Modulo II
JUNIT

Prof. Ismael H F Santos

Bibliografia

- *JUnit in Action*

Agenda:

- JUNIT 3.8
- JUNIT 4

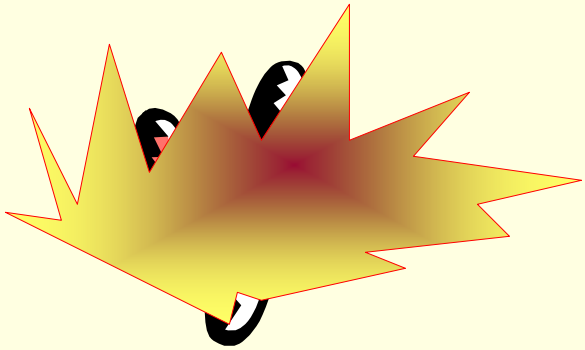
MA-JUNIT

Introdução

Teste Unitário

Imagine se um avião só fosse testado após a conclusão de sua construção....

Seria um desastre....



Teste Unitário

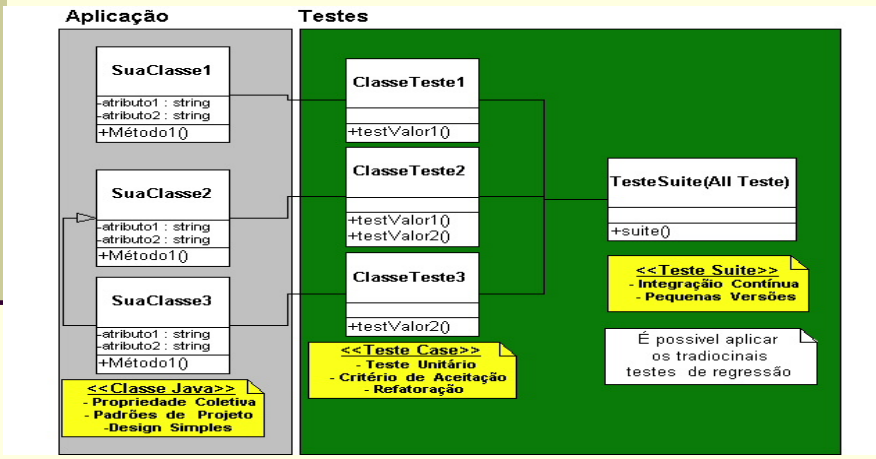
- O que é ?
  - O **teste unitário** é uma modalidade de testes que se concentra na verificação da menor unidade do projeto de software. É realizado o teste de uma **unidade lógica**, com uso de dados suficientes para se testar apenas a lógica da unidade em questão.
  - Em sistemas construídos com uso de linguagens orientadas a objetos, essa unidade pode ser identificada como um **método**, uma **classe** ou mesmo um **objeto**.

Teste Unitário

- **Por que ?**
  - **Previne** contra o aparecimento de “BUG’S” oriundos de códigos mal escritos.
  - Código testado é mais confiável.
  - Permite alterações sem medo(**coragem**)
  - Testa situações de sucesso e de falha.
  - Resulta em outras práticas XP como : **Código coletivo**, **refatoração**, **integração contínua.**
  - Serve como métrica do projeto (*teste* **==***requisitos*)
  - Gera e preserva um “conhecimento” sobre o projeto.

Teste Unitário

- **Organização dos testes e práticas XP**



Teste Unitário

- **Quando fazer?**
  - **No início - TDD**
    Primeiro projetar e escrever as classes de testes, **depois** as classes com regra de negócios
  - **Diariamente**
    È SUGERIDO que seja rodado os testes várias vezes ao dia (é fácil corrigir **pequenos** problemas do que corrigir um "**problemão**" somente no final do projeto.

Teste Unitário

- **Quem faz?**
  - **Test Case**(para cada classe)
    Desenvolvedor(Projeta, escreve e roda)
  - **Test Suite**(Rodas vários test cases)
    Coordenador e Desenvolvedor (Projeta, escreve e roda)

* *Teste de aceitação(homologação) é feito junto ao cliente.*

Teste Unitário

- **O Que Testar?**
  - A principal regra para saber o que testar é: “Tenha **criatividade** para imaginar as **possibilidades** de testes”.
  - Comece pelas mais simples e deixe os testes “complexos“ para o final.
  - Use apenas dados suficientes (não teste 10 condições se três forem suficientes)
  - Não teste métodos triviais, tipo get e set.
  - No caso de um método set, só faça o teste caso haja validação de dados.
  - Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)

MA-JUNIT

Introdução

JUnit

- A unit test framework for Java
  - Authors: Erich Gamma, Kent Beck
- Objective:
  - "If tests are simple to create and execute, then programmers will be more inclined to create and execute tests."

JUnit – O que é?

- Um **framework** que facilita o desenvolvimento e execução de **testes de unidade** em código Java
  - Uma API para construir os testes
  - Aplicações para executar testes
- A API
  - Classes Test, TestCase, TestSuite, etc. oferecem a infraestrutura necessária para criar os testes
  - Métodos assertTrue(), assertEquals(), fail(), etc. são usados para testar os resultados
- Aplicação TestRunner
  - Roda testes individuais e suites de testes
  - Versões texto, Swing e AWT
  - Apresenta diagnóstico sucesso/falha e detalhes

Para que serve?

- 'Padrão' para testes de unidade em Java
  - Desenvolvido por Kent Beck (XP) e Erich Gamma (GoF)
  - Design muito simples
- Testar é uma boa prática, mas é chato; JUnit torna as coisas mais agradáveis, facilitando
  - A criação e execução automática de testes
  - A apresentação dos resultados
- JUnit pode verificar se cada unidade de código funciona da forma esperada
  - Permite agrupar e rodar vários testes ao mesmo tempo
  - Na falha, mostra a causa em cada teste
- Serve de base para extensões

Introduction

- What do we need to do automated testing?
  - Test script
    - Actions to send to system under test (SUT).
    - Responses expected from SUT.
    - How to determine whether a test was successful or not?
  - Test execution system
    - Mechanism to read test scripts, and connect test case to SUT.
    - Keeps track of test results.

Test case verdicts

- A verdict is the declared result of executing a single test.
- Pass: the test case achieved its intended purpose, and the software under test performed as expected.
- Fail: the test case achieved its intended purpose, but the software under test did not perform as expected.
- Error: the test case did not achieve its intended purpose.
  - Potential reasons:
    - An unexpected event occurred during the test case.
    - The test case could not be set up properly

A note on JUnit versions...

- The version is 4.4, available from 2008
  - To use JUnit 4.x, you must use Java version 5 or 6
- JUnit 4, introduced April 2006, is a significant (i.e. not compatible) change from prior versions.
- JUnit 4 is used in this presentation.
- Much of the JUnit documentation and examples currently available are for JUnit 3, which is slightly different.
  - JUnit 3 can be used with earlier versions of Java (such as 1.4.2).
  - The junit.org web site shows JUnit version 4 unless you ask for the old version.
  - Eclipse (3.2) gives the option of using JUnit 3.8 or JUnit 4.1, which are both packaged within Eclipse.

