JUnit & Mockito

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

Topics covered

- Introduction to JUnit
- JUnit: Hands-on session
- Introduction to Mockito
- Mockito: Hands-on session

Introduction to JUnit

What is JUnit?

- **JUnit is a testing framework for Java**
- It is a simple framework to write repeatable tests
- A test case is a program written in Java

How to use JUnit

- JUnit is linked as a JAR at compile-time
- Integrate JUnit in your project (with Maven)
  - With Maven

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
...
</dependencies>
```

  - Without Maven: add **junit.jar** on your classpath

JUnit: key concepts

- JUnit is based on **Java annotations**
- Java annotations are a form of metadata, provide data about a program that is not part of the program itself.
- Java annotations have several uses:
  - Information for the compiler
  - Compile-time and deployment-time processing
  - Runtime processing

JUnit most used annotations

- @org.junit.Test
- @org.junit.BeforeClass
- @org.junit.Before
- @org.junit.AfterClass
- @org.junit.After

Test class template

```
import org.junit.*;
public class TestClass1 {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }
    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }
    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }
    @After
    public void tearDown() throws Exception {
        // Code executed after each test
    }
}
```

Test class template

```
@Test
public void testOne() {
    // Code that performs test one
}
@Test
public void testTwo() {
    // Code that performs test two
}
```

JUnit assertions

- JUnit provides **assertion methods** for all primitive types and Objects and arrays
- In these methods the **expected value** is compared with the **actual value**.
- The parameter order is:
  - Optional: a string that is output on failure
  - expected value
  - actual value

JUnit assertions

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

assertEquals("failure - strings not equal", "text",
"text");
assertFalse("failure - should be false", false);
assertSame("should be same", number, number);
assertArrayEquals("failure - byte arrays not same",
expected, actual);
```

Ignore a test

- If you want to ignore/disable temporarily a test you can do it with the `@Ignore` annotation

```
@Ignore("Test is ignored as a demonstration")
@Test
public void testMethod() {
    assertThat(1, method());
}
```

Set a timeout

- Tests that take too long, can be automatically failed
- Two option for timeout:
  - Timeout parameter on `@Test` annotation. (Timeout on a single test method)

```
@Test(timeout=1000)
public void testWithTimeout() {
 ...
}
```

  - Timeout Rule. Timeout on all the test methods in the class

```
public class TestClassGlobalTimeout {
 @Rule
 public Timeout globalTimeout = new Timeout(10000);
 @Test
 public void testMethod(){
     …
 }
}
```

Run tests

Two ways to run tests:

- Using the JUnitCore class and see the results on console

  `java org.junit.runner.JUnitCore TestClass1 [...other test classes...]`

  - Both your test class and JUnit JARs must be on the classpath
- **Using Maven (simpler!), just execute**

  `mvn test`

  The Surefire plugin of Maven will execute all the JUnit test classes under `src/test/java`

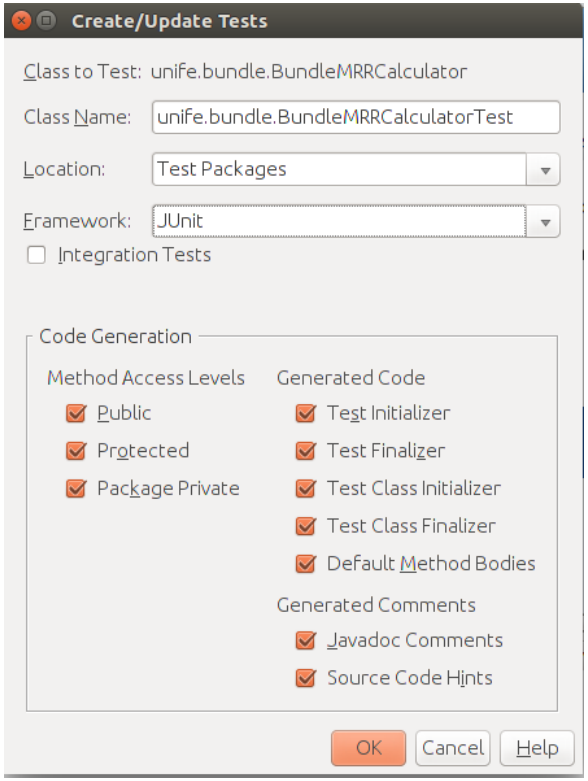
References

- JUnit official site:
  - http://junit.org
- Tutorials:
  - http://www.vogella.com/tutorials/JUnit/article.html
  - http://www.html.it/articoli/junit-unit-testing-per-applicazioni-java-1/
- Other:
  - http://en.wikipedia.org/wiki/JUnit

JUnit: Hands-on session

Generate test in NetBeans

- Right click on a class and
  Tools > Create Junit Tests



Unit test of Counter Class

- Requirement: Create a Counter that always starts to count from 0 (0 is the minimum value) and that can increase and decrease a given number
- Code: https://bitbucket.org/giuseta/junit-counter/
  - src/main/java/Counter.java
  - src/test/java/CounterTest.java
- We will see that:
  - testIncrease() will succeed. The Counter increments correctly the number 10. The expected value (11) is equal to the actual value (11)
  - testDecrease(), instead, will fail, because we want a counter that will never have a value below 0. The expected value (0) is NOT equal to the actual value (-1).

