Advanced Object-Oriented Design
Lecture 3

Unit Testing

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

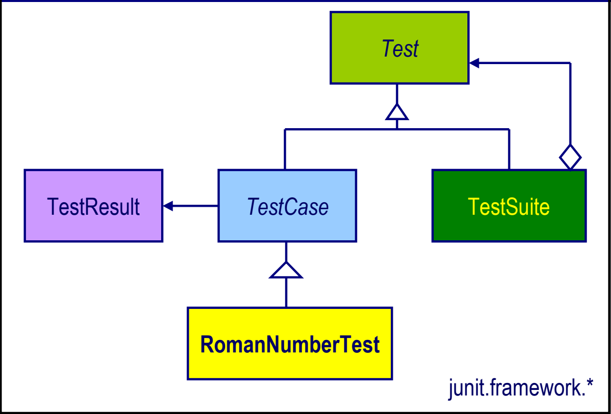
Agenda

1. Basic principles of unit testing
2. Test cases in JUnit 3.x
3. Guidelines for creating test cases
4. Tasks for implementation
5. Annotated test cases in JUnit 4.0
6. Mock objects

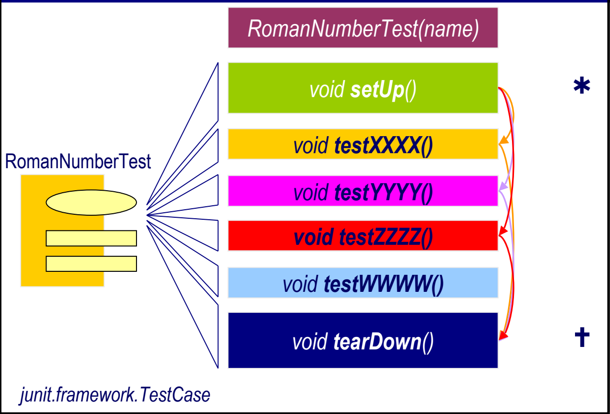
Unit testing

Test cases in JUnit 3.x

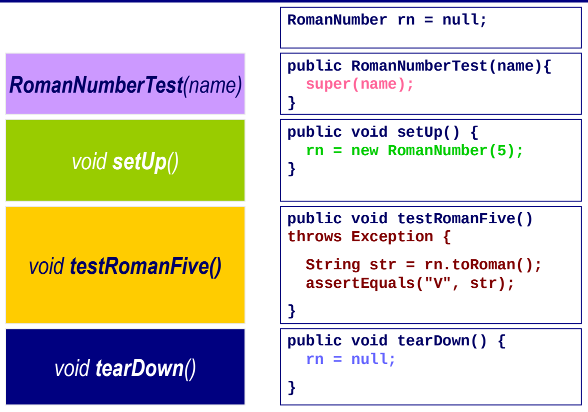
JUnit 3.x core classes



TestCase



Simplest test case



```
RomanNumber rn = null;

public RomanNumberTest(name){
   super(name);
}

public void setUp() {
   rn = new RomanNumber(5);
}

public void testRomanFive()
throws Exception {
   String str = rn.toRoman();
   assertEquals("V", str);
}

public void tearDown() {
   rn = null;
}
```

JUnit 3.x: failure vs. error

Failure:
- anticipated violation of the test assertion
- signals that the test actually fails
- marked by *junit.framework.AssertionFailedError*

Error:
- unanticipated runtime exception caught by the test runner
- the test could not be run properly due to external reasons
- gives no advice on the tested object!

Failure vs. Error (*)

```
public void testNonexistentFileRead()
throws IOException {
    try {
        File file = new File("doesNotExist.txt");
        FileReader reader = new FileReader(file);
        assertEquals('a', (char) reader.read());
        fail("Read from a nonexistent file?!");
    } catch (FileNotFoundException success) {}
}
```

```
public void testExistingFileRead()
throws IOException {
    // assuming that exists.txt already exists
    File file = new File("exists.txt");
    FileReader reader = new FileReader(file);
    assertEquals('a', (char) reader.read());
}
```

