

Testes de Unidade Automatizados: JUnit

Prof. André Takeshi Endo

Introdução

- O que é teste de unidade?
- Em orientação a objetos, o que seria nossa unidade?
- Hoje o teste de unidade vem associado com a ideia de estar automatizado
- Habilidade exigida do desenvolvedor
 - Implemente tal funcionalidade e os testes de unidade associados

xUnit

- Projetar casos de teste
- Escrever apenas uma vez, mas reexecutá-los várias vezes
- Basicamente, ***configurar o contexto (pré-condições)***, fornecer ***entradas***, obter ***saídas reais***, comparar com ***saídas esperadas***
- Frameworks que apoiam testes automatizados são chamados coletivamente de xUnit
- JUnit para Java (Kent Beck)

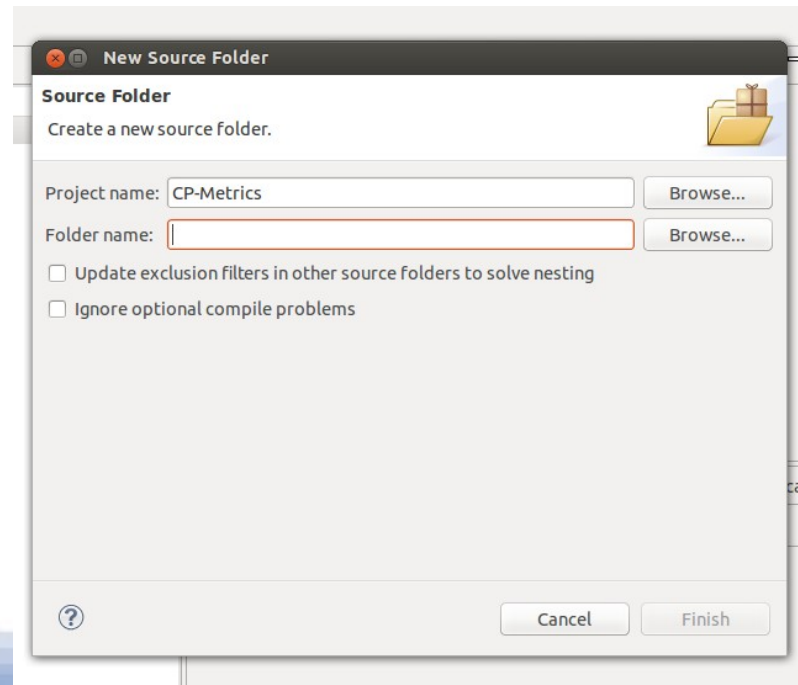


Separando testes do código

- Projetos em Java
- Existe um *Source folder* para o código fonte
 - /src
- Devemos criar um outro *source folder* para os testes
 - /test

