Lenguaje Java Avanzado

Sesión 4: Pruebas con JUnit

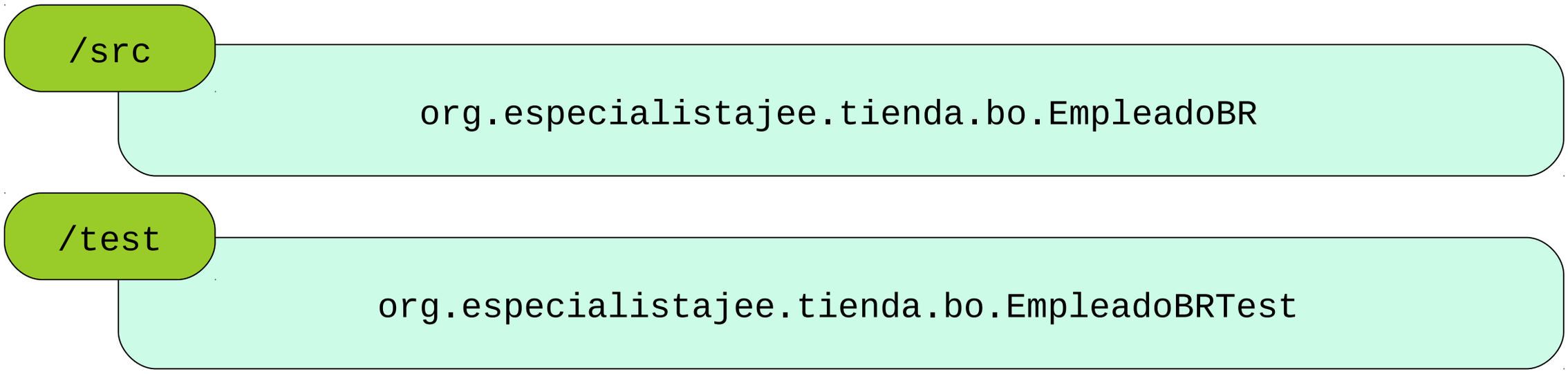
Índice

Introducción a JUnit

- JUnit es una librería que permite automatizar las pruebas de los diferentes módulos de una aplicación Java
  - *Caso de prueba*: clase o módulo con métodos para probar los métodos de una clase o módulo concreto
  - *Suite de prueba*: organización de casos de prueba, en forma de una jerarquía determinada

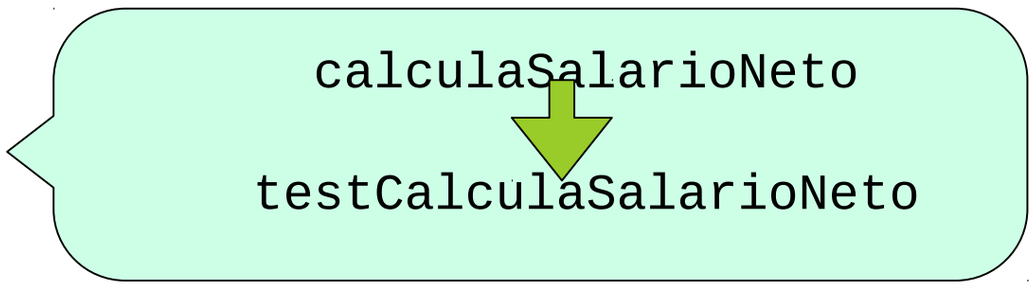
Implementar los casos de prueba

- Una clase de prueba por cada clase a probar
- Mismo nombre, pero con sufijo `Test`
- Mismo paquete, directorios separados



Método de prueba

- Anotar con `@Test` los métodos de prueba
- Nombre con prefijo `test-`



- Ejecutar método a probar
- Comprobar resultado con `assert-`

Un caso sencillo

- Probamos `EmpleadoBR.testCalculaSalario`
- Se recomienda un método por caso de prueba

```
public class EmpleadoBRTest {

  @Test
  public void testCalculaSalarioNeto1() {
    float resultadoReal =
                EmpleadoBR.calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado,
                  resultadoReal, 0.01);
  }
}
```