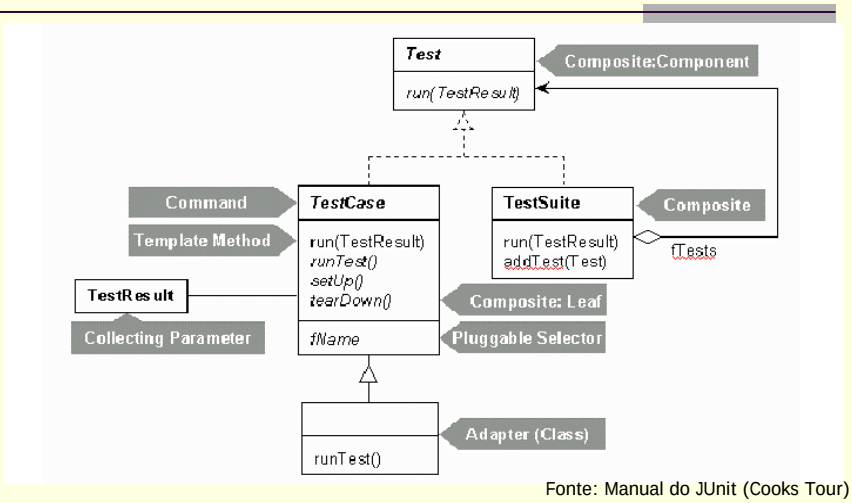
What is a JUnit Test?

- A test "script" is just a collection of Java methods.
  - General idea is to create a few Java objects, do something interesting with them, and then determine if the objects have the correct properties.
- What is added? Assertions.
  - A package of methods that checks for various properties:
    - "equality" of objects
    - identical object references
    - null / non-null object references
  - The assertions are used to determine the test case verdict.

When is JUnit appropriate?

- As the name implies…
  - for unit testing of small amounts of code
- On its own, it is not intended for complex testing, system testing, etc.
- In the test-driven development methodology, a JUnit test should be written first (before any code), and executed.
  - Then, implementation code should be written that would be the minimum code required to get the test to pass – and no extra functionality.
  - Once the code is written, re-execute the test and it should pass.
  - Every time new code is added, re-execute all tests again to be sure nothing gets broken.

JUnit- Arquitetura das Classes

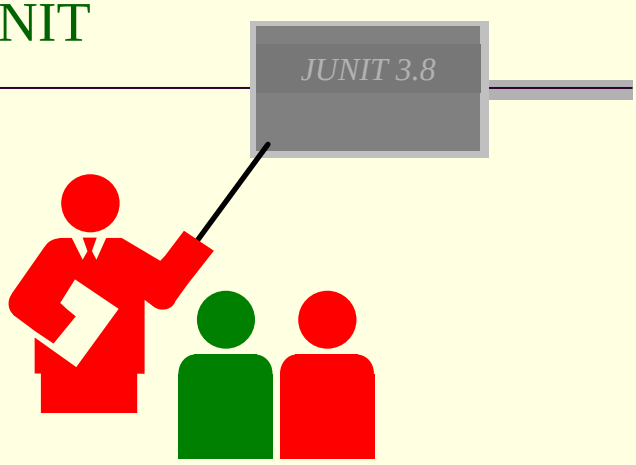


Fonte: Manual do JUnit (Cooks Tour)

Como usar o JUnit?

- Há várias formas de usar o JUnit. Depende da metodologia de testes que está sendo usada
  - Código existente:
    - precisa-se escrever testes para classes que já foram implementadas
  - Desenvolvimento guiado por testes (TDD):
    - código novo só é escrito se houver um teste sem funcionar
  - Onde obter?
    - www.junit.org
  - Como instalar?
    - Incluir o arquivo junit.jar no classpath para compilar e rodar os programas de teste

MA-JUNIT



JUnit para testar código existente

Exemplo de um roteiro típico

1. *Crie uma classe que estenda junit.framework.TestCase para cada classe a ser testada*

```
import junit.framework.*;
class SuaClasseTest extends TestCase {...}
```

2. *Para cada método xxx(args) a ser testado defina um método public void testXxx() no test case*
   - *SuaClasse:*
     - `public boolean equals(Object o) { ... }`
   - *SuaClasseTest:*
     - `public void testEquals() {...}`
- *Sobreponha o método setUp(), se necessário*
- *Sobreponha o método tearDown(), se necessário*

JUnit 3.8 – Como implementar

- *Dentro de cada teste, utilize os métodos herdados da classe TestCase*
  - *assertEquals(objetoEsperado, objetoRecebido),*
  - *assertTrue(valorBooleano), assertNotNull(objeto)*
  - *assertSame(objetoUm, objetoDois), fail (), ...*
- *Exemplo de test case com um setUp() e um teste:*

```
public class CoisaTest extends TestCase {
  // construtor padrão omitido
  private Coisa coisa;
  public void setUp() { coisa = new Coisa("Bit"); }
  public void testToString() {
    assertEquals("<coisa>Bit</coisa>",
                 coisa.toString());
  }
}
```


JUnit 3.8 – Como implementar

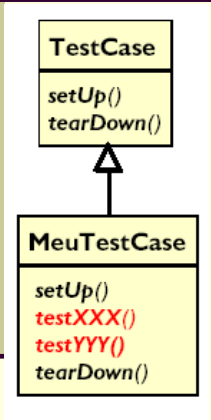
- 1. Crie uma classe que estenda **junit.framework.TestCase** para cada classe a ser testada

  import junit.framework.*;
  class SuaClasseTest extends TestCase
  {...
  }
- 2. Para cada método a ser testado defina um método **public void test???()** no test case
  - SuaClasse:

    public int Soma(Object o ...)
    { ...
    }
  - SuaClasseTest:

    public void testSoma()

JUnit 3.8 – Funcionamento



- O TestRunner recebe uma subclasse de junit.framework.TestCase
  - Usa reflection (Cap 14) para achar métodos
- Cada método testXXX(), executa:
  - *1. o método setUp()* */* Opcional */*
  - *2. o próprio método testXXX()*
  - *3. o método tearDown()* */* Opcional */*
- O test case é instanciado para executar um método testXXX() de cada vez.
  - As alterações que ele fizer ao estado do objeto não afetarão os demais testes
- Método pode terminar, falhar ou provocar exceção

JUnit – Analisando o Resultado

- **Em modo gráfico, os métodos testados podem apresentar o seguintes resultados:**
- ***Sucesso***
- ***Falha***
- ***exceção***

Exemplo: um test case

```
package junitdemo;

import junit.framework.*;
import java.io.IOException;

public class TextUtilsTest extends TestCase {

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

    public void testRemoveWhiteSpaces() throws IOException {
        String testString = "one, ( two | three+) ,       "+
                            "(((four+ |\t  five)?\n \n,  six?";
        String expectedString = "one,(two|three+)"+
                                ",(((four+|five)?,six?";
        String results = TextUtils.removeWhiteSpaces(testString);
        assertEquals(expectedString, results);
    }
}
```

Construtor precisa ser publico, receber String name e chamar `super(String name)` (JUnit 3.7 ou anterior)

*Método começa com "**test**" e é sempre **public void***

Exemplo: uma classe que faz o teste passar

```
package junitdemo;
import java.io.*;

public class TextUtils {

  public static String removeWhiteSpaces(String text)
                                          throws IOException {
      return "one,(two|three+),(((four+|five)?,six?";
  }
}
```

- *O teste passa... e daí? A solução está pronta? Não! Tem dados duplicados! Remova-os!*
- *Escreva um novo teste que faça com que esta solução falhe, por exemplo:*

```
String test2 = " a   b\nc   ";
assertEquals("abc", TextUtils.removeWhiteSpaces(test2));
```