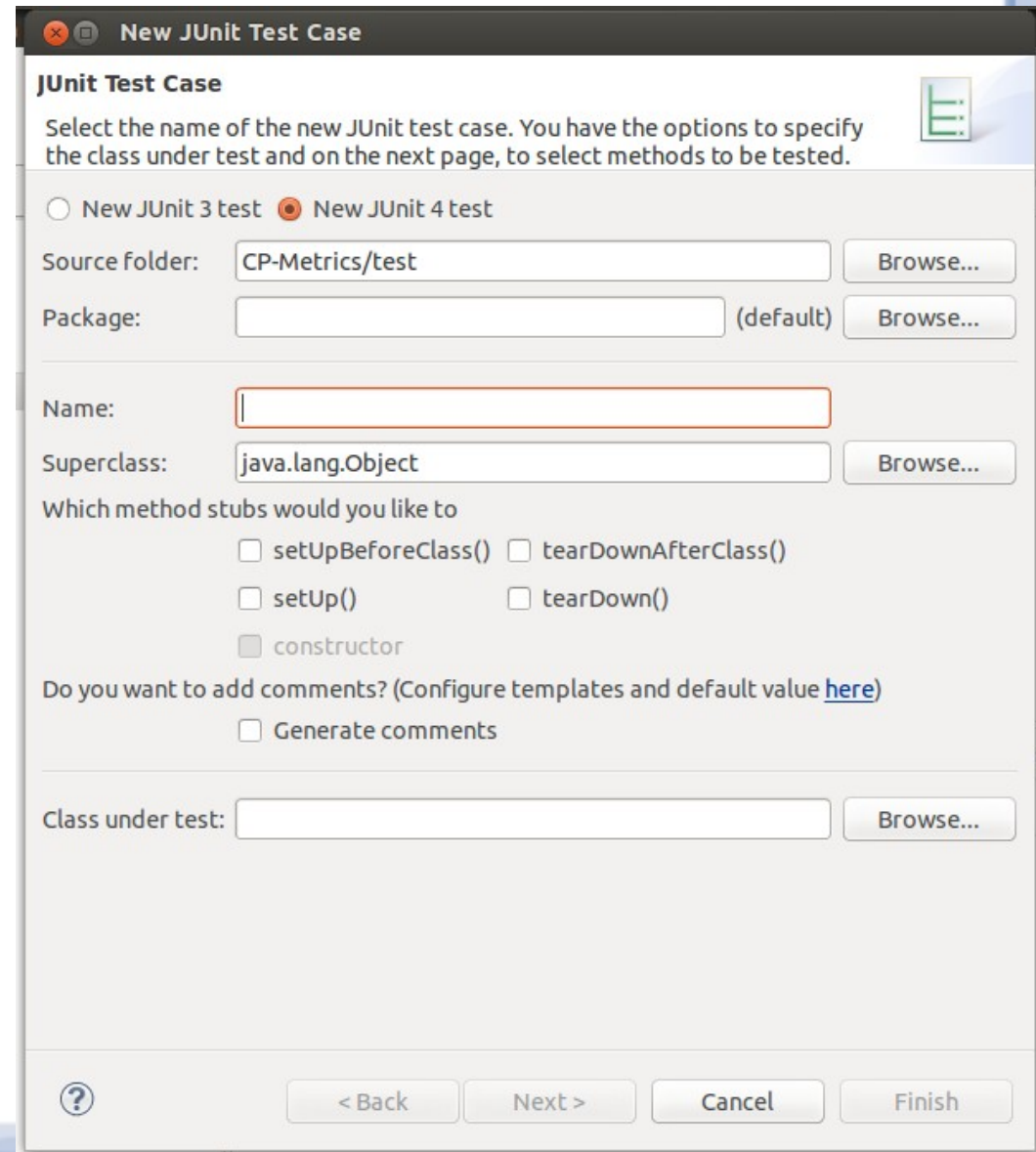
Adicionando o JUnit no Eclipse

- Criando o projeto
 - File → new → Java Project
- Criando o source folder para os teste
 - Sob o projeto, botão direito, new → source folder



