

Curso de formación

Pruebas de Software

Pruebas Unitarias

- Introducción
- Test y Suites
- Parámetros
- Aserciones
- Dobles
- Paralelismo
- Interceptores y Listeners
- Resultados
- Conclusiones

- **Introducción**
- Test y Suites
- Parámetros
- Aserciones
- Dobles
- Paralelismo
- Interceptores y Listeners
- Resultados
- Conclusiones

- Cada **lenguaje de programación** dispone de uno o varios **frameworks** para implementar pruebas automáticas unitarias y de integración
- Los **frameworks** aprovechan toda la potencia del lenguaje para facilitar la tarea del desarrollador de tests
- Aunque existen algunas diferencias dependiendo del lenguaje, **todos los frameworks son muy parecidos:**
 - Cada **test** se implementa en una **función / método**
 - Existen **funciones / métodos** para las **aserciones**

- Test en Java con el framework JUnit

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

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {

        // Given
        MyClass tester = new MyClass();

        // When
        double result = tester.multiply(10, 0)

        // Then
        assertEquals("10 x 0 must be 0", 0, result);
    }
}
```

- Test en Groovy con Spock

```
class InterpolateServiceTest extends Specification {  
  
    @Shared def interpolateService = new InterpolateService()  
  
    def "interpolate two numbers with even no. of steps"() {  
        expect:  
            interpolateService.interpolate(a, b, c) == d  
        where:  
            a | b | c | d  
            5.0 | 25.0 | 4 | [5.0, 10.0, 15.0, 20.0, 25.0]  
            2.0 | 14.0 | 6 | [2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0]  
    }  
}
```

- Test en JavaScript con Jasmine

```
describe( "distance converter", function () {  
    it("converts inches to centimeters", function () {  
        expect(Convert(12, "in").to("cm")).toEqual(30.48);  
    });  
    it("converts centimeters to yards", function () {  
        expect(Convert(2000, "cm").to("yards")).toEqual(21.87);  
    });  
});
```

- **Test en C++ con GoogleTest**