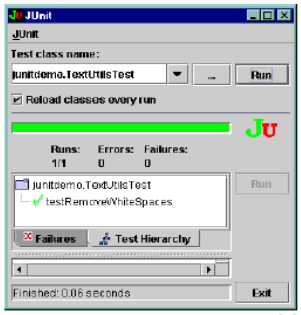
Exemplo: Outra classe que faz o teste passar

```
package junitdemo;
import java.io.*;

public class TextUtils {

  public static String removeWhiteSpaces(String text)
                                          throws IOException {
    StringReader reader = new StringReader(text);
    StringBuffer buffer = new StringBuffer(text.length());
    int c;
    while( (c = reader.read()) != -1) {
      if (c ==' '||c =='\n'||c =='\r'|| c =='\f'||c =='\t') {
          ; /* do nothing */
      } else {
         buffer.append((char)c);
      }
    }
    return buffer.toString();
  }
}
```


Exemplo: como executar

- *Use a interface de texto*
  - *java -cp junit.jar junit.textui.TestRunner junitdemo.TextUtilsTest*
- *Ou use a interface gráfica*
  - *java -cp junit.jar junit.swingui.TestRunner junitdemo.TextUtilsTest*
- *Use Ant <junit>*
  - *tarefa do Apache Ant*
- *Ou forneça um main():*

```
public static void main (String[] args) {
    TestSuite suite =
      new TestSuite(TextUtilsTest.class);
    junit.textui.TestRunner.run(suite);
}
```



TestSuite

- *Permite executar uma coleção de testes*
  - *Método addTest(TestSuite) adiciona um teste na lista*
- *Padrão de codificação (usando reflection):*
  - *retornar um TestSuite em cada test-case:*

```
public static TestSuite suite() {
    return new TestSuite(SuaClasseTest.class);
}
```

  - *criar uma classe AllTests que combina as suites:*

```
public class AllTests {
    public static Test suite() {
        TestSuite testSuite =
                        new TestSuite("Roda tudo");
        testSuite.addTest(pacote.AllTests.suite());
        testSuite.addTest(MinhaClasseTest.suite());
        testSuite.addTest(SuaClasseTest.suite());
        return testSuite;
    }
}
```

Pode incluir outras suites

Fixtures

- *São os dados reutilizados por vários testes*

```
public class AttributeEnumerationTest extends TestCase {
    String   testString;
    String[] testArray;
    AttributeEnumeration testEnum;
    public void setUp() {
        testString  = "(alpha|beta|gamma)";
        testArray = new String[]{"alpha", "beta", "gamma"};
        testEnum = new AttributeEnumeration(testArray);
    }
    public void testGetNames() {
        assertEquals(testEnum.getNames(), testArray);
    }
    public void testToString() {
        assertEquals(testEnum.toString(), testString);
    }
(...)
```

Fixture

- *Se os mesmos dados são usados em vários testes, inicialize-os no setUp() e faça a faxina no tearDown() (se necessário)*
- *Não perca tempo pensando nisto antes. Escreva seus testes. Depois, se achar que há duplicação, monte o fixture.*

Teste situações de falha

- *É tão importante testar o cenário de falha do seu codigo quanto o sucesso*
- *Método fail() provoca uma falha*
  - *Use para verificar se exceções ocorrem quando se espera que elas ocorram*
- *Exemplo*

```
public void testEntityNotFoundException() {
   resetEntityTable(); // no entities to resolve!
   try {
      // Following method call must cause exception!
      ParameterEntityTag tag = parser.resolveEntity("bogus");
      fail("Should have caused EntityNotFoundException!");
   } catch (EntityNotFoundException e) {
       // success: exception occurred as expected
   }
}
```


JUnit vs. asserções

- *Afirmações do J2SDK 1.4 são usadas dentro do código*
  - *Podem incluir testes dentro da lógica procedural de um programa*

```
if (i%3 == 0) {
   doThis();
} else if (i%3 == 1) {
   doThat();
} else {
   assert i%3 == 2: "Erro interno!";
}
```

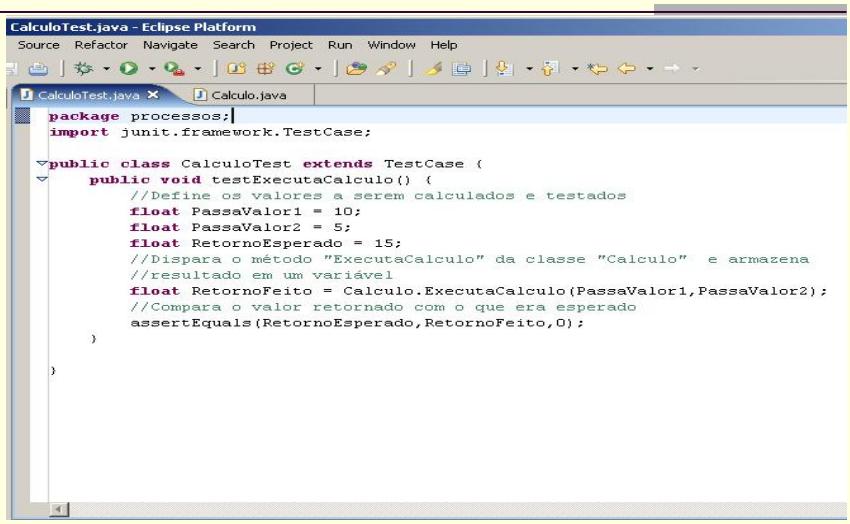
  - *Provocam um AssertionError quando falham (que pode ser encapsulado pelas exceções do JUnit)*
- *Afirmações do JUnit são usadas em classe separada (TestCase)*
  - *Não têm acesso ao interior dos métodos (verificam se a interface dos métodos funciona como esperado)*
- *Afirmações do J2SDK1.4 e JUnit são complementares*
  - *JUnit testa a interface dos métodos*
  - *assert testa trechos de lógica dentro dos métodos*

Limitações do JUnit

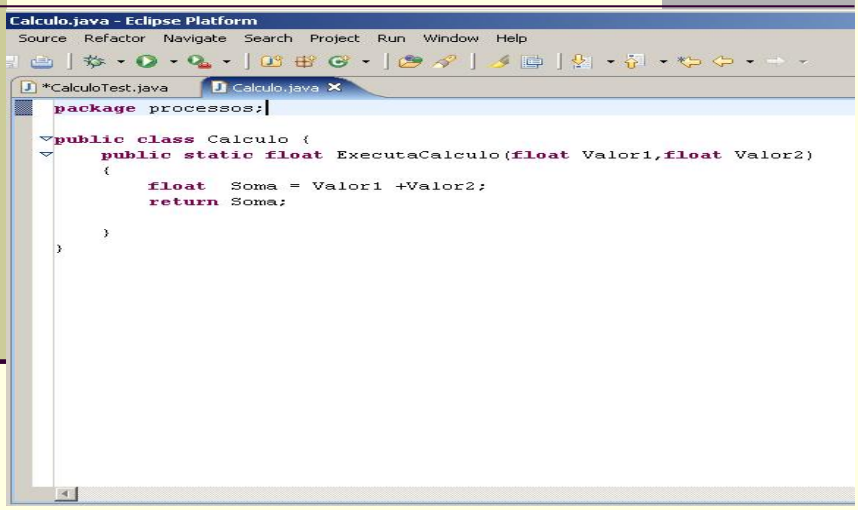
- Acesso aos dados de métodos sob teste
  - Métodos private e variáveis locais não podem ser testadas com JUnit.
  - Dados devem ser pelo menos package-private (friendly)
- Soluções com refatoramento
  - Isolar em métodos private apenas código inquebrável
  - Transformar métodos private em package-private
    - Desvantagem: quebra ou redução do encapsulamento
    - Classes de teste devem estar no mesmo pacote que as classes testadas para ter acesso
- Solução usando extensão do JUnit (open-source)
  - **JUnitX**: usa reflection para ter acesso a dados private
  - http://www.extreme-java.de/junitx/index.html

Usando Eclipse .....