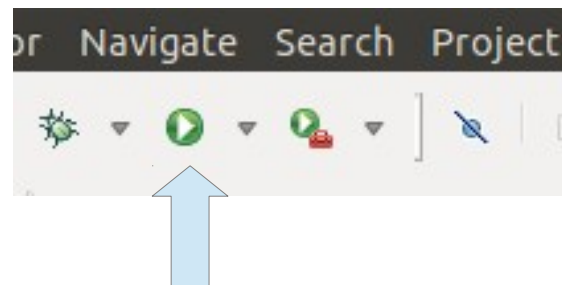
Adicionando o JUnit no Eclipse

- Sob o pacote
- default do /test,
- Botão direito,
- New → Junit Test Case



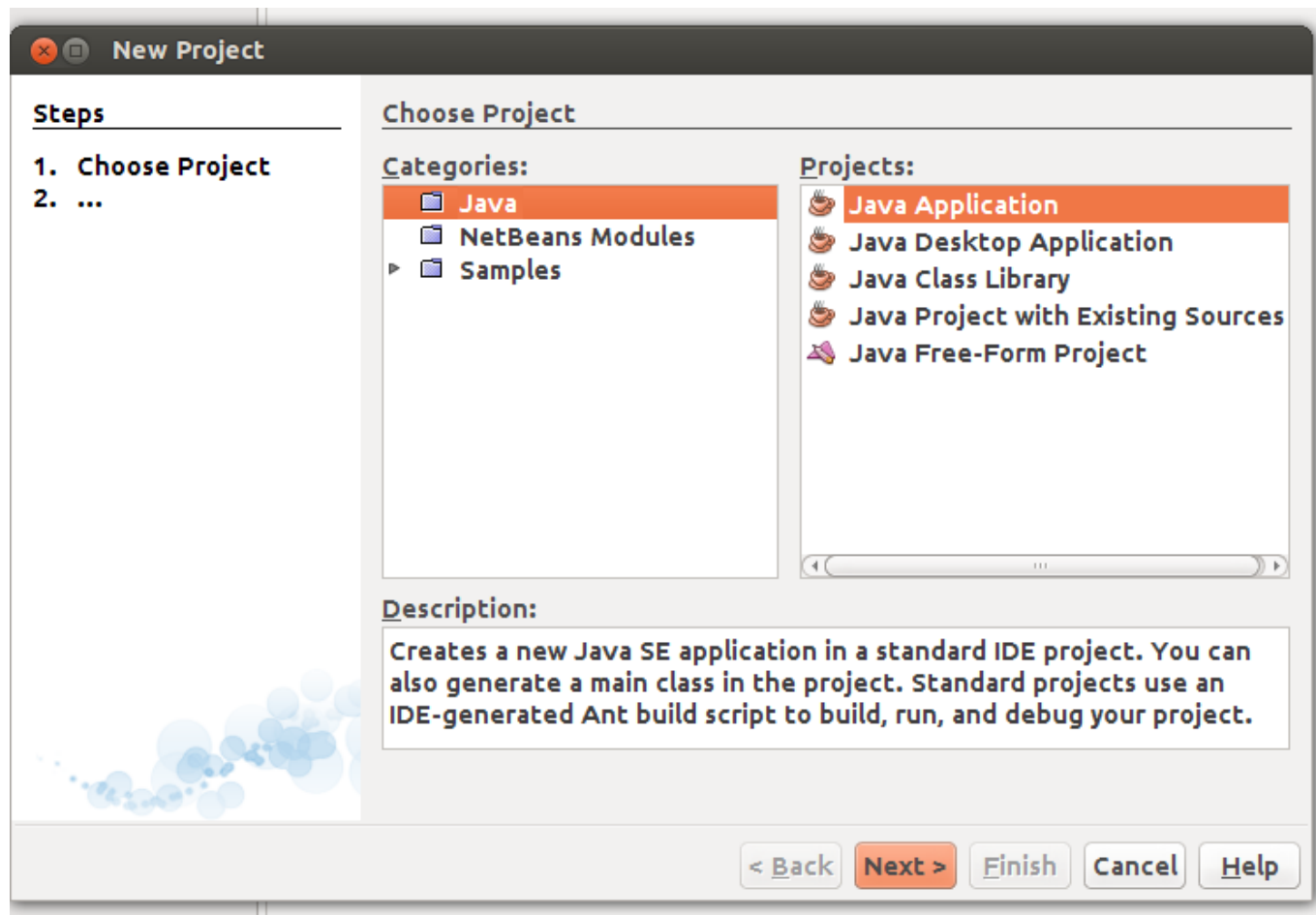
Adicionando o JUnit no Eclipse

- Botão direito na classe de teste,
 - Run as → JUnit Test
- Mais detalhes sobre Eclipse + JUnit em:
 - <http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2FgettingStarted%2Fqs-junit.htm>