Exercise

- Create **some** classes which extend Counter
  - One class should count the number of even numbers less than or equal to the current value.
  - One class should count the number of odd numbers less than or equal to the current value.
  - One class should count the number of prime numbers less than or equal to the current value.
- Which design pattern could we use?
- Test all these classes with JUnit

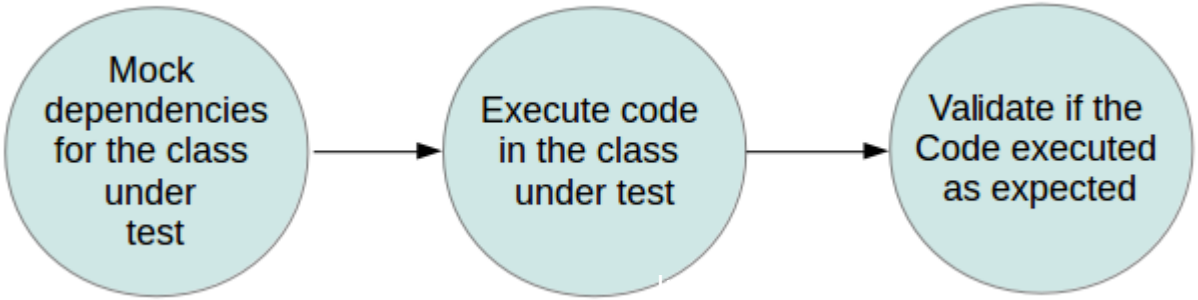
Introduction to Mockito

What is Mockito?

- **Mockito** is a Java framework allowing the creation of **mock objects** in automated unit tests
- A **mock object** is a dummy implementation for an interface or a class in which you define the output of certain method calls.

Why mocking?

- Some “real” objects required in Unit tests are really complex to instantiate and/or configure
- Sometimes, only interfaces exist, implementations are not even coded.
- If you use Mockito in tests you typically:
  - Mock away external dependencies and insert the mocks into the code under test
  - Execute the code under test
  - Validate that the code executed correctly



http://www.vogella.com/tutorials/Mockito/article.html

How to use Mockito

- Integrate Mockito in your project with Maven
  - With Maven

```
<dependencies>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.10.19</version>
        <scope>test</scope>
    </dependency>
    ...
</dependencies>
```

  - Without Maven: add Mockito JARs on your classpath

Mocking a class

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

@Test
public void test1()  {
        //  create mock
        MyClass test = mock(MyClass.class);

        // define return value for method getUniqueId()
        when(test.getUniqueId()).thenReturn(43);

        // use mock in test....
        assertEquals(test.getUniqueId(), 43);
}
```

Mockito: Verify

- Once created, mock will remember all interactions
- Then you can **verify** whatever an interaction happened

```
import static org.mockito.Mockito.*;
...
//mock creation
List mockedList = mock(List.class);
//using mock object
mockedList.add("one");
mockedList.clear();
//verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

Argument matchers

- Mockito verifies argument values by using an equals() method
- When flexibility is required then you should use **argument matchers**

```
//stubbing using anyInt() argument matcher
when(mockedList.get(anyInt())).thenReturn("element");
//verify using an argument matcher
verify(mockedList).get(anyInt());
```

- Other argument matchers: `anyString()`, `anyObject()`, `anyVararg()`, …
- **Attention!** If you are using argument matchers, all arguments have to be provided by matchers

Mockito: Spy

- With Mockito you can **spy** a real class. When you use the spy then the real methods are called (unless a method was stubbed)

```
List<String> list = new LinkedList<>();
List<String> spy = spy(list);
//optionally, you can stub out some methods:
when(spy.size()).thenReturn(100);
//using the spy calls *real* methods
spy.add("one");
spy.add("two");
//prints "one" - the first element of a list
System.out.println(spy.get(0));
//size() method was stubbed - 100 is printed
System.out.println(spy.size());
//optionally, you can verify
verify(spy).add("one");
verify(spy).add("two");
```

References

- Mockito official site:
  - http://site.mockito.org/
- Tutorials:
  - http://www.vogella.com/tutorials/Mockito/article.html
- Other:
  - https://en.wikipedia.org/wiki/Mockito

Mockito: Hands-on session

Unit test with Mockito of a class that uses Counter

- Create a class that takes an instance of the class Counter. This class should have a method that multiplies the value of the Counter instance with a given integer value
- Code: https://bitbucket.org/giuseta/junit-counter/
  - src/main/java/ClassUsesCounter.java
  - src/test/java/ClassUsesCounter/Test.java
- Create a mock object of the Counter class