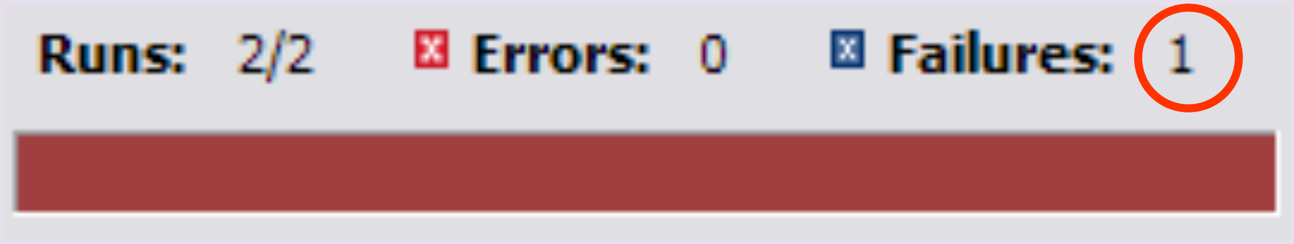


Runs: 2/2 Errors: 1 Failures: 0

Unexpected exception test

```
try{
   //  e.g. method invoked with right args
   obj.method(“Right Argument");
   assertTrue(true); // OK
}catch(PossibleException e){
   fail(“Method should not raise except.");
}
```

Exception → Failure


Runs: 2/2   Errors: 0   Failures: 1

TestSuite

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

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

Assertion Styles

Assertions Styles

- Different ways of writing assertions
  - Standard JUnit assert methods
  - Hamcrest matchers
  - AssertJ fluent assertions

Hamcrest matchers

- A single assert method:

```
assertThat(value, matcher)
```

- Accepts the actual value and a matcher, e.g.

```
assertThat(res, is(equalTo(expect)));
```

  - This is equivalento to:
    - `assertEquals(expect, res);`
- More readable, better message

Hamcrest matchers – Object

- **`equalTo()`**
  - test object equality using Object.equals
- **`instanceOf()`**
  - test type
- **`notNullValue()`**, **`nullValue()`**
  - test for null
- **`sameInstance()`**
  - test object identity

Hamcrest matchers – Numbers

- **`closeTo()`**
  - test floating point values are close to a given value
- **`greaterThan()`**, **`greaterThanOrEqualTo()`**, **`lessThan()`**, **`lessThanOrEqualTo()`**
  - test ordering

Hamcrest matchers – Logical

- **`allOf()`**
  - matches if all matchers match, short circuits (like Java &&)
- **`anyOf()`**
  - matches if any matchers match, short circuits (like Java ||)
- **`not()`**
  - matches if the wrapped matcher doesn't match and vice versa

Hamcrest matchers – String

- **`equalToIgnoringCase()`**
  - test string equality ignoring case
- **`equalToIgnoringWhiteSpace()`**
  - test string equality ignoring differences in runs of whitespace
- **`containsString()`**, **`endsWith()`**, **`startsWith()`**
  - test string matching

Hamcrest matchers – Collections

- **`array()`**
  - test an array's elements against an array
- **`hasItemInArray()`**
  - test an array contains an element
- **`hasItem()`**, **`hasItems()`**
  - test a **`Collection`** contains elements
- **`hasEntry()`**, **`hasKey()`**, **`hasValue()`**
  - test a **`Map`** contains an entry, key or value

Hamcrest imports

- Required imports
  - `import static org.hamcrest.MatcherAssert.assertThat;`
  - `import static org.hamcrest.Matchers.*;`
- The jar with only core matchers is included in Eclipse when using the Java4 library

AssertJ Fluent

- A single assert builder method:

  `assertThat(actual)`

  - Accepts the actual value
- Returns an Assert object the provides building methods
- Method **`as()`** can be used to define a contextual message in case of failure
- More readable, better messages

AssertJ Fluent example

```
    assertThat(res).
        as("Checking return value").
        isNotNull().
        isEqualTo(expected);
```

- Equivalent to:

```
    assertNotNull("Checking return value", res);
    assertEquals("Checking return value",
                  expected, res);
```

Using AssertJ

- Download latest jar from Maven
  - https://search.maven.org/remotecontent?filepath=org/assertj/assertj-core/3.12.2/assertj-core-3.12.2.jar
- Include the jar in the classpath
- Import the static definitions:

```
import static org.assertj.core.api.Assertions.*;
```

AssertJ – Object

- `isEqualTo()`
  - test object equality using `Object.equals`
- `isInstanceOf()`
  - test type
- `isNotNull()`, `isNull()`
  - test for `null`
- `isSameAs()`
  - test object identity

AssertJ – Object

- **`returns()`**
  - test a value is returned by **`Function`**
- **`hasFieldOrPropertyWithValue()`**
  - Retrieves value (using reflection) looking for:
    - Getter: **`field`** → **`getField()`**
    - Field: attribute in class
- **`extracting()`**
  - Applies extractor **`Function`** and produces an object or a list that becomes the actual value

Checking method return

- Equivalent assertions:

```
assertThat(new Counter())
.isNotNull()
.hasFieldOrPropertyWithValue("value", 0)

.returns(0, Counter::getValue)

.extracting(c -> c.getValue())
          .isEqualTo(0);
```

Chec

- Equ

```
asser
.isNotNull()
.hasFieldOrPropertyWithValue("value", 0)


.returns(0,

.extracting(c -        getValue())
            .isEqualTo(0);
```

```
java.lang.AssertionError: [Initial counter state]
Expecting
  <TestExampleAsserJ$Counter@3339ad8e>
to have a property or a field named <"value"> with value
  <0>
but value was:
  <1>
```

```
org.junit.ComparisonFailure: [Initial counter state]
expected:<[0]> but was:<[1]>
```

AssertJ – Numbers

- `isCloseTo()`
  - test fp value is close to a given value
- `isGreaterThan()`, `isGreaterThanOrEqualTo()`, `isLessThan()`, `isLssThanOrEqualTo()`
  - test ordering
- `isBetween()`
  - test actual value is in range

AssertJ – Strings

- **`startsWith()`**, **`endsWith()`**
  - test endings
- **`contains()`** , **`doesNotContain()`**
  - test content contains a substring
- **`isEqualToIgnoringCase()`**
  - test content

AssertJ – Containers

- `hasSize()`, `hasSizeLessThan()`, `hasSizeGreaterThan()`,…
  - test collection size
- `isSorted()`
  - test if list is sorted
- `contains()`, `containsExactly()`,
  - test that elements are present
- `extracting()`
  - Map each element to a list
- `filteredOn()`
  - Filters the element of the list

AssertJ – Maps

- **`containsKeys()`**, **`containsEntry()`**
  - test if list is sorted
- **`contains()`**
  - checks entries, that can be defined with **`entry()`**
- **`extractingFromEntries()`**
  - Map each entry to a list or a list of tuples that can be matched with a contains of **`tuple()`**

Maps example

```
assertThat(turnout).as("Turnout map").
contains(
   entry("PIEMONTE",67.45),
   entry("SICILIA",42.88),
   entry("UMBRIA",70.50)
);
```

AssertJ – Exceptions

- **`assertThatThrownBy()`**
  - Create an assert on the thrown exception
- **`hasMessage()`**
  - Checks the message of exception