Basic functionality of a *TestCase*

Groups of methods:
- equality tests
  - `void assertEquals([msg], expected, actual)`
- identity tests
  - `void assertSame([msg], expected, actual)`
  - `void assertNotSame ([msg], expected, actual)`
- boolean tests
  - `void assertTrue([msg], condition)`
  - `void assertFalse([msg], condition)`
- null tests
  - `void assertNull([msg], object)`
  - `void assertNotNull([msg], object)`
- unconditional failure
  - `void fail([msg])`

Creating *TestSuites* statically

```
public class RomanNumberTest extends TestCase {

    public RomanNumberTest(String name) {
        super(name);
    }
    // testing methods
    public void testSimpleConv() {}
    public void testAddition() {}

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(
            new RomanNumberTest("testSimpleConv"));
        suite.addTest(
             new RomanNumberTest("testAddition"));
        return suite;
    }
}
```

Creating *TestSuites* dynamically

```
public class RomanNumberTest extends TestCase {

    public RomanNumberTest(String name) {
        super(name);
    }

    // testing methods
    public void testSimpleConv() {}
    public void testAddition() {}

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(MyTest.class)
        return suite;
    }
}
```

Unit testing

Guidelines for creating test-cases

Guidelines for creating *TestCases*

Don't use constructor for initializing the *TestCase*

```
public class RomanNumberTest extends TestCase {
   private RomanNumber = null;

   public RomanNumberTest (String testName) {
      super (testName);
      rn = new RomanNumber(-1);
   }
}
```

```
junit.framework.AssertionFailedError: Cannot instantiate
     test case: RomanNumberTest
at junit.framework.Assert.fail(Assert.java:143)
at junit.framework.TestSuite$1.runTest(TestSuite.java:178)
at junit.framework.TestCase.runBare(TestCase.java:129)
at junit.framework.TestResult$1.protect
     (TestResult.java:100)
...
```

Guidelines for creating *TestCases*

Don't use constructor for initializing the *TestCase*

```
public class RomanNumberTest extends TestCase {
   private RomanNumber = null;

   public void setUp() {
      rn = new RomanNumber(-1);
   }
}
```

```
java.lang.IllegalStateException: No negatives allowed...
at com.xyz.RomanNumberTest.setUp(RomanNumberTest.java:8)
at junit.framework.TestCase.runBare(TestCase.java:127)
at junit.framework.TestResult$1.protect(
     TestResult.java:100)
...
```

Guidelines for creating TestCases

- Don't assume the order in which tests within a *TestCase* are executed
- Avoid writing *TestCases* with side effects

```
public class RomanNumberTest extends TestCase {

   public RomanNumberTest(String testName) {
      super (testName);
   }

   public void testDoThisFirst () {
   }

   public void testDoThisSecond () {
   }
}
```

Guidelines for creating TestCases

- Don't assume the order in which tests within a *TestCase* are executed
- Avoid writing *TestCases* with side effects

```
public class RomanNumberTest extends TestCase {
   public void testDoThisFirst () {}
   public void testDoThisSecond () {}

   public static Test suite() {
      TestSuite suite = new TestSuite()
      suite.addTest(
         new RomanNumberTest("testDoThisFirst";));
      suite.addTest(
         new RomanNumberTest("testDoThisSecond";));

      return suite;
   }
}
```

Guidelines for creating TestCases

- Avoid using hardcoded resources
- Write self-contained tests
- Place tests in the same packages as source code

```
public class SomeTestCase extends TestCase {
   public void setUp () {
      InputStream input = new FileInputStream
         ("C:\\TestData\\dataSet.dat");
      // ..
   }
}
```

Guidelines for creating TestCases

- Avoid using hardcoded resources
- Write self-contained tests
- Place tests in the same packages as source code

```
public class SomeTestCase extends TestCase {
   public void setUp () {
      InputStream inp = new FileInputStream("dataSet.dat");
   }
}
```

```
public class SomeTestCase extends TestCase {
   public void setUp () {
      InputStream inp = class.getResourceAsStream
         (this.getClass(), "dataSet1.dat");
   }
}
```