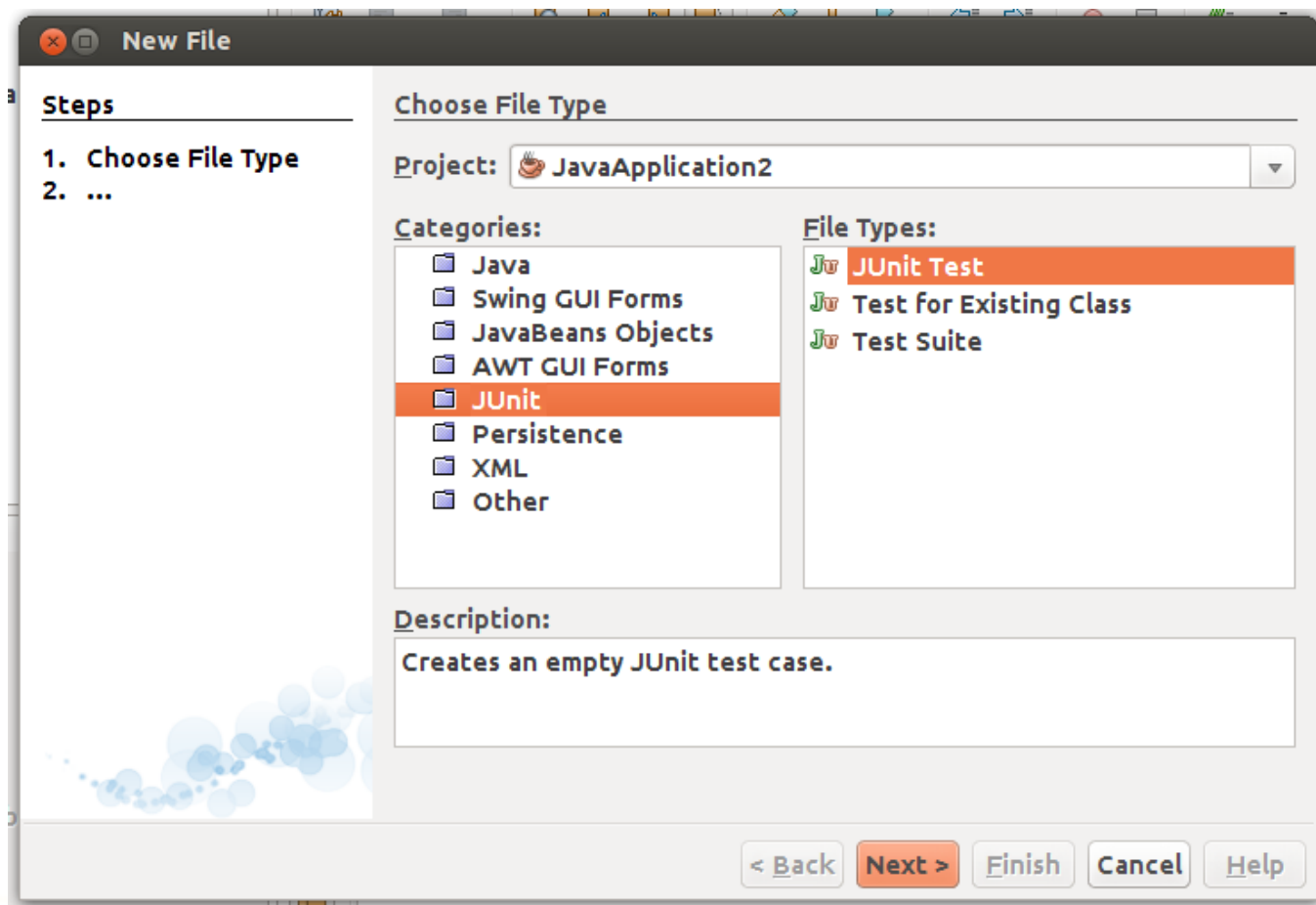
Adicionando o JUnit no NetBeans

- Crie um projeto Java normalmente



Adicionando o JUnit no NetBeans

- Raiz do projeto, botão direito, new → other → JUnit



Adicionando o JUnit no NetBeans

- Após esse passo, selecione JUnit 4.x

New JUnit Test

Steps

1. Choose File Type
2. Name and Location

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

Generated Code

☒ Test Initializer

☒ Test Finalizer

Generated Comments

☒ Source Code Hints

< Back Next > **Finish** Cancel Help

Adicionando o JUnit no NetBeans

- As operações anteriores já criam automaticamente o *source folder /test*
- Para executar
 - Botão direito sob o arquivo da “Test Case Class” → Run File
- Mais detalhes sobre como usar o NetBeans e o Junit juntos em:
 - <https://netbeans.org/kb/docs/java/junit-intro.html>

O que precisamos saber sobre Java?