```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(40320, Factorial(8));
}
```


- Vamos a estudiar la librería **TestNG** para testing Java

TestNG

<https://testng.org/>

Testing Java con TestNG

- TestNG es un framework para desarrollo de pruebas automáticas basado en Junit (Java) y NUnit (.NET)
- Cuenta con integración para los IDEs Java y en las **herramientas de construcción de proyectos**

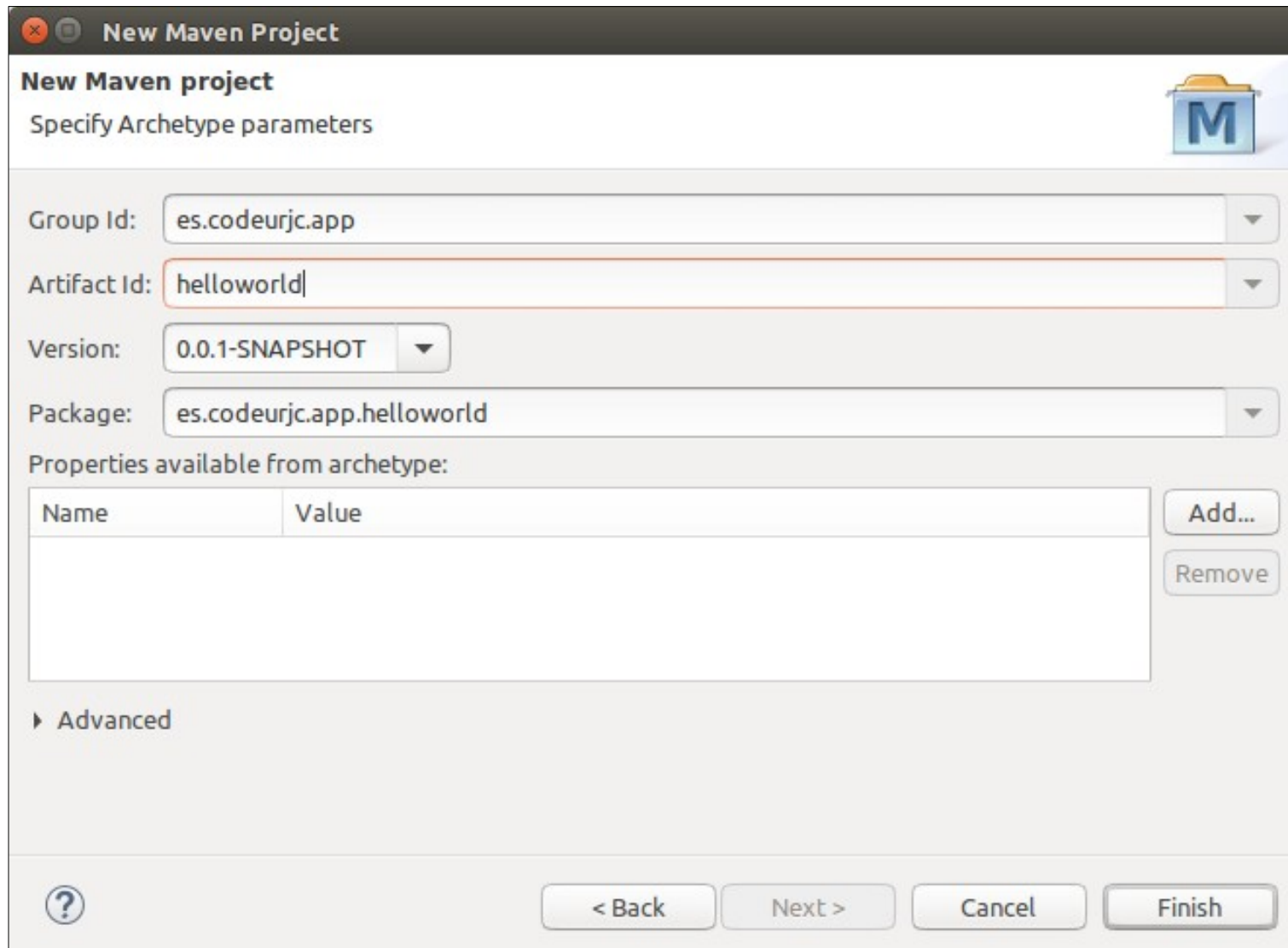


- **Maven**

- Tecnología más usada para definir proyectos Java
- Sistema de **gestión de dependencias** (*librerías*) y sus versiones
- Sistema de **construcción de proyectos** (de código a entregable .zip)
- Estructura única de proyecto compatible con todos los **entornos de desarrollo** y sistemas de **integración continua**

- **Cómo crear un proyecto Maven en Eclipse**
 - File > New > Other > Maven Project
 - Dejar la plantilla que aparece por defecto seleccionada (**maven-archetype-quickstart**)
 - Indicar el nombre del proyecto en dos partes:
 - **GroupId: es.codeurjc.app**
 - **ArtifactId: helloworld**

Maven



New Maven Project

New Maven project
Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

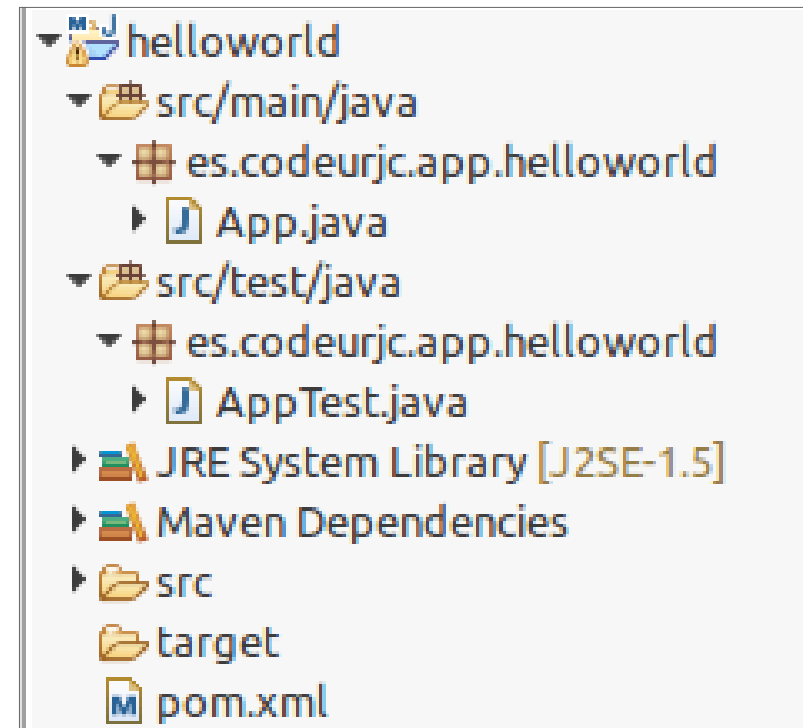
Properties available from archetype:

Name	Value

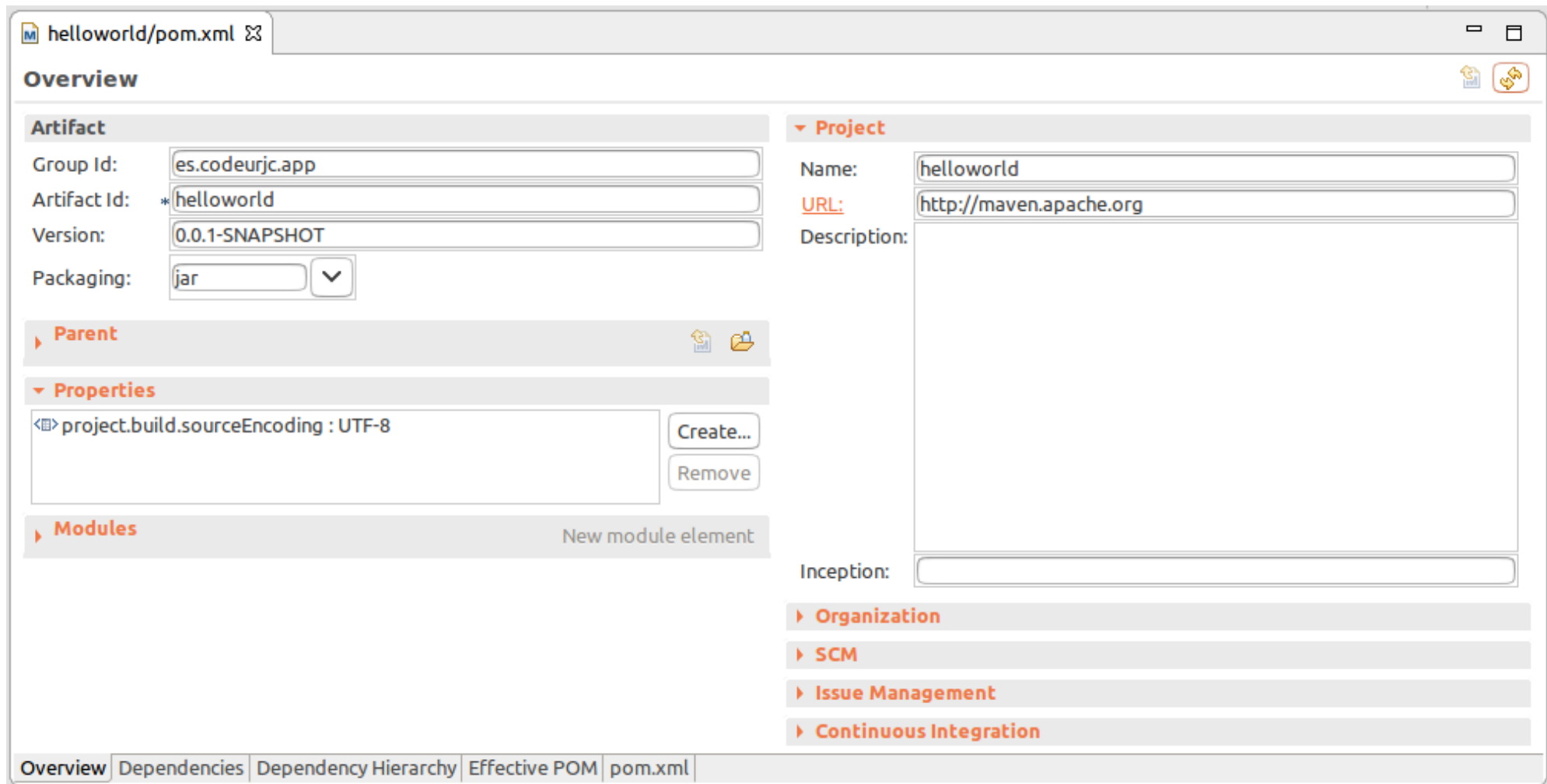
Advanced

Maven