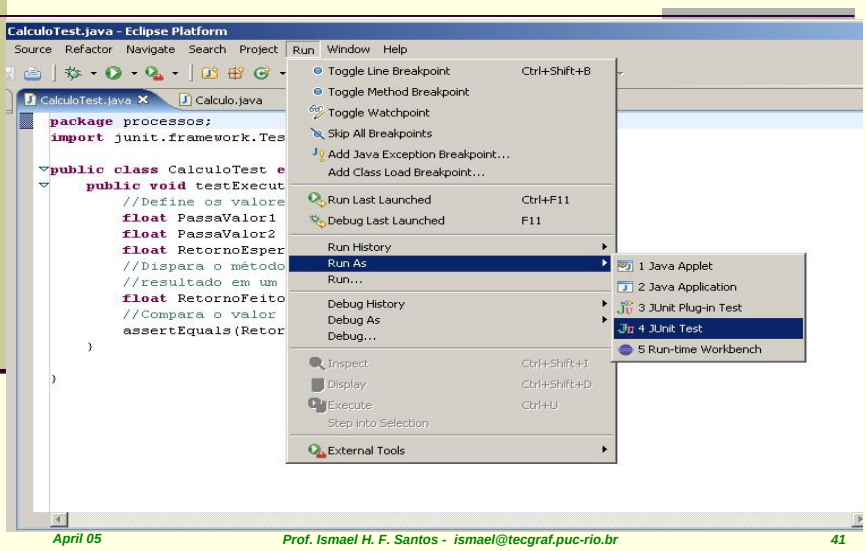
Criando a classe de teste no Eclipse



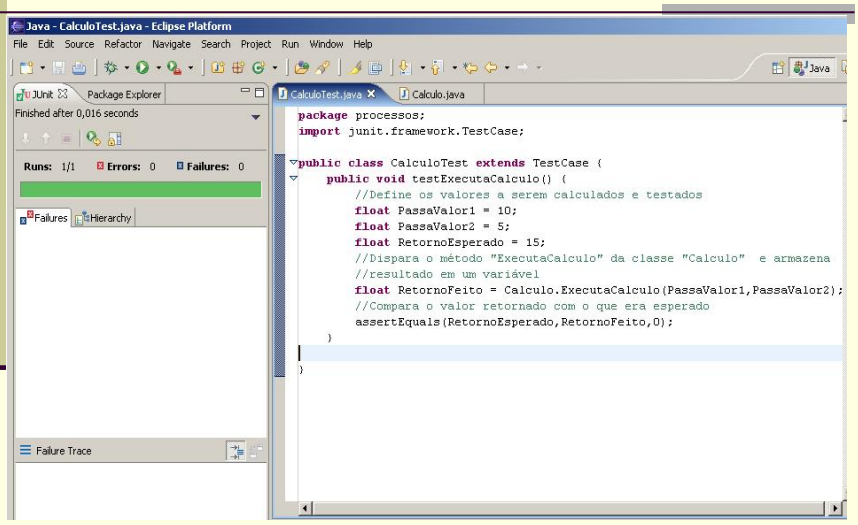
Sua Classe a ser testada



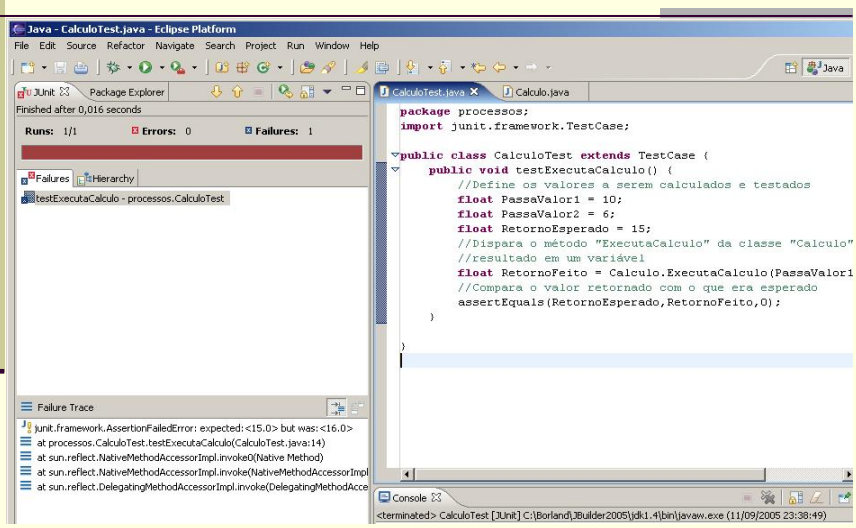
Rodando o teste em modo gráfico



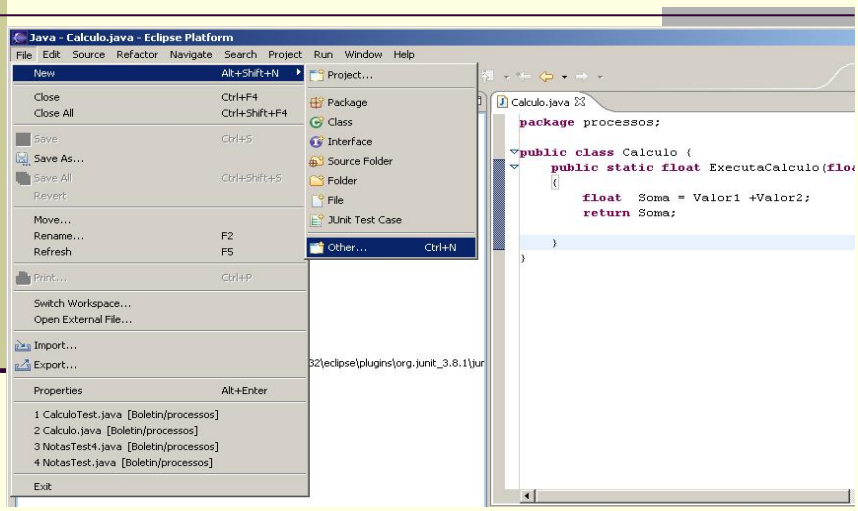
Resultado em caso de SUCESSO



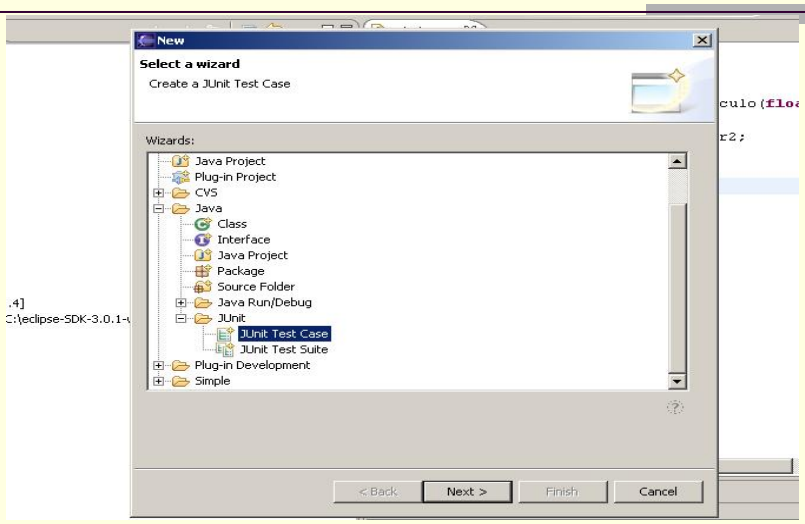
Resultado em caso de FALHA



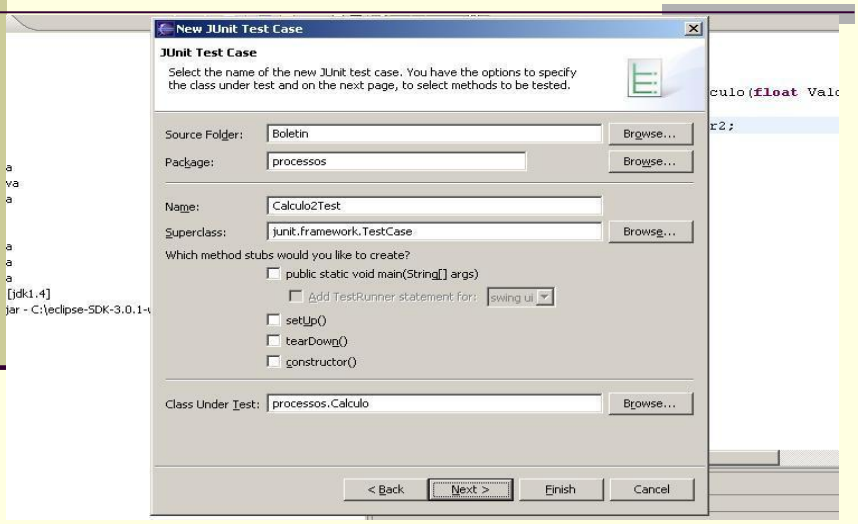
Automatizando criação dos TestCases



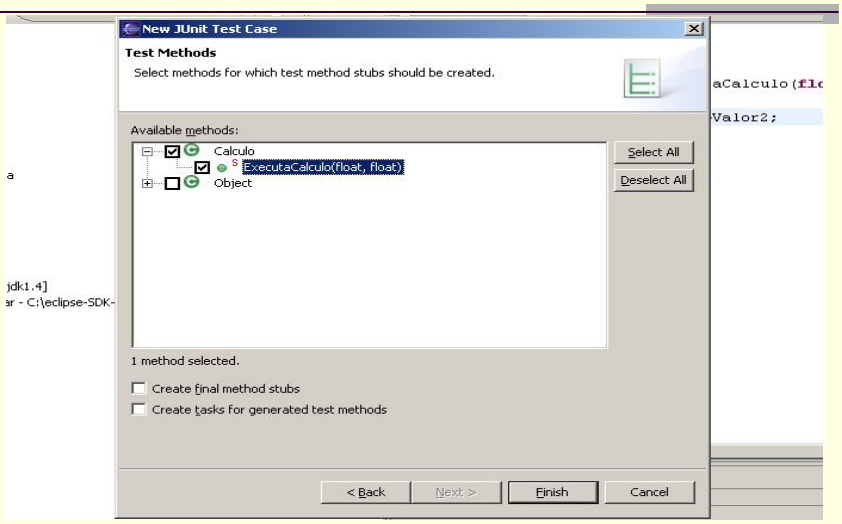
Automatizando criação dos TestCases



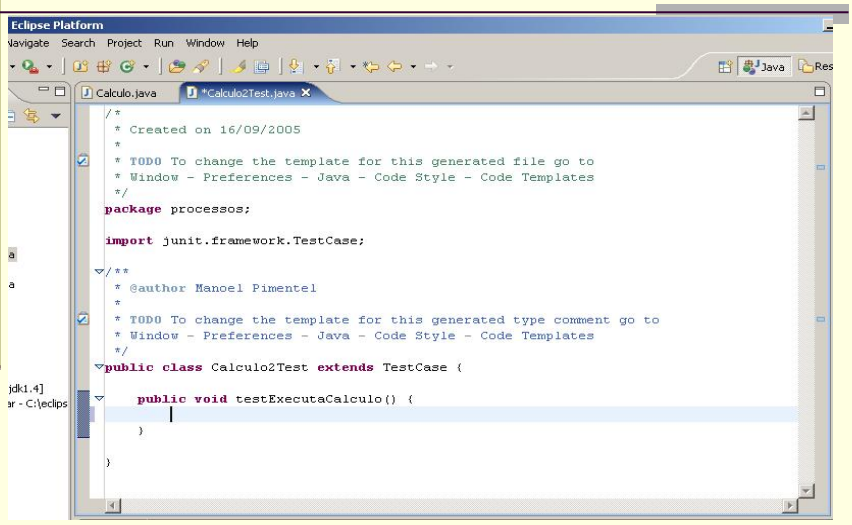
Automatizando criação dos TestCases



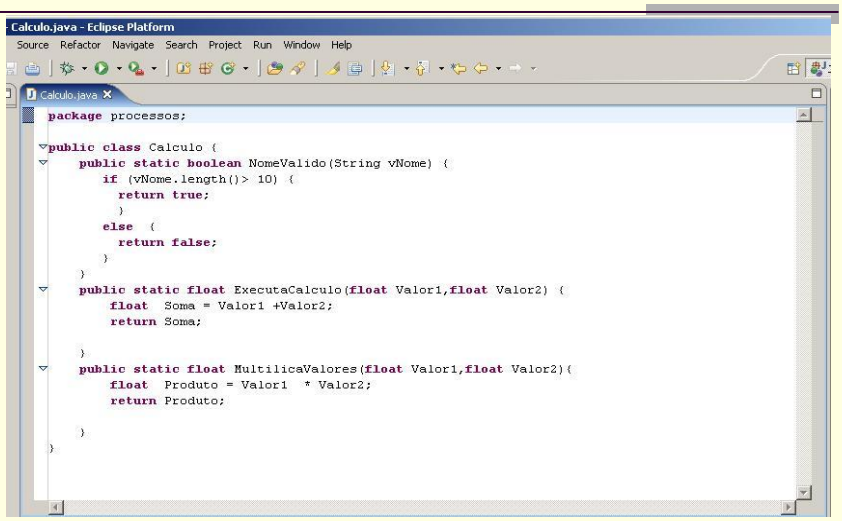
Automatizando criação dos TestCases



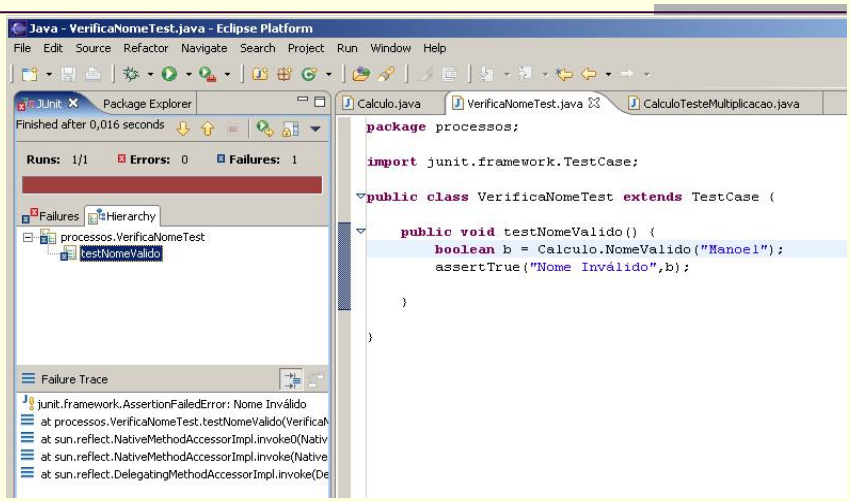
Automatizando criação dos TestCases



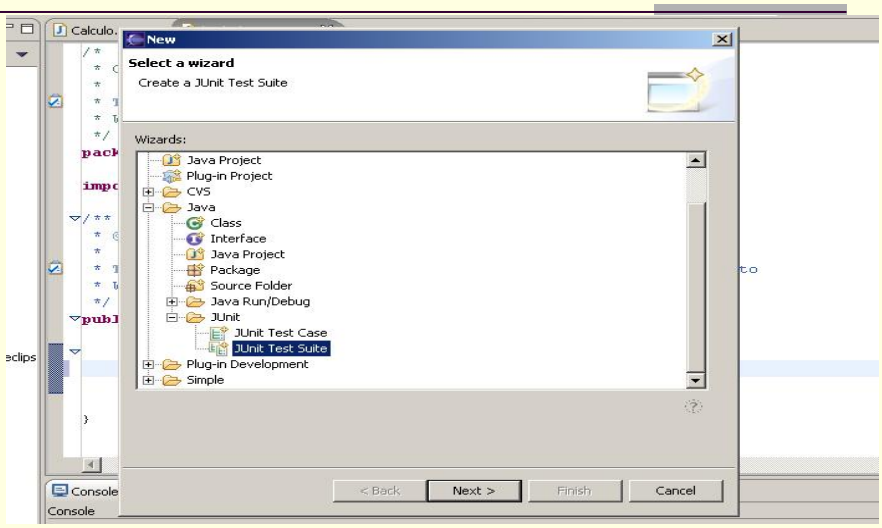
Outros Exemplo de teste