Guidelines for creating TestCases

Avoid time/locale-sensitive tests

```
Date date = DateFormat.getInstance().parse("dd/mm/yyyy");


Calendar cal = Calendar.getInstance();
Cal.set(yyyy, mm-1, dd);
Date date = Calendar.getTime();

Locale locale = Locale.getDefault();
```

Guidelines for creating TestCases

- Use JUnit assert/fail methods for throwing Exceptions
- Don't catch exceptions unless they are expected to be thrown
- Don't translate exceptions

```
public void testExemplaryException () {
   try {
      // do some testing
   } catch (AnApplicationException ex) {
      fail ("Caught AnApplicationException exception");
   }
}
```

Guidelines for creating TestCases

- Use JUnit assert/fail methods for throwing Exceptions
- Don't catch exceptions unless they are expected to be thrown
- Don't translate exceptions

```
public void exampleTest ()
throws SomeApplicationException {
   // do some test
}
```

Guidelines for creating TestCases

If the test should pass on exception thrown, catch the exception within the test and invoke *fail()* if it is not thrown

```
public void testIndexOutOfBounds() {

    ArrayList emptyList = new ArrayList();
    try {
        Object o = emptyList.get(0);
        fail("IndexOutOfBoundsException expected");
    } catch (IndexOutOfBoundsException success) {}
}
```

Guidelines for creating TestCases

If the test should pass on exception thrown, use *junit.extensions.ExceptionTestCase* instead

```
// ignore possible exceptions
public void testIndexOutOfBounds() {
  ArrayList emptyList = new ArrayList();
  Object o = emptyList.get(0);
}

public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(
    new ExceptionTestCase("testIndexOutOfBounds",
      IndexOutOfBoundsException.class);

  return suite;
}
```

Guidelines for creating TestCases

Beware of floating-point comparison errors

```
assertEquals ("The result is different from expected",
     result, expected);

assertEquals ("The result is different from expected",
     0.05 + 0.05, 1/10.0);
```

```
assertEquals ("The result is definitely different from
     expected", result, expected, delta);

assertEquals ("The result is definitely different from
     expected", 0.05 + 0.05, 1/10.0, 0.01);
```

Guidelines for creating TestCases

Testing private methods

Package-private:
- Place the tests in the same package along with the tested classes

Private:
- avoid
- use *junitx.util.PrivateAccessor*

Guidelines for creating TestCases

Organizing source files in catalog

```
src
 +--main
 |    |
 |    +--com
 |        |
 |        +--xyz
 |            |
 |            +--SomeClass.java
 +--test
      |
      +--com
          |
          +--xyz
              |
              +--SomeClassTest.java
```

com.xyz.SomeClass

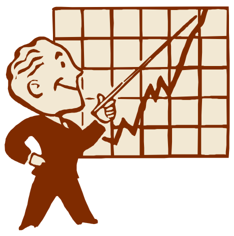
com.xyz.SomeClassTest

Unit testing

Tasks for implementation

Task 1

Write tests for existing code of *RomanNumber* class



RomanNumber

```
class RomanNumber {
   // initialize with an Arabic year
   public RomanNumber(int year)
      throws IllegalArgumentException;
   // initialize with a Roman year
   public RomanNumber(String year)
      throws IllegalArgumentException;
   // return the Roman value
   public String toRoman();
   // return the Arabic value
   public int toArabic();
   // return a Collection of valid Roman symbols
   public static Collection getRomanSymbols();
}
```

Task 2

Implement *RomanNumber* class using test-first approach

Testing first

TestCase:
- INPUT: 1, EXPECTED: "I"

Implementation:
```
String toArabic() {
   return "I";
}
```

Testing first

TestCase:
- INPUT: 2, EXPECTED: "II"

Implementation:
```
String toArabic() {
   if (num == 1)
      return "I";
   else
      return "II";
}
```

Testing first

TestCase:
- INPUT: 3, EXPECTED: "III"

Implementation:
```
String toArabic() {
   if (num == 1)
      return "I";
   else if (num == 2)
      return "II";
   else
      return "III";
}
```

Testing first

TestCase:
- INPUT: 3, EXPECTED: "III"

Refactoring

Implementation:
```
String toArabic() {
   while (num-- > 0)
      result += "I";
   return result;
}
```