- Los proyectos Maven tienen la siguiente **estructura**
 - **src/main/java**: Código de la aplicación
 - **src/test/java**: Código de los tests
 - **pom.xml**: Fichero de descripción del proyecto (nombre, dependencias, configuraciones, etc...)



- **pom.xml**: Configuración del proyecto

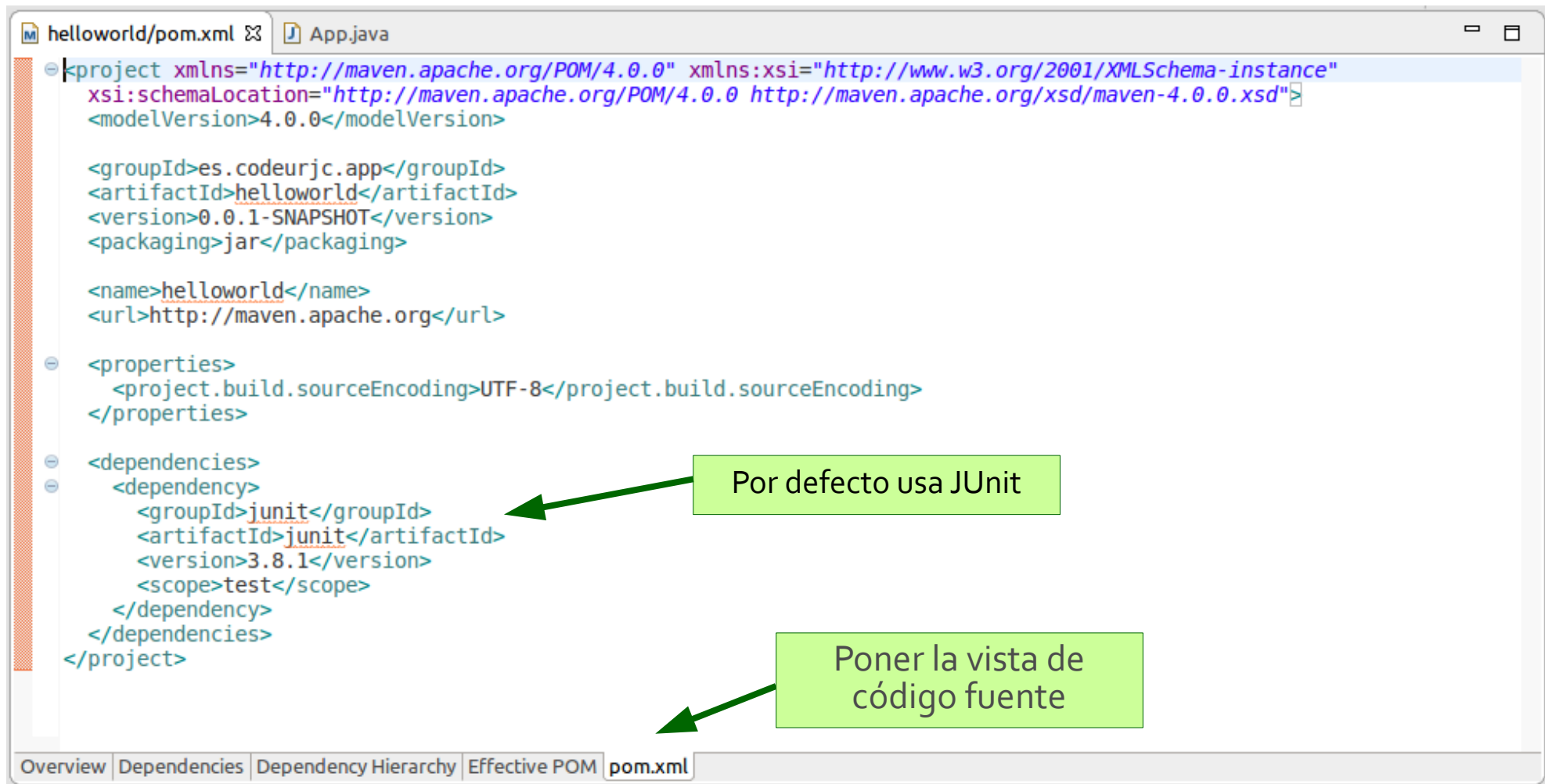


The screenshot displays the Maven IDE interface for a project named 'helloworld/pom.xml'. The interface is divided into several sections:

- Overview**: The main section for configuring the project. It includes fields for:
 - Artifact**: Group Id (es.codeurjc.app), Artifact Id (*helloworld), Version (0.0.1-SNAPSHOT), and Packaging (jar).
 - Project**: Name (helloworld), URL (http://maven.apache.org), and Description (empty).
 - Parent**: A section for selecting a parent project.
 - Properties**: A section for defining project properties. It shows a property 'project.build.sourceEncoding : UTF-8' with 'Create...' and 'Remove' buttons.
 - Modules**: A section for adding new module elements.
- Organization**: A section for defining the organization.
- SCM**: A section for defining the Source Control Management system.
- Issue Management**: A section for defining the issue management system.
- Continuous Integration**: A section for defining the continuous integration system.

The bottom of the interface features a tabbed navigation system with the following tabs: Overview, Dependencies, Dependency Hierarchy, Effective POM, and pom.xml.

- **pom.xml**: Configuración del proyecto



The screenshot shows an IDE window with two tabs: 'helloworld/pom.xml' and 'App.java'. The 'pom.xml' tab is active, displaying the following XML content:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>es.codeurjc.app</groupId>
  <artifactId>helloworld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloworld</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Two green callout boxes with arrows point to specific parts of the XML:

- An arrow points from the box "Por defecto usa JUnit" to the `<groupId>junit</groupId>` tag.
- An arrow points from the box "Poner la vista de código fuente" to the `<scope>test</scope>` tag.

The IDE's bottom status bar shows tabs for 'Overview', 'Dependencies', 'Dependency Hierarchy', 'Effective POM', and 'pom.xml'.

- **pom.xml:** Configuración del proyecto
 - **groupId:** Organización, familia
 - **artifactId:** Nombre del proyecto
 - **version:** Versión del proyecto (especialmente útil para librerías)
 - **packaging:** Tipo de aplicación (jar es una aplicación normal)
 - **name:** Nombre “bonito” del proyecto (para documentación)
 - **url:** Web del proyecto (para documentación)