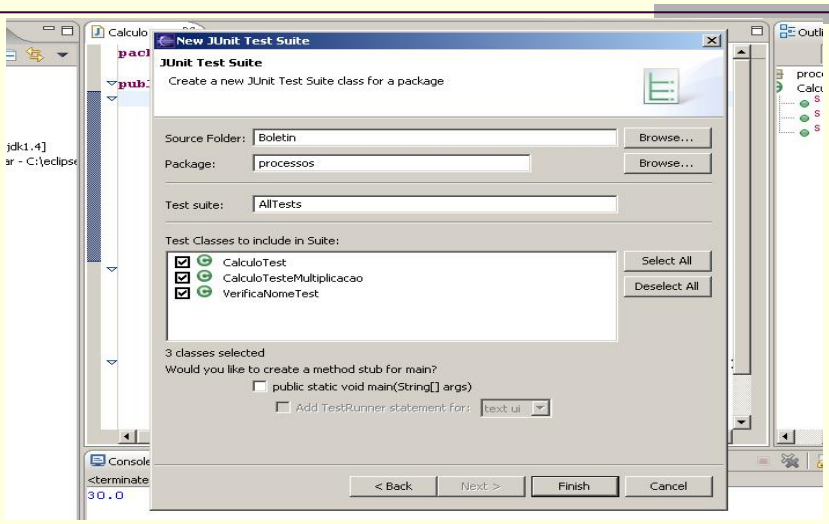
Outros Exemplo de teste



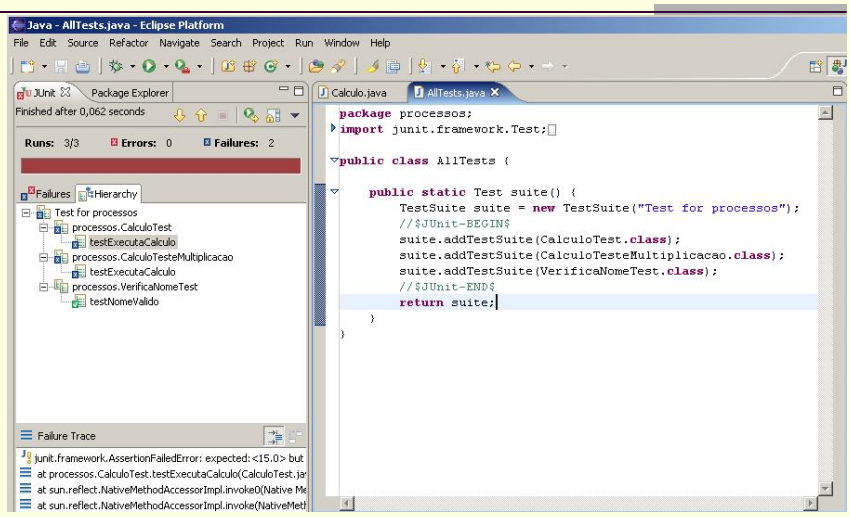
Criando Test Suite para rodar vários test cases



Criando Test Suite para rodar vários test cases



Criando Test Suite para rodar vários test cases



JUnit 3.8 - Outros **Métodos** de **Testes**

- **assertEquals**
  - Testa igualdade entre dois objetos(esperado x retornado)
- **assertFalse( )**
  - Testa Retorno booleano FALSO
- **assertTrue( )**
  - Testa Retorno booleano VERDADEIRO
- **assertNotNull( )**
  - Testa se um valor de um objeto NÃO está NULO
- **assertNull( )**
  - Testa se um valor de um objeto está NULO

JUnit 3.8 – métodos **setUp()** e **tearDown()**

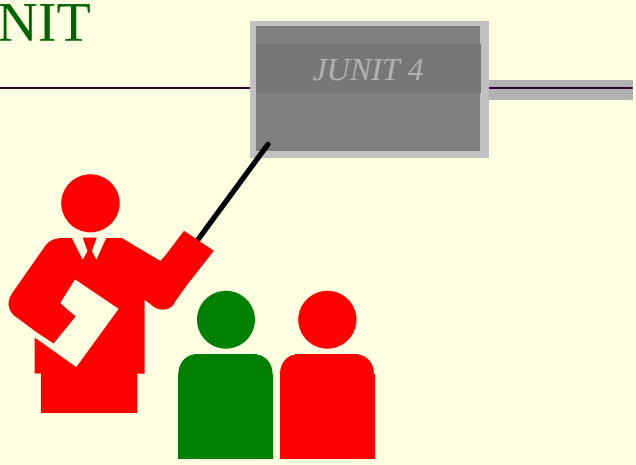
- São os dados **reutilizados** por vários testes, **Inicializados** no setUp() e **destruídos** no tearDown() (se necessário)

```
package processos;
import junit.framework.TestCase;
public class CalculoVariosTest extends TestCase {
    float PassaValor1;
    float PassaValor2;
    protected void setUp(){
          PassaValor1 = 10;
          PassaValor2 = 5;  }
    public void testExecutaCalculo() {
        float RetornoEsperado = 15;
        float RetornoFeito = Calculo.ExecutaCalculo(PassaValor1,PassaValor2);
        assertEquals(RetornoEsperado,RetornoFeito,0);   }
    public void testMultiplicaValores() {
        float RetornoEsperado = 60;
        float RetornoFeito = Calculo.MultiplicaValores(PassaValor1,PassaValor2);
        assertEquals(RetornoEsperado,RetornoFeito,0);}
}
```


Técnicas complementares

- É importante também, ser aplicado tipos de testes como:
  - Teste de Performance,
  - Teste de Carga,
  - Teste de estresse,
  - Teste de aceitação, etc.