- Anotações (Annotation)
 - @Something
 - Metadado sintático
 - Classes, métodos, variáveis, parâmetros, etc
 - Extraído em tempo de execução
- Importação estática (import static)
 - Importa membros estáticos das classes, permitindo que esses sejam usados sem qualificação

Test Case Class

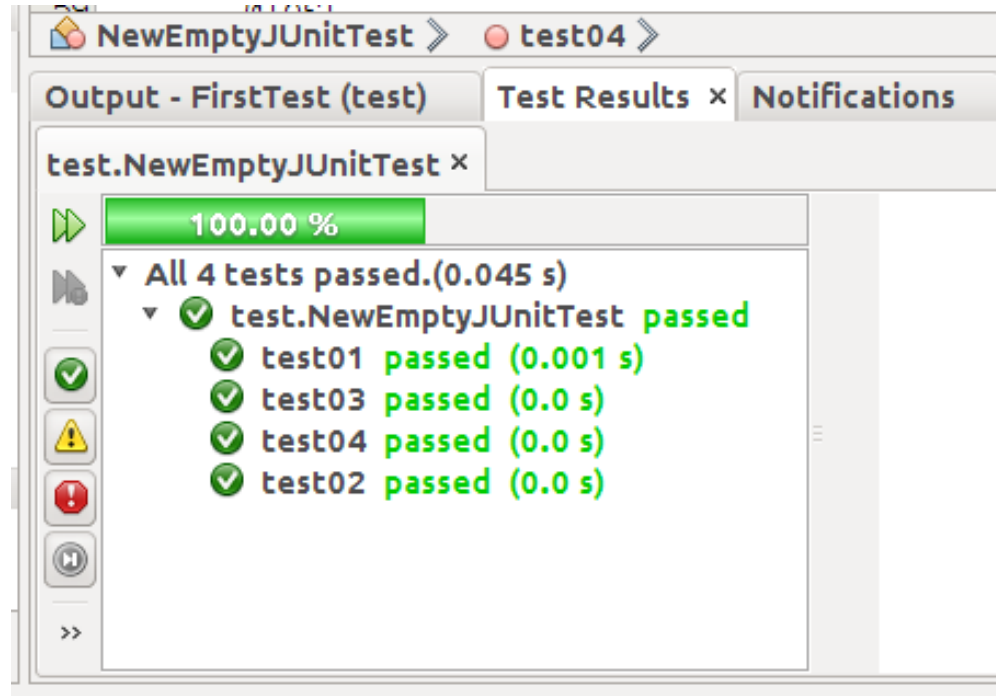
- Uma classe normal no qual iremos fazer algumas distinções
- Adicionar o seguinte import
 - `import static org.junit.Assert.*;`
- **Cada método será um caso de teste**
 - public, retornar void e sem parâmetros
 - `public void teste01() { }`
 - Adicionar a anotação `@Test` para cada método que representa um caso de teste

Test Case Class

- Exercício: Cria uma classe TestandoJUnit em /test
 - Adicione o import static
 - Crie quatro métodos denominados teste01, teste02, teste03 e teste04.
 - Anote-os com @Test
 - Execute a classe