Testing first

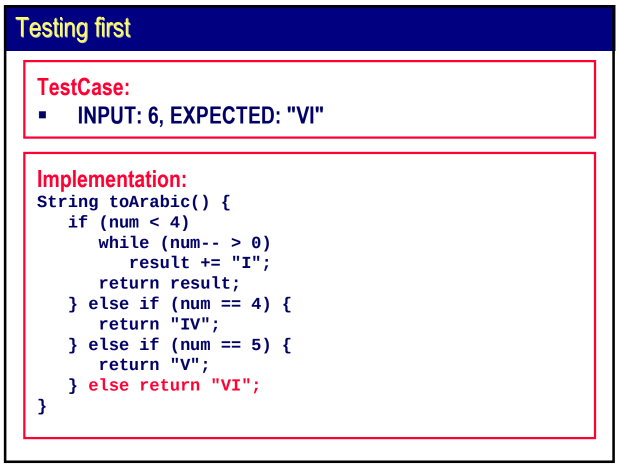
TestCase:
- INPUT: 4, EXPECTED: "IV"

Implementation:
```
String toArabic() {
   if (num < 4) {
      while (num-- > 0)
         result += "I";
      return result;
   }
   return "IV";
}
```

Testing first

TestCase:
- INPUT: 5, EXPECTED: "V"

Implementation:
```
String toArabic() {
   if (num < 4) {
      while (num-- > 0)
         result += "I";
      return result;
   } else if (num == 4) {
      return "IV";
   } else {
     return "V";
   }
}
```



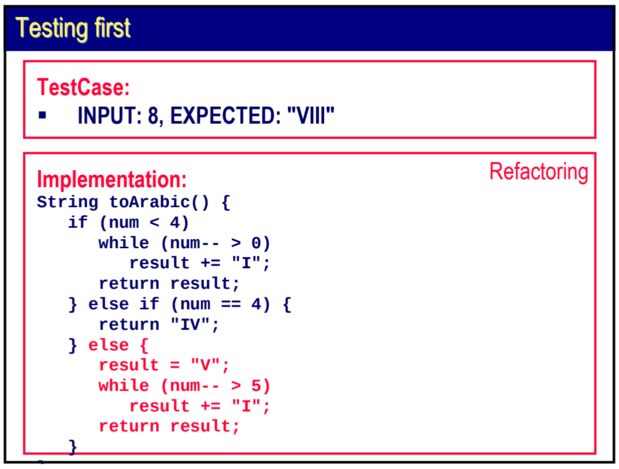
Testing first

TestCase:

- INPUT: 6, EXPECTED: "VI"

Implementation:

```
String toArabic() {
   if (num < 4)
      while (num-- > 0)
         result += "I";
      return result;
   } else if (num == 4) {
      return "IV";
   } else if (num == 5) {
      return "V";
   } else return "VI";
}
```



Testing first

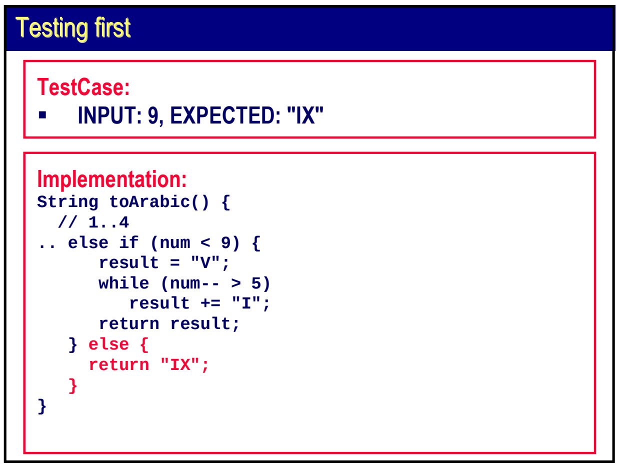
TestCase:

- INPUT: 8, EXPECTED: "VIII"

Implementation:

Refactoring

```
String toArabic() {
   if (num < 4)
      while (num-- > 0)
         result += "I";
      return result;
   } else if (num == 4) {
      return "IV";
   } else {
      result = "V";
      while (num-- > 5)
         result += "I";
      return result;
   }
```