MA-JUNIT



A JUnit 4 Test Case

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
        Value v1 = new Value( );

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
}
```

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
        Value v1 = new Value( );

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
}
```

Identifies this Java method as a test case, for the test runner

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
        Value v1 = new Value();

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName();

        Assert.assertEquals( expected, actual );
}
```

Objective: confirm that `setName` saves the specified name in the `Value` object

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
        Value v1 = new Value();

        v1.setName( "Y" );

        String expected = "Y"
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
}
```

Check to see that the **`Value`** object really did store the name

A JUnit 4 Test Case

```
/** Test of setName() method, of class Value  */

@Test
public void createAndSetName()
{
        Value v1 = new Value( );

        v1.setName( "Y" );

        String expected = "Y";
        String actual = v1.getName( );

        Assert.assertEquals( expected, actual );
}
```

We want **`expected`** and **`actual`** to be equal.

If they aren't, then the test case should fail.

Assertions

- Assertions are defined in the JUnit class **`Assert`**
  - If an assertion is true, the method continues executing.
  - If any assertion is false, the method stops executing at that point, and the result for the test case will be fail.
  - If any other exception is thrown during the method, the result for the test case will be error.
  - If no assertions were violated for the entire method, the test case will pass.
- All assertion methods are static methods

Assertion methods (1)

- Boolean conditions are true or false
  `assertTrue(condition)`
  `assertFalse(condition)`
- Objects are null or non-null
  `assertNull(object)`
  `assertNotNull(object)`
- Objects are identical (i.e. two references to the same object), or not identical.
  `assertSame(expected, actual)`
  - true if: **`expected == actual`**

  `assertNotSame(expected, actual)`

Assertion methods (2)

- "Equality" of objects:
  `assertEquals(expected, actual)`
  - valid if: `expected.equals( actual )`
- "Equality" of arrays:
  `assertArrayEquals(expected, actual)`
  - arrays must have same length
  - for each valid value for `i`, check as appropriate:
    `assertEquals(expected[i],actual[i])`
    or
    `assertArrayEquals(expected[i],actual[i])`
- There is also an unconditional failure assertion `fail()` that always results in a fail verdict.

Assertion method parameters

- In any assertion method with two parameters, the first parameter is the expected value, and the second parameter should be the actual value.
  - This does not affect the comparison, but this ordering is assumed for creating the failure message to the user.
- Any assertion method can have an additional `String` parameter as the first parameter. The string will be included in the failure message if the assertion fails.
  - Examples:
    ```
    fail( message )
    assertEquals( message, expected, actual)
    ```

Equality assertions

- `assertEquals(a,b)` relies on the `equals()` method of the class under test.
  - The effect is to evaluate `a.equals( b ).`
  - It is up to the class under test to determine a suitable equality relation. JUnit uses whatever is available.
  - Any class under test that does not override the `equals()` method from class `Object` will get the default `equals()` behaviour – that is, object identity.
- If `a` and `b` are of a primitive type such as `int`, `boolean`, etc., then the following is done for `assertEquals(a,b)` :
  - `a` and `b` are converted to their equivalent object type (`Integer`, `Boolean`, etc.), and then `a.equals( b )` is evaluated.

New assertions

- JUnit 4 adds two assert() methods for comparing arrays.
- These methods compare arrays in the most obvious way: Two arrays are equal if they have the same length and each element is equal to the corresponding element in the other array; otherwise, they're not. The case of one or both arrays being null is also handled.

```
public static void
assertEquals(Object[] expected, Object[] actual);

public static void
assertEquals(String message, Object[] expected,
                             Object[] actual);
```

Floating point assertions

- When comparing floating point types (`double` or `float`), there is an additional required parameter `delta`.
- The assertion evaluates

  `Math.abs( expected – actual ) <= delta`

  to avoid problems with round-off errors with floating point comparisons.
- Example:

```
assertEquals( aDouble, anotherDouble, 0.0001 )
```


Organization of JUnit tests

- Each method represents a single test case that can independently have a verdict (pass, error, fail).
- Normally, all the tests for one Java class are grouped together into a separate class.
  - Naming convention:
    - Class to be tested: `Value`
    - Class containing tests: `ValueTest`

Running JUnit Tests (1)

- The JUnit framework does not provide a graphical test runner. Instead, it provides an API that can be used by IDEs to run test cases and a textual runner than can be used from a command line.
- Eclipse and Netbeans each provide a graphical test runner that is integrated into their respective environments.

Running JUnit tests (2)

- With the runner provided by JUnit:
  - When a class is selected for execution, all the test case methods in the class will be run.
  - The order in which the methods in the class are called (i.e. the order of test case execution) is not predictable.
- Test runners provided by IDEs may allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests with are independent of execution order, and that are without dependencies on the state any previous test(s).

Test fixtures

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: "setup" and "teardown".

Setup and Teardown

- For a collection of tests for a particular class, there are often some repeated tasks that must be done prior to each test case.
  - Examples: create some "interesting" objects to work with, open a network connection, etc.
- Likewise, at the end of each test case, there may be repeated tasks to clean up after test execution.
  - Ensures resources are released, test system is in known state for next test case, etc.
  - Since a test case failure ends execution of a test method at that point, code to clean up cannot be at the end of the method.

Setup and Teardown

- Setup:
  - Use the **@Before** annotation on a method containing code to run before each test case.
- Teardown (regardless of the verdict):
  - Use the **@After** annotation on a method containing code to run after each test case.
  - These methods will run even if exceptions are thrown in the test case or an assertion fails.
- It is allowed to have any number of these annotations.
  - All methods annotated with **@Before** will be run before each test case, but they may be run in any order.

Example: Using a file as a text fixture

```
public class OutputTest
{
    private File output;

    @Before public void createOutputFile() {
       output = new File(...);
    }

    @After public void deleteOutputFile() {
        output.delete();
    }

    @Test public void test1WithFile() {
       // code for test case objective
    }

    @Test public void test2WithFile() {
       // code for test case objective
    }
}
```

Method execution order

1. `createOutputFile()`
2. `test1WithFile()`
3. `deleteOutputFile()`
4. `createOutputFile()`
5. `test2WithFile()`
6. `deleteOutputFile()`

Assumption: `test1WithFile` runs before `test2WithFile`– which is not guaranteed.