```
<groupId>es.codeurjc.app</groupId>
<artifactId>helloworld</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>helloworld</name>
<url>http://maven.apache.org</url>
```

- **pom.xml:** Configuración del proyecto
 - **properties:**
 - Configuraciones generales del proyecto
 - Codificación de los ficheros fuente

```
<properties>  
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
</properties>
```

- **pom.xml: Configuración del proyecto**
 - **dependencies:**
 - ▮ Dependencias (**librerías**)
 - ▮ Cada librería está identificada por su **groupId**, **artifactId** y **versión** (coordenadas)
 - ▮ Se pueden poner tantas dependencias como se quiera

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- Actualizamos el pom.xml para usar una versión más recientes de **Java (1.8)** y **cambiamos a TestNG (6.10)**
- **properties**

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

- **dependencies**

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>6.10</version>
  <scope>test</scope>
</dependency>
```

- **pom.xml:** Configuración del proyecto
 - **Cuidado!** Algunos cambios en el fichero pom.xml no se reflejan en eclipse de forma automática
 - Cuando se hace un cambio y eclipse no se actualiza con esos cambios, se tiene que indicar explícitamente
 - ▮ Botón derecho proyecto > Maven > Update Project...

- Cambiamos el código del test (*src/test/java*)

```
package es.codeurjc.app.helloworld;

import org.testng.annotations.*;
import org.testng.Assert;

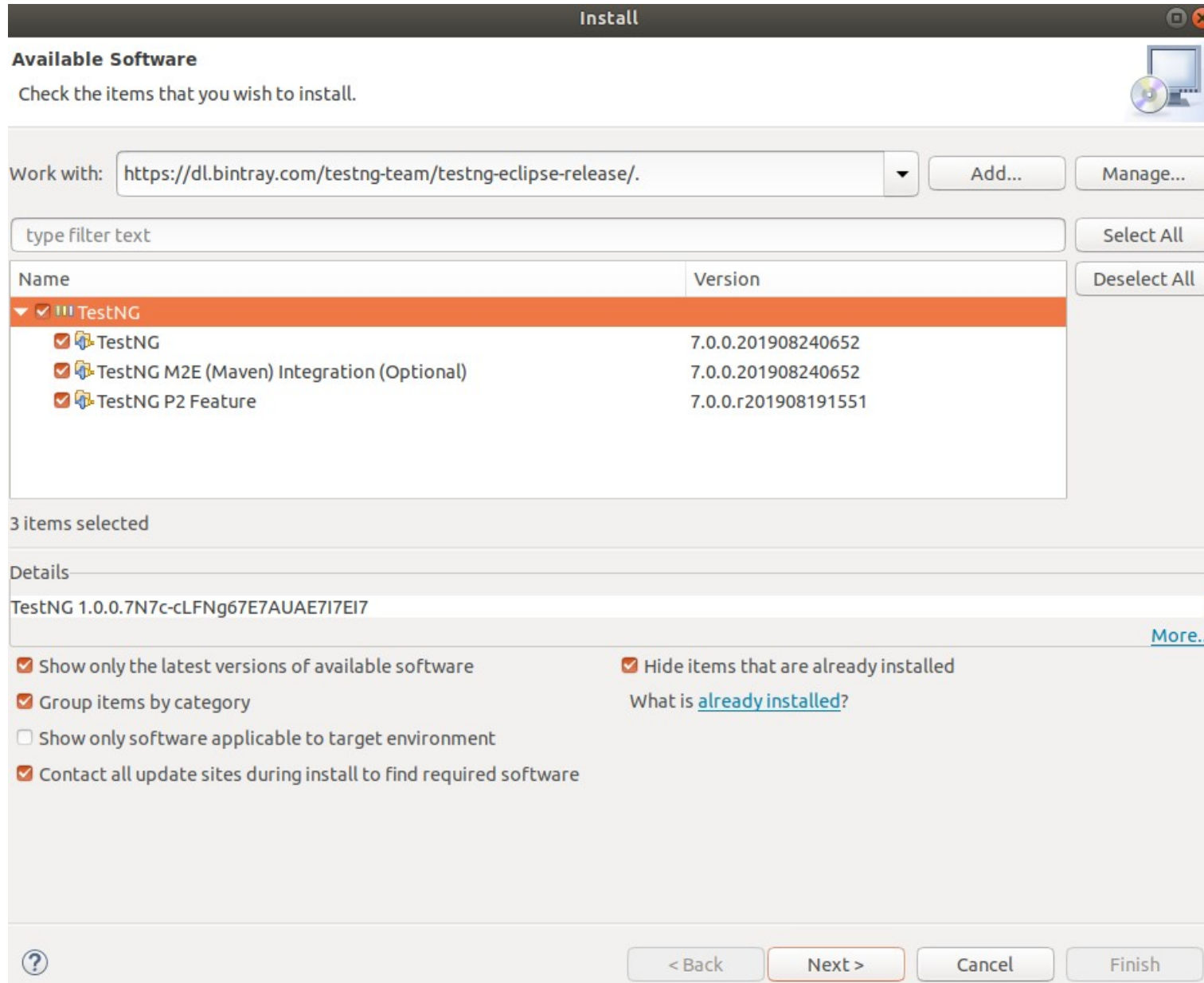
/**
 * Unit test for simple App.
 */
public class AppTest {

    @Test
    public void pruebaSuma() {
        int result = 3 + 4;
        Assert.assertEquals(result, 7);
    }

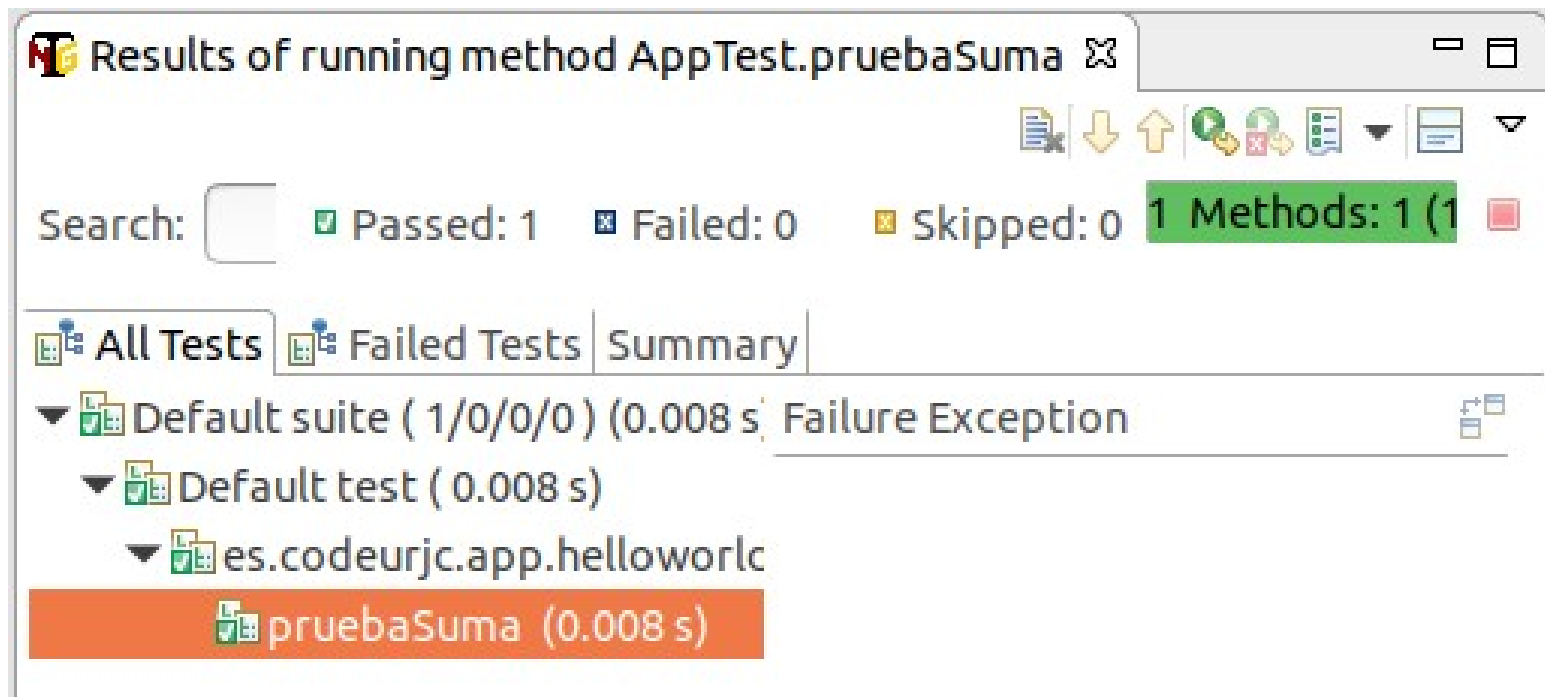
}
```