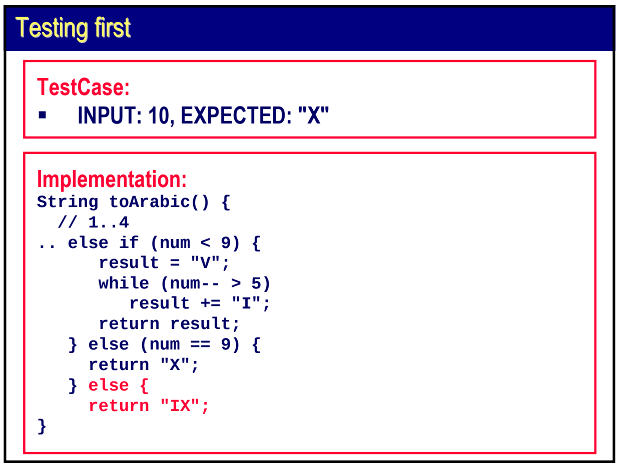
Testing first

TestCase:

- INPUT: 9, EXPECTED: "IX"

Implementation:

```
String toArabic() {
  // 1..4
.. else if (num < 9) {
      result = "V";
      while (num-- > 5)
         result += "I";
      return result;
   } else {
     return "IX";
   }
}
```



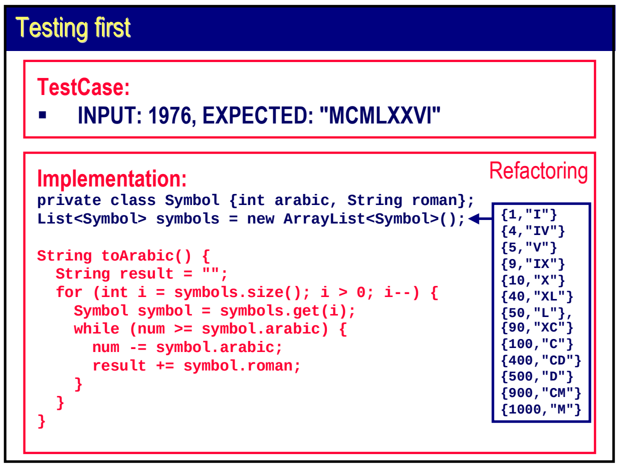
Testing first

TestCase:

- INPUT: 10, EXPECTED: "X"

Implementation:

```
String toArabic() {
  // 1..4
.. else if (num < 9) {
      result = "V";
      while (num-- > 5)
         result += "I";
      return result;
   } else (num == 9) {
     return "X";
   } else {
     return "IX";
}
```



Testing first

TestCase:

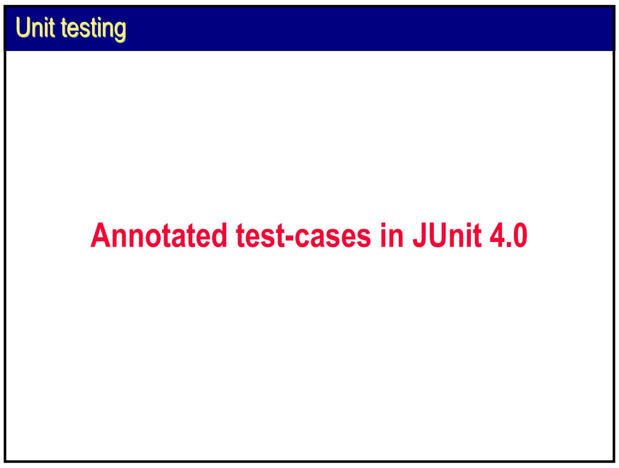
- INPUT: 1976, EXPECTED: "MCMLXXVI"

Implementation:

Refactoring

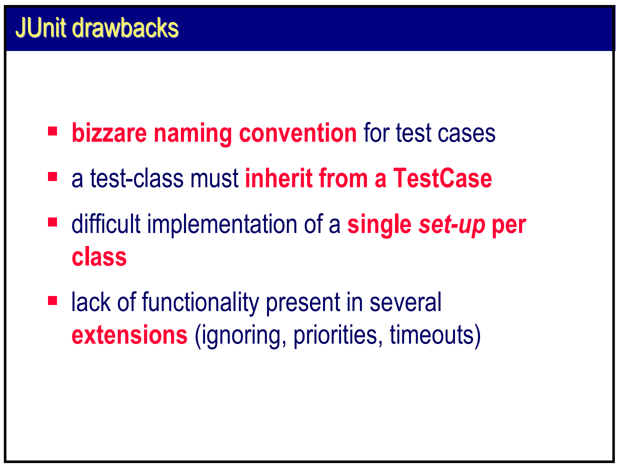
```
private class Symbol {int arabic, String roman};
List<Symbol> symbols = new ArrayList<Symbol>();

String toArabic() {
  String result = "";
  for (int i = symbols.size(); i > 0; i--) {
    Symbol symbol = symbols.get(i);
    while (num >= symbol.arabic) {
      num -= symbol.arabic;
      result += symbol.roman;
    }
  }
}
```

```
{1,"I"}
{4,"IV"}
{5,"V"}
{9,"IX"}
{10,"X"}
{40,"XL"}
{50,"L"},
{90,"XC"}
{100,"C"}
{400,"CD"}
{500,"D"}
{900,"CM"}
{1000,"M"}
```



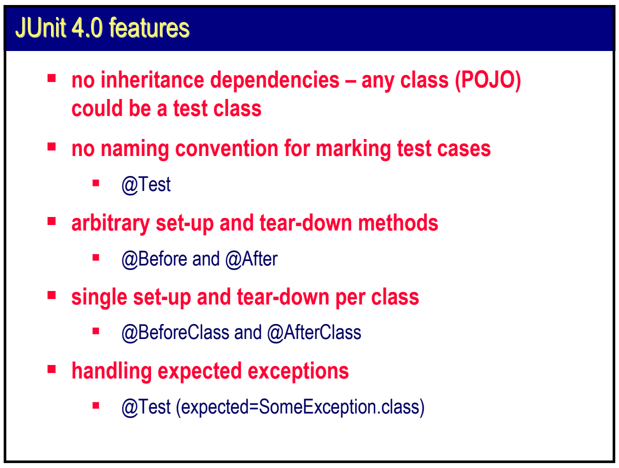
Unit testing

Annotated test-cases in JUnit 4.0



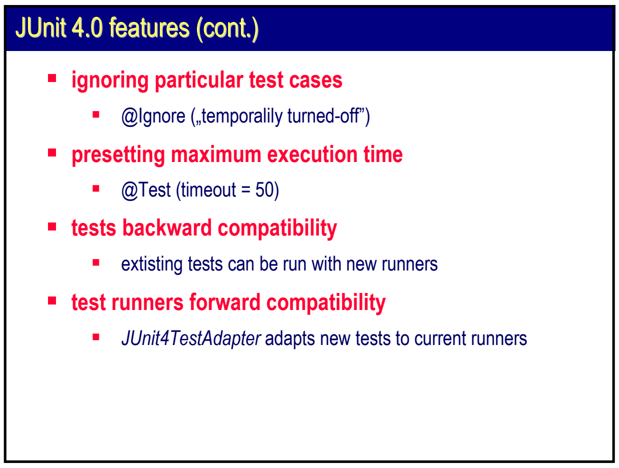
JUnit drawbacks

- **bizzare naming convention** for test cases
- a test-class must **inherit from a TestCase**
- difficult implementation of a **single *set-up* per class**
- lack of functionality present in several **extensions** (ignoring, priorities, timeouts)