Ejecución de pruebas

- Desde línea de comando

```
java -cp ./junit.jar junit.swingui.TestRunner
```

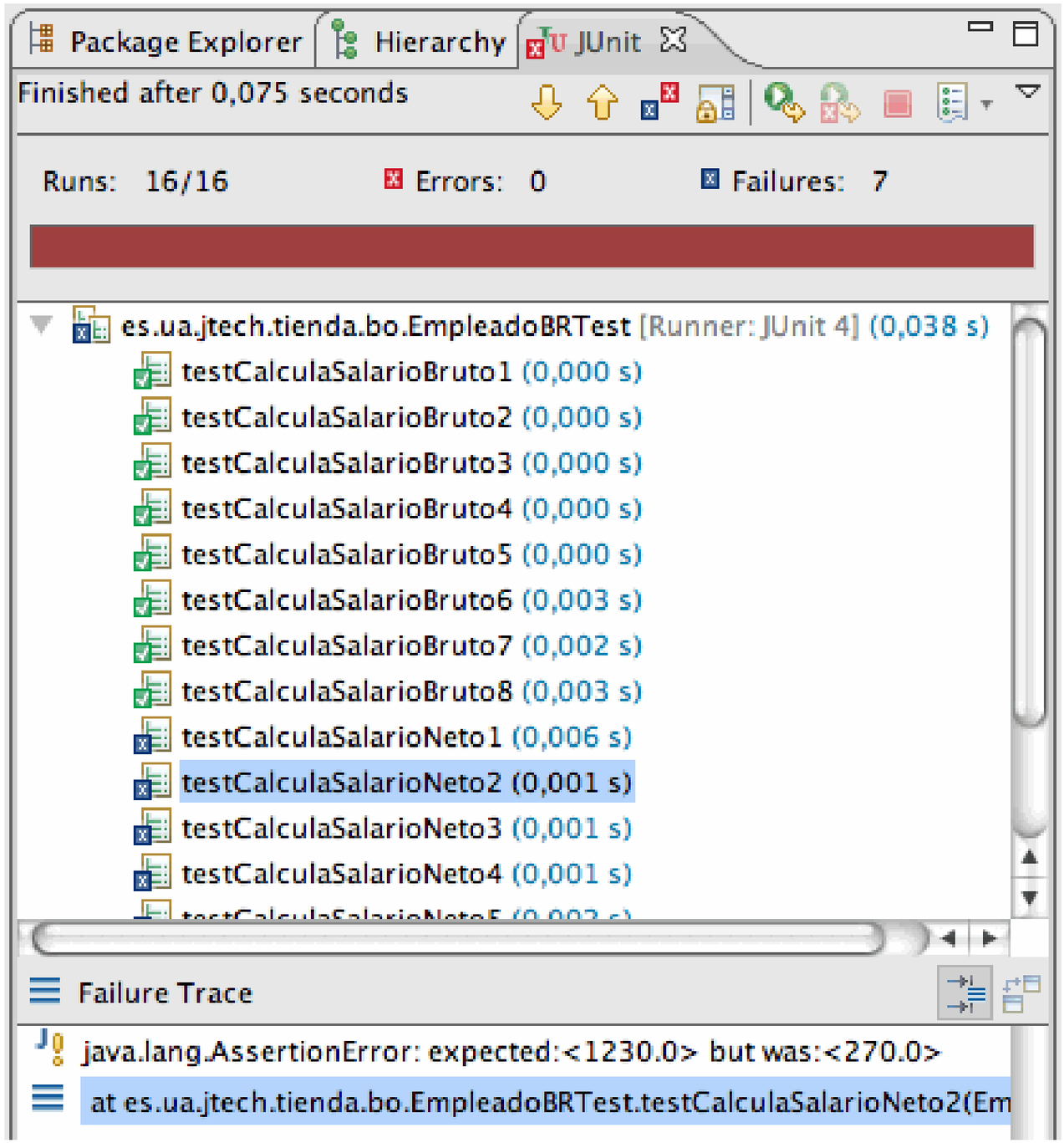
- Desde código Java

```
String[] nombresTest = {EmpleadoBRTest.class.getName()};
junit.swingui.TestRunner.main(nombresTest);
```

- Desde Eclipse

Run As > JUnit test

Resultados de las pruebas



Prueba con excepciones

- Usar la anotación `@Test`

```
@Test(expected=BRException.class)
public void testCalculaSalarioNeto9() {
  EmpleadoBR.calculaSalarioNeto(-1.0f);
}
```

- Usar la instrucción `fail()`

```
@Test
public void testCalculaSalarioNeto9() {
  try {
    EmpleadoBR.calculaSalarioNeto(-1.0f);
    fail("Se esperaba excepcion BRException");
  } catch(BRException e) {}
}
```

Fixtures

- Elementos fijos
  - Se reutilizan en diferentes pruebas

`@Before`	Antes de cada test
`@After`	Después de cada test
`@BeforeClass`	Antes de todos los tests
`@AfterClass`	Después de todos los tests

Suite de pruebas

- Agrupa pruebas

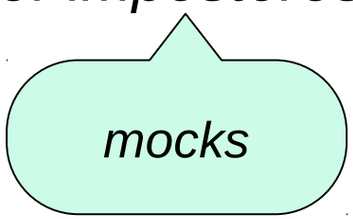
```
import junit.framework.Test;
import junit.framework.TestCase
import junit.framework.TestSuite

public class MyTestSuiteRunner extends TestCase {

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

Ventajas

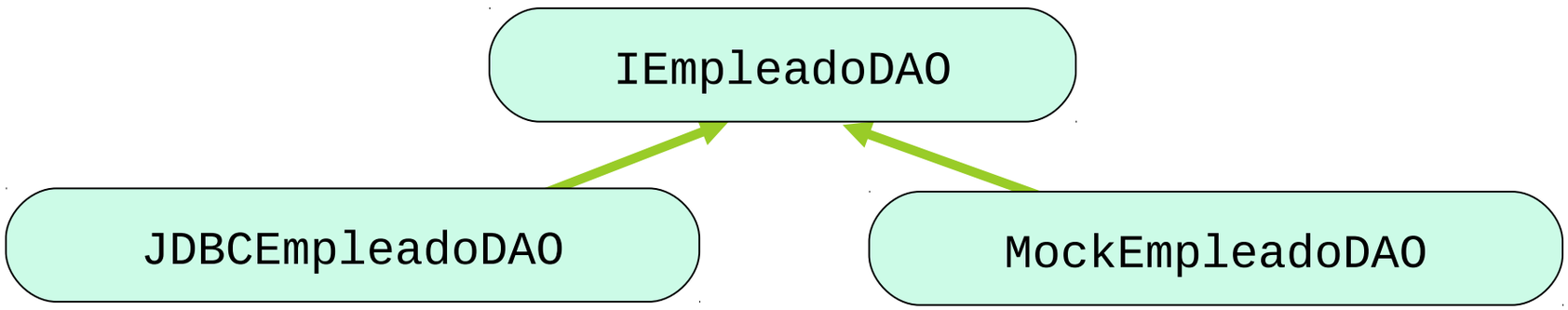
- *Framework* estándar para pruebas
- Batería de pruebas reutilizables
- Permite realizar pruebas de regresión
- Interfaz para presentación de resultados

Objetos *mock*

- Debemos poder predecir el resultado de los métodos a probar
- Algunos componentes dificultan las pruebas
  - *DAOs*
  - *Proxys*
  - etc ...
- Toman como entrada datos que no controlamos
- Solución:
  - Sustituir estos componentes por *impostores*



Implementación de los *mock*




```
@Override
IEmpleadoDAO getEmpleadoDAO() {
    return new MockEmpleadoDAO();
}
```

Pruebas de base de datos

- Si nuestro objetivo es probar el DAO no tendría sentido sustituirlo por un *mock*

- Restablecer el estado de la BD antes de cada prueba (*fixtures*)

- Utilizar DBUnit

http://www.dbunit.org/

Test-Driven Development

- Para cada funcionalidad a implementar
  - Escribir las pruebas y comprobar que fallan
  - Escribir el mínimo código para que funcionen
  - Refactorizar el código escrito
- También conocida como *red*-*green*-*refactor*

Ventajas de TDD

- Código probado desde el principio
- Todo el código bajo el control de las pruebas
- No es necesario depurar código complejo
- Código de gran calidad
- Alta confianza en el código desarrollado

¿Preguntas...?