- Instalamos el **plugin de TestNG para Eclipse**
 - Seleccionamos *Help > Install New Software*
 - En el input *Work with* introducimos:
 - <https://dl.bintray.com/testng-team/testng-eclipse-release/>
 - Seguimos la instalación guiada de Eclipse.
- Podemos realizar estos pasos en otros IDEs siguiendo la documentación oficial
 - <https://testng.org/doc/>

Maven



- Ejecución de test en Eclipse con TestNG
 - Click derecho sobre el proyecto
 - Run As... > TestNG



- **Importar proyectos Maven en eclipse**
 - Si está dentro de un .zip, descomprimir
 - File > Import > Existing Maven project
 - Seleccionar la carpeta raíz de todos los proyectos

- Introducción
- **Test y Suites**
- Parámetros
- Aserciones
- Dobles
- Paralelismo
- Interceptores y Listeners
- Resultados
- Conclusiones

- Los tests son métodos de una clase Java cuyo nombre termina con **Test (Test Class)**
- Cada test es un método anotado con la anotación **@Test** del paquete `org.testng.annotations` (**Test Method**)

```
import org.testng.annotations.Test;

public class AppTest {

    @Test
    public void firstTest() {
        // Test implementation
    }

}
```

- Existen **3 partes** diferenciadas en la implementación de un test:
 - **Arrange / given:** Definir el estado inicial del SUT. Las condiciones del test
 - **Act / when:** Actuar sobre esa SUT. Ejercitarlo.
 - **Assert / then:** Verificar que el comportamiento obtenido es el esperado

Test y Suites

```
ejem1/src/test/java/es/codeurjc/test/ejem/Calculadora1Test.java
```

```
import static org.testng.Assert.assertEquals;
import org.testng.annotations.Test;

public class Calculadora1Test {

    @Test
    public void testSuma() {

        // Arrange / Given
        Calculadora calculadora = new Calculadora();

        // Act / When
        int res = calculadora.suma(1, 1);

        // Assert / Then
        assertEquals(2, res, 0);
    }
}
```

- La verificación se realiza mediante **aserciones** (assertions) que determinan si una condición se cumple
- El **resultado** de una aserción puede ser:
 - **Éxito (Success):** Test pasa. Se cumplen las expectativas
 - **Fallo (Failure):** Test no pasa. No se cumplen
 - **Error:** Se ha producido una excepción no controlada
- Un test puede tener **varias aserciones** (incluso podría no contener ninguna, pero no es una buena práctica)

- Las Test Classes pueden tener **varios Test Methods**

ejem1/src/test/java/es/codeurjc/test/ejem/Calculadora2Test.java

```
import static org.testng.Assert.assertEquals;
import org.testng.annotations.Test;

public class Calculadora2Test {

    @Test
    public void testSuma() {
        ...
    }

    @Test
    public void testResta() {
        ...
    }
}
```


- Las aserciones se implementan como métodos estáticos de la clase **org.testng.Assert**
- Algunos ejemplos de aserciones:
 - `assertTrue(boolean condition)`
 - `assertFalse(boolean condition)`
 - `assertEquals(Object actual, Object expected)`
 - `assertArrayEquals(...)`

- La clase **org.junit.Assert** también dispone de métodos para hacer fallar el test si el código del test determina que no se cumplen las expectativas
 - `fail()`
 - `fail(String message)`
- En cuanto una aserción falla, el test no continúa su ejecución. Aunque se ejecutarán el resto de tests.

Test y Suites

fail(String)	Used to flag a certain section of the test should not be reached.
assertTrue(boolean condition)	Assert that the condition should evaluate to true.
assertEquals(expected, actual)	Assert that the value of the actual result is equal to the expected result.
assertEquals(expected, actual, delta)	Asserts that two doubles or floats are equal to within a positive delta.
assertNull(object)	Assert that the object is null
assertNotNull(object)	Assert that the object is not null
assertSame(expected, actual)	Asserts that two objects refer to the same object.
assertNotSame(expected, actual)	Asserts that two objects do not refer to the same object.

- Los métodos de las aserciones están sobrecargados para recibir el **mensaje** que aparecerá si la aserción falla

```
ejem1/src/test/java/es/codeurjc/test/ejem/Calculadora3Test.java
```

```
import static org.testng.Assert.assertEquals;
import org.testng.annotations.Test;

public class Calculadora3Test {

    @Test
    public void testSuma() {

        Calculadora calculadora = new Calculadora();
        int res = calculadora.suma(1, 1);
        assertEquals(2, res, "1+1 should be equal to 2");
    }
}
```

- **Ejercicio 1**

- Implementa varios tests de la clase Complex (proporcionada)
- Comprueba que el complejo Complex(0,0) tiene la parte real y la parte imaginaria con valor 0
- Comprueba que Complex(0,0) es el valor neutro de la operación suma:

$\text{Complex}(0,0) + \text{Complex}(1,1) == \text{Complex}(1,1)$

$\text{Complex}(1,1) + \text{Complex}(0,0) == \text{Complex}(1,1)$

- **Test fixtures (Partes fijas de un test)**
 - Como los tests son métodos de una clase, pueden compartir información usando los atributos de la clase
 - Los métodos anotados con **@BeforeMethod** se ejecutan siempre antes de ejecutar cada test
 - Los métodos anotados con **@AfterMethod** se ejecuta siempre después de ejecutar cada test (aunque haya fallado)
 - Al ejecutar cada **test** se crea **una instancia** nueva de la clase y se ejecuta el método del test

- **Test fixtures**

ejem1/src/test/java/es/codeurjc/test/ejem/Calculadora4Test.java

```
import static org.testng.Assert.assertEquals;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class Calculadora4Test {

    Calculadora calc;

    @BeforeMethod
    public void setUp() {
        this.calc = new Calculadora();
    }

    @Test
    public void testSuma() {
        // Test
    }
}
```

- **Ejercicio 2**

- Transforma el Ejercicio 1 para usar Test fixtures
- Define un atributo zero que se inicializa en un método setUp anotado como **@BeforeMethod**
- Ese atributo se usará siempre que se necesite el número complejo $0+0i$

`zero + Complex(1,1) == Complex(1,1)`

`Complex(1,1) + zero == Complex(1,1)`

- **Test fixtures (Partes fijas de un test)**
 - Por cada test case se crea una instancia de la clase y se ejecuta el método **@BeforeMethod**, luego el test y por último el método **@AfterMethod**
 - A veces es conveniente ejecutar un código antes de todos los tests de una clase y después de todos ellos
 - Para eso existen las anotaciones **@AfterClass** y **@BeforeClass**
 - Estas anotaciones tienen que estar en **método estáticos** de la clase porque serán llamados antes de crear ningún objeto

Test y Suites

ejem1/src/test/java/es/codeurjc/test/ejem/Calculadora5Test.java

```
public class Calculadora5Test {  
  
    Calculadora calc;  
  
    @BeforeClass  
    public static void setUpClass() {  
        System.out.println("Before all tests");  
    }  
  
    @BeforeMethod  
    public void setUp() {  
        System.out.println("Before test");  
        this.calc = new Calculadora();  
    }  
  
    @Test  
    public void testSuma() { ... }  
  
    @Test  
    public void testResta() { ... }  
  
    @AfterMethod  
    public void teardown() {  
        System.out.println("After test");  
    }  
  
    @AfterClass  
    public static void teardownClass() {  
        System.out.println("After all tests");  
    }  
}
```

Consola

Before all tests
Before test
After test
Before test
After test
After all tests

- **Suites**

- Hasta ahora hemos podido ejecutar todos los test de forma automática desde el IDE.
- Podemos definir en un archivo XML que test ejecutar:

ejem1/testng.xml

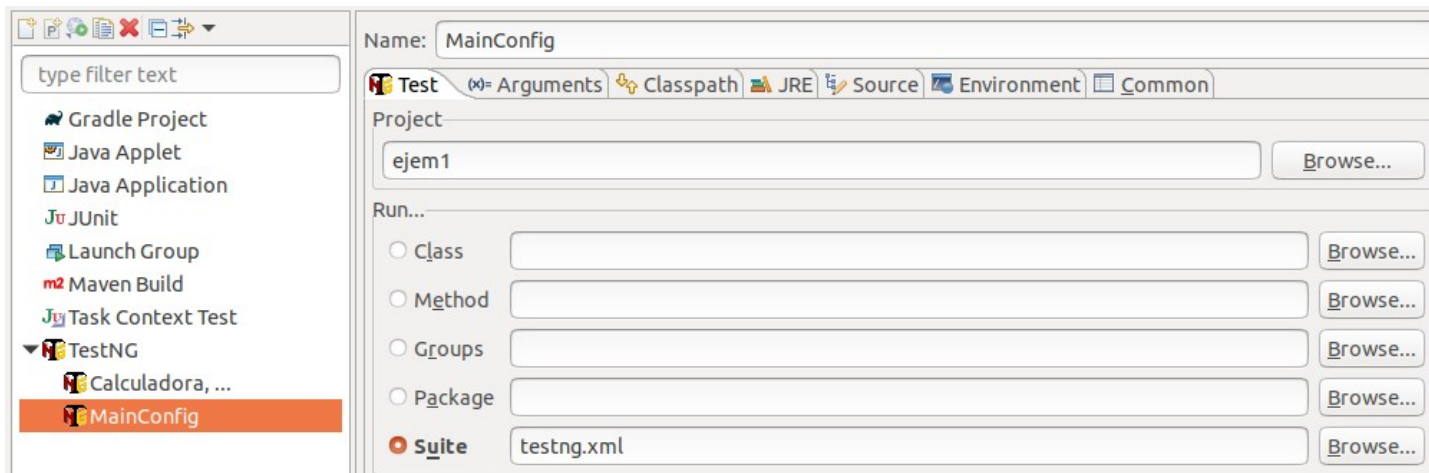
```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1" >
  <test name="Regression1">
    <packages>
      <package name="es.codeurjc.test.ejem.other" />
    </packages>
    <classes>
      <class name="es.codeurjc.test.ejem.AppTest" />
    </classes>
  </test>
</suite>
```

Podemos seleccionar
paquetes enteros

Podemos seleccionar
clases concretas

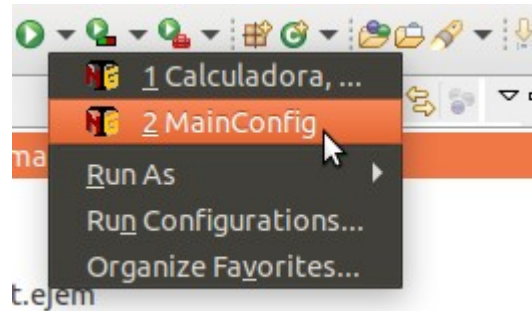
- **Suites**

- Para poder ejecutar los test definidos en el archivo, debemos crear una nueva configuración:
 - Click derecho sobre el proyecto > Run As... > Run configurations
 - Seleccionamos Suite y buscamos la configuración testng.xml
 - Click derecho sobre TestNG > New ConfigurationApply > Run



- **Suites**

- Podemos reutilizar esta configuración desde el IDE



- Es posible definir más de un archivo de configuración.
- La primera vez que lanzamos los test de un proyecto, crea una configuración por defecto que incluye todas las clases de test con sus métodos.

- **Test groups**

- Podemos asignar grupos a los métodos de test para poder ejecutar solo los de un mismo grupo.

ejem1/src/test/java/es/codeurjc/test/ejem/GroupsTest.java

```
public class GroupsTest {
    @Test(groups = { "suma" })
    public void testSuma() {
        int sum = 1 + 1;
        assertTrue(sum == 2);
    }

    @Test(groups = { "resta", "suma" })
    public void testSumaYResta() {
        int sum = 1 + 1;
        assertTrue(sum == 2);
        int res = 1 - 1;
        assertTrue(res == 0);
    }

    @Test(groups = { "resta" })
    public void testResta() {
        int res = 1 - 1;
        assertTrue(res == 0);
    }
}
```

ejem1/SumaGroupTest.xml

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="SumaTestSuite" verbose="1" >
    <test name="TestSuma">
        <groups>
            <run>
                <include name="suma"/>
            </run>
        </groups>
        <classes>
            <class name="es.codeurjc.test.ejem.GroupsTest"/>
        </classes>
    </test>
</suite>
```

- Introducción
- Test y Suites
- **Parámetros**
- Aserciones
- Dobles
- Paralelismo
- Interceptores y Listeners
- Resultados
- Conclusiones

- **Tests parametrizados**
 - En ciertas ocasiones es útil escribir un único test pero que se ejecute con varios datos
 - Se podría recorrer una estructura de datos en el propio test, pero si un valor falla, no se ejecutan los demás valores
 - Lo ideal es que cada conjunto de datos se identifique como un test, de forma que si falla, el mensaje indique el caso que ha fallado de forma independiente

- **Tests parametrizados**
 - La clase tiene que tener un método estático para generar los valores
 - Tiene que estar anotado con `@DataProvider(name = "anyName")`
 - Tiene que devolver un array de arrays de `Object(Object[][])` o un iterador de arrays de `Object (Iterator<Object[]>)`
 - Cada array contiene todos los valores usados por cada ejecución del test
 - Se indicará en que metodos Test se quiere utilizar este conjunto de datos.
 - El test usará los atributos en su implementación

- Tests parametrizados

ejem1/src/test/java/es/codeurjc/test/ejem/SumTest.java

```
public class SumTest {

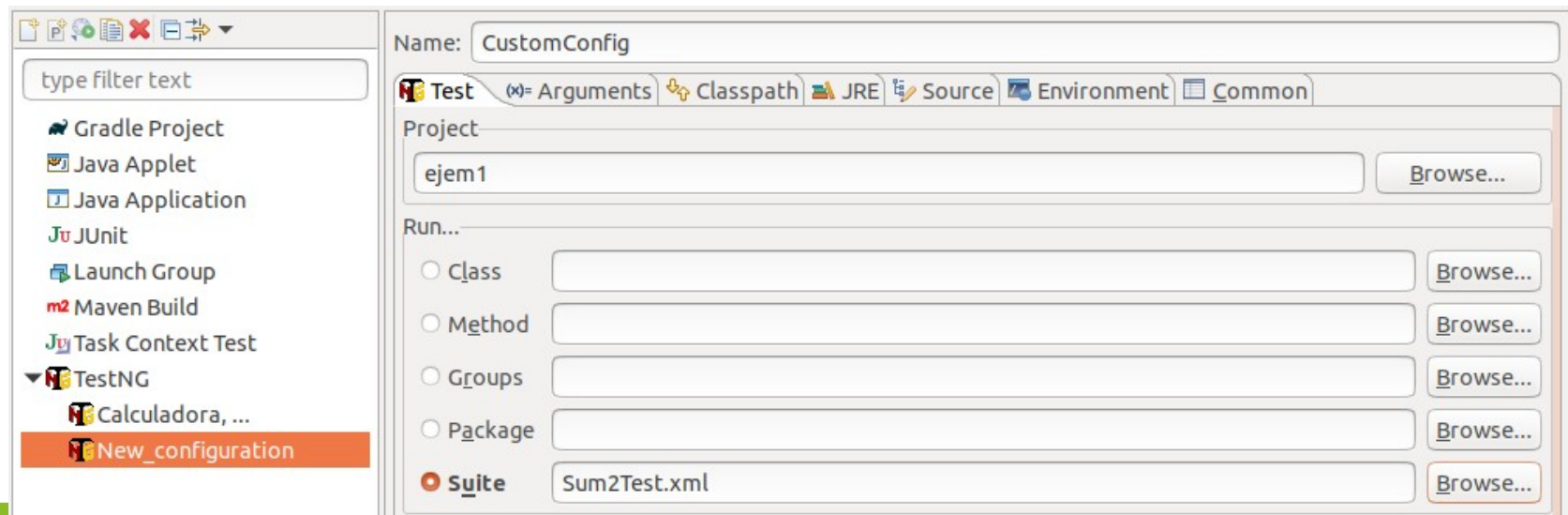
    @DataProvider(name = "numbers")
    public