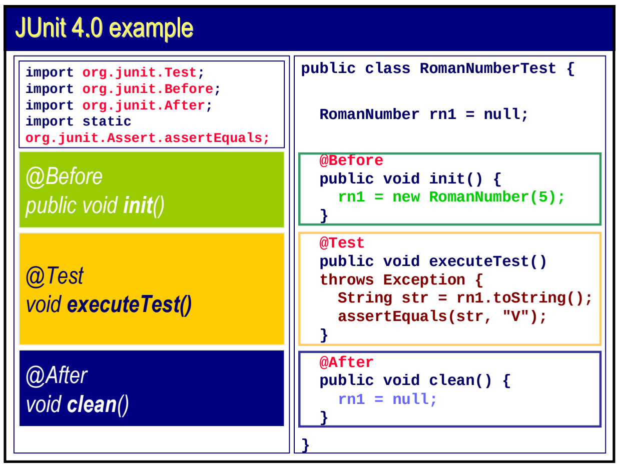
JUnit 4.0 features

- **no inheritance dependencies – any class (POJO) could be a test class**
- **no naming convention for marking test cases**
  - @Test
- **arbitrary set-up and tear-down methods**
  - @Before and @After
- **single set-up and tear-down per class**
  - @BeforeClass and @AfterClass
- **handling expected exceptions**
  - @Test (expected=SomeException.class)



JUnit 4.0 features (cont.)

- **ignoring particular test cases**
  - @Ignore („temporalily turned-off”)
- **presetting maximum execution time**
  - @Test (timeout = 50)
- **tests backward compatibility**
  - extisting tests can be run with new runners
- **test runners forward compatibility**
  - *JUnit4TestAdapter* adapts new tests to current runners



JUnit 4.0 example

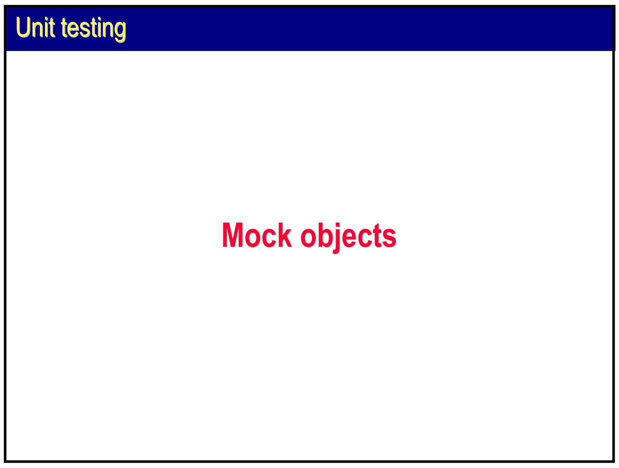
```
import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static
org.junit.Assert.assertEquals;
```

@Before
*public void **init()***

@Test
*void **executeTest()***

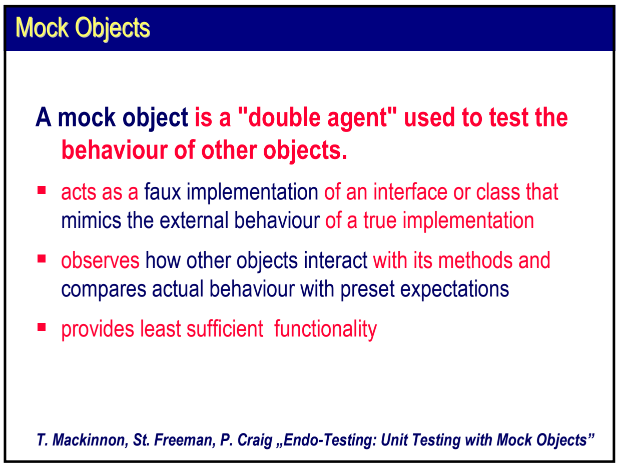
@After
*void **clean()***

```
public class RomanNumberTest {

  RomanNumber rn1 = null;

  @Before
  public void init() {
    rn1 = new RomanNumber(5);
  }

  @Test
  public void executeTest()
  throws Exception {
    String str = rn1.toString();
    assertEquals(str, "V");
  }

  @After
  public void clean() {
    rn1 = null;
  }
}
```



Unit testing

Mock objects



Mock Objects

A mock object is a "double agent" used to test the behaviour of other objects.

- acts as a faux implementation of an interface or class that mimics the external behaviour of a true implementation
- observes how other objects interact with its methods and compares actual behaviour with preset expectations
- provides least sufficient functionality

T. Mackinnon, St. Freeman, P. Craig „Endo-Testing: Unit Testing with Mock Objects”

When mock objects are useful?