Test Case Class

- Exercício: Cria uma classe TestandoJUnit em /test
 - Saída obtida no NetBeans
 - Veja as funções disponíveis



Test Case Class

- Dentro de cada método anotado (@Test) é onde codificamos (automatizamos) os casos de teste.
- Considere a class ArrayList, teste:
 - a inclusão
 - unidade → método: add()
 - a remoção
 - unidade → método: remove()
 - a obtenção do índice
 - unidade → método: indexOf()
 - se um objeto está incluído
 - unidade → método: contains()

Use `ArrayList<String> list = new ArrayList<String>();`

Test Case Class

- Assertivas
- assertEquals(<saida esperada>, <saida real>);
 - Usado como mecanismo para verificar/comparar a saída esperada com a saída real.

- Complete os casos de teste com as assertivas!!

Inclusão e Exclusão → Usar o list.size()

IndexOf() → o retorno

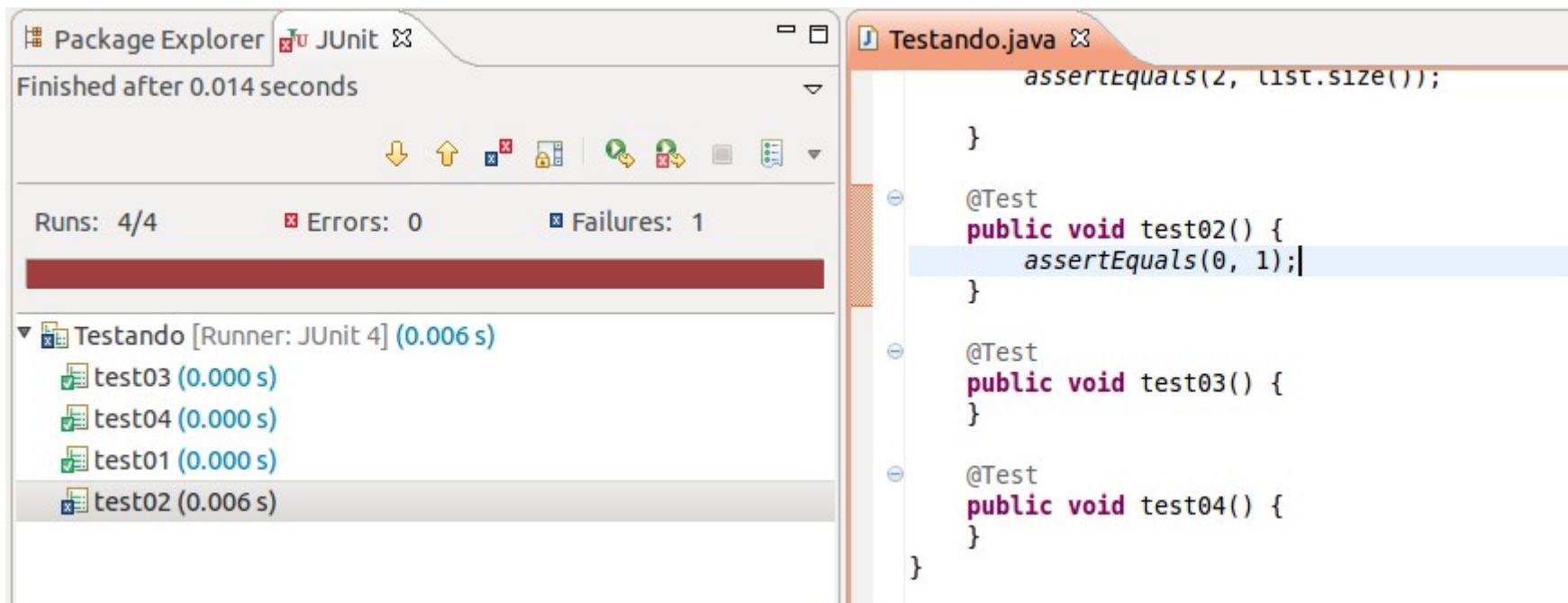
contains → retorno

Test Case Class

- Outras assertivas
 - `assertTrue(<condicao>);`
 - `assertFalse(<condicao>);`
 - `assertNull(<object>);`
 - `assertNotNull(<object>);`
 - `assertSame(<obj esperado>, <obj real>);`
 - `assertNotSame(<obj nao esperado>, <obj real>);`
- Crie uma nova classe de teste
 - Crie um caso de teste para cada assertiva
 - Use as classes `ArrayList` e `String`