Once-only setup

- It is also possible to run a method once only for the entire test class, before any of the tests are executed, and prior to any `@Before` method(s).
- Useful for starting servers, opening communications, etc. that are time-consuming to close and re-open for each test.
- Indicate with `@BeforeClass` annotation (can only be used on one method, which must be static):

```
@BeforeClass public static void anyNameHere() {
    // class setup code here
}
```

Once-only tear down

- A corresponding once-only cleanup method is also available. It is run after all test case methods in the class have been executed, and after any `@After` methods
- Useful for stopping servers, closing communication links, etc.
- Indicate with `@AfterClass` annotation (can only be used on one method, which must be static):

```
@AfterClass public static void anyNameHere(){
    // class cleanup code here
}
```


Example

```
// This class tests a lot of error conditions, which
// Xalan annoyingly logs to System.err.This hides System.err
// before each test and restores it after each test.

private PrintStream systemErr;

@BeforeClass protected void redirectStderr() {
  systemErr = System.err; // Hold on to the original value
  System.setErr(new PrintStream(new
   ByteArrayOutputStream()));
}

@AfterClass protected void tearDown() {
  // restore the original value
  System.setErr(systemErr);
}
```

Exception testing (1)

- Add parameter to `@Test` annotation, indicating that a particular class of exception is expected to occur during the test.

```
@Test(expected=ExceptedTypeOfException.class)
public void testException()
{
    exceptionCausingMethod();
}
```

- If no exception is thrown, or an unexpected exception occurs, the test will fail.
  - That is, reaching the end of the method with no exception will cause a test case failure.
- Testing contents of the exception message, or limiting the scope of where the exception is expected requires using the approach on the next slide.

Exception testing in JUNIT 3.8

- Catch exception, and use `fail( )` if not thrown

```
public void testException()
{
   try
   {
      exceptionCausingMethod();

      // If this point is reached, the expected
      // exception was not thrown.

      fail("Exception should have occurred");
   }
   catch ( ExceptedTypeOfException exc )
   {
      String expected = "A suitable error message";
      String actual = exc.getMessage();
      Assert.assertEquals( expected, actual );
   }
}
```

Ignoring a test

- The @Ignore annotation tells the runner to ignore the test and report that it was not run. You can pass in a string as a parameter to @Ignore annotation that explains why the test was ignored.
- E.g. The new JUnit will not run a test method annotated with @Ignore("Database is down") but will only report it.

```
// Java doesn't yet support the UTF-32BE and UTF32LE encodings
  @Ignore("Java doesn't yet support the UTF-32BE")
  @Test public void testUTF32BE()
  throws ParsingException, IOException, XIncludeException {
    File input = new File("data/xinclude/input/UTF32BE.xml");
    Document doc = builder.build(input);
    Document result = XIncluder.resolve(doc);
    Document expectedResult = builder.build(
                new File(outputDir, "UTF32BE.xml"));
    assertEquals(expectedResult, result);
  }
```


Timing out a test

- You can pass in a timeout parameter to the test annotation to specify the timeout period in milliseconds. If the test takes more, it fails.
- E.g. A method annotated with @Test (timeout=10) fails if it takes more than 10 milliseconds.

```
@Test(timeout=500)
public void retrieveAllElementsInDocument() {
     doc.query("//*");
}
```

Compatibility between JUNIT3.8 e 4

- JUnit4Adapter enables compatibility with the old runners so that the new JUnit 4 tests can be run with the old runners. The suite method in the diagram above illustrates the use of JUnit4Adapter.

```
package example.junitold;

import junit.framework.TestCase;

public class LibraryTest extends TestCase {   1. Class must inherit from TestCase

    public void testBookAvailableInLibrary(){   2. Old JUnit requires Test methods
        Library library = new Library();            to be prefixed with 'test'
        boolean result =
                library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                     true,        3. Use one of the several assert methods available
                     result);
    }
}
```


Running JUnit4 tests with JUnit3.8 runner

- Use a normal class and not extend from junit.framework.TestCase.
- Use the Test annotation to mark a method as a test method. To use the Test annotation import org.junit.Test
- Run the test using JUnit4TestAdapter. If you want to learn more about JUnit4TestAdapter, keep reading ahead.
- Alternatively, you can use the JUnitCore class in the org.junit.runner package. JUnit 4 runner can also run tests written using the old JUnit. To run the tests using the JUnitCore class via the command line, type:

  java org.junit.runner.JUnitCore LibraryTest

Running JUnit4 tests with JUnit3.8 runner

```
package example.junit4;

import org.junit.Test;  1. Import Test annotation
import static org.junit.Assert.assertEquals;  2. Import static assertEquals
import junit.framework.JUnit4TestAdapter;  3. Import JUnit4TestAdapter

public class LibraryTest{

    @Test public void bookAvailableInLibrary(){
        Library library = new Library();
        boolean result = library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                true,
                result);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTest.class);
    }
}
```

4. To declare a method as a test method use the '@Test' annotation

5. Use one of the assert methods

6. JUnit4TestAdapter is required to run JUnit4 tests with the old Junit runner

Exercícios:

- A classe *Valores* é capaz de armazenar até 10 valores inteiros positivos (v > 0). Esta classe deve implementar a seguinte interface

```
interface ValoresITF{
      boolean ins(int v);//insere um valor
      int del(int i);    // remove/retorna valor indice i
      int size();        // retorna qtdade valores armazenados
      double mean();     // retorna média valores armazenados
      int greater();     // retorna maior valor armazenado
      int lower();       //retorna o menor valor armazenado
}
```

  - O método ins retorna true se o valor pode ser inserido
  - Os método del retorna o valor removido ou -1 se a lista está vazia
  - O método mean retorna 0 se a lista está vazia
  - Os métodos greater e lower retornam -1 se a lista está vazia

- Determine um conjunto de casos de teste para esta classe.
- Defina uma classe de teste para a classe *Valores*.
- Implemente a classe *Valores*.
- Verifique a cobertura dos testes incrementando-os para garantir cobertura de condição quando for o caso.
- Execute os testes e verifique os resultados.

Solução

```
import junit.framework.TestCase;

public class testValores extends TestCase {
   private Valores val;
      protected void setUp() throws Exception {
                super.setUp();
                val = new Valores();
                val.ins(5);
                val.ins(12);
                val.ins(1);
                val.ins(30);
                val.ins(152);
                val.ins(6);
      }
      public void testIns() {
                assertEquals(false,val.ins(-10));
                assertEquals(false,val.ins(0));
                val.ins(2);
                assertEquals(7,val.size());
                val.ins(3);
                assertEquals(8,val.size());
                val.ins(4);
                assertEquals(9,val.size());
                val.ins(5);
                assertEquals(10,val.size());
                assertEquals(false,val.ins(11));
      }
      public void testDel() {
                assertEquals(5,val.del(0));
                assertEquals(6,val.del(4));
                assertEquals(-1,val.del(4));
                assertEquals(1,val.del(1));
                assertEquals(12,val.del(0));
                assertEquals(30,val.del(0));
                assertEquals(152,val.del(0));
                assertEquals(-1,val.del(0));
      }
      public void testMean() {
                assertTrue(Math.round(34.3) ==
Math.round(val.mean()));
                assertTrue(Math.round(0.0) ==
Math.round((new Valores()).mean()));
      }
      public void testGreater() {
                assertEquals(152,val.greater());
                assertEquals(-1,(new Valores()).greater());
      }

      public void testLower() {
                assertEquals(1,val.lower());
                assertEquals(-1,(new Valores()).lower());
      }
}
```