- The real object has nondeterministic behavior (it produces unpredictable results; as in a stock-market quote feed.)
- The real object is difficult to set up.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to check to see that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

T. Mackinnon, St. Freeman, P. Craig „Endo-Testing: Unit Testing with Mock Objects"

Using the mock objects

1. **Create instances of mock objects**
2. **Setup the values for mock objects attributes to be returned to the tested objects**
3. **Define expected behavior of the tested objects for the mocks**
4. **Execute the tested method, passing the mock objects as parameters**
5. **Verify consistency of the mock objects**

Mock objects: initialization

```
public class SomeServletTest {
    MockHttpServletRequest mockRequest = null;
    MockHttpServletResponse mockResponse = null;
    MockServletConfig mockConfig = null;
    MyServlet servlet = null;

    public void setUp() {
       mockRequest = new MockHttpServletRequest();
       mockResponse = new MockHttpServletResponse();
       mockConfig = new MockServletConfig();
       servlet = new MyServlet();
    }
```

Mock objects: test execution

```
public void testAliceAged34IsLoggedIn () {
   mockRequest.setupAddParameter("name", "Alice");
   mockRequest.setupAddParameter("age", "34");
   mockRequest.setExpectedAttribute("loggedIn", "true");
   mockResponse.setExpectedOutput(
      "<html><head/><body>Hello, Ala</body></html>");

   servlet.init(mockConfig);
   servlet.doGet(mockRequest, mockResponse);

   assertEquals("Expected attribute is not set", "true",
      mockRequest.getAttribute("loggedIn"));
   mockRequest.verify();
   mockResponse.verify();
}
```

Testing the mocks

- When a discrepancy occurs, a mock object can interrupt the test and report the anomaly.
- If the discrepancy cannot be noted during the test, a verification method called by the tester ensures that all expectations have been met or failures reported.

Mock objects: verification

Asserts

verify (**on the test-case's side**) if the the **actual results** of tested objects matches the **expected behavior**

Verifications

check (**on the mock objects' side**) if the tested objects **properly interact** with mock object

- number of method calls.
- consumption of parameters, etc.

Advantages of mock objects

- allow for unit testing **independently of the production infrastructure**
- tests **do not need to rely on external resources** (database connections, service containers, systems etc.)
- **simulate states that may be difficult or time-consuming to realize in a runtime environment**; a mock object can throw any exception or produce any error condition on demand.
- **provide expectations** as an additional level of contract verification

Implementation: EasyMock

- creates a mock object for an interface at runtime
- benefits from JDK 1.4 *dynamic proxies*
- defines only methods actually needed for testing
- operates in two modes:
  - *recording* (specifying the behavior)
  - *replay* (interacting with tested object)

T. Freese, http://www.easymock.com

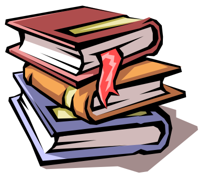
Implementation: EasyMock (cont.)

```
import junit.framework.*;
import org.easymock.MockControl;
public class RomanNumberTest extends TestCase {
   private RomanNumber rn = null;
   private MockControl control;
   public void setUp() {
      // create a control handle to the Mock object
      control = MockControl.createControl(RomanNumber.class);
     // create the Mock object itself
      rn = (RomanNumber) control.getMock();
   }
```

Implementation: EasyMock (cont.)

```
   public void testEasyMockDemo() {
      // set up the mock object by calling methods you want to exist
      rn.getArabic()
      control.setReturnValue(5);
      // switch from record to playback
      control.replay();
      // now it's ready to use:
      assertEquals(5, rn.getArabic());
   }
}
```

Readings



1. ***JUnit*, http://www.junit.org/**
2. ***Test Infected – Programmers love writing tests*, http://junit.sourceforge.net/doc/testinfected/testing.htm**
3. ***JUnit Cook's Tour*, http://junit.sourceforge.net/doc/cookstour/cookstour.htm**
4. ***J. Rainsberger „JUnit Recipes"***
5. ***T. Mackinnon et al. „Endo Testing – Unit testing with Mock Objects"***
6. ***A. Shneider*, http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html**

Q&A