Test Case Class

- Cause algumas falhas



The screenshot displays an IDE interface with two main panels. The left panel, titled 'JUnit', shows the results of a test run. It indicates 'Finished after 0.014 seconds' and provides a summary: 'Runs: 4/4', 'Errors: 0', and 'Failures: 1'. Below this, a list of test cases is shown: 'test03 (0.000 s)', 'test04 (0.000 s)', 'test01 (0.000 s)', and 'test02 (0.006 s)'. The 'test02' entry is highlighted with a red background, indicating it failed. The right panel shows the source code for 'Testando.java'. The code includes a package declaration, imports for 'List' and 'Assert', and four test methods: 'test01', 'test02', 'test03', and 'test04'. The 'test02' method is highlighted in blue and contains the line 'assertEquals(0, 1);', which is the cause of the failure.

```
package Testando;

import java.util.List;
import static org.junit.Assert.*;

public class Testando {

    @Test
    public void test01() {
        assertEquals(2, list.size());
    }

    @Test
    public void test02() {
        assertEquals(0, 1);
    }

    @Test
    public void test03() {
    }

    @Test
    public void test04() {
    }
}
```

Test Case Class

- É possível forçar via código que um determinado falhe
- Use:
 - fail(“mensagem.”);

- Teste o lançamento da exceção `IndexOutOfBoundsException` quando o método `list.get(int)` é chamado.
- Use o `fail()`;

Test Fixture

- Todo caso de teste é formado por entrada e saída esperada
- Assertivas são usadas para comparar as **saídas esperadas** com as **saídas reais**
- As entradas é muito mais complexa em OO
- Precisa de um Test Fixture
 - Estado fixado do software sob teste usando como base para executar o software

Test Fixture

- Métodos especiais para preparar o teste
- Método público, void e sem parâmetro
- @Before → execute **antes** de cada caso de teste
- @After → execute **depois** de cada caso de teste
- Antes de executar todos os testes (única vez)
 - @BeforeClass
- Depois de executar todos os testes (única vez)
 - @AfterClass
- @BeforeClass e @AfterClass → método estático

Exercício: verificando a ordem das anotações

- Crie uma classe de teste com 3 casos de teste
 - Imprima “caso de teste X”
- Adicione métodos para as anotações @Before, @After, @BeforeClass, @AfterClass
 - Imprima “@Anotacao”
- Execute e verifique as saídas no console
- Determine a ordem de execução

Test Case Class

- Algum teste não está totalmente codificado
- Não quer considerar esse teste
- Anotar com **@Ignore** os métodos que você não quer executar
 - @Ignore
@Test
public void teste01() { ... }

Test Case Class

- Melhorar a forma de verificar o lançamento de exceções
 - @Test(***expected***=MyException.class)
- Estabelecer o tempo limite para a execução de caso de teste
- @Test(***timeout***=100)
 - Milisegundos

- Crie casos de teste que aplicam essas funções

Test Suite Class

- Executar 2 ou mais classes de teste
- Crie uma classe que é um conjunto de teste
- `@RunWith(Suite.class)`
`@SuiteClasses({`
 `ClasseTeste1.class,`
 `ClasseTeste2.class,`
 `... })`
`public class AllTests { }`

- Crie um conjunto de teste com todas as classes de teste
- Execute e veja o resultado

Bibliografia

- <http://junit.org/>
- <http://www.tutorialspoint.com/junit/>
- <http://www.vogella.com/articles/JUnit/article.html>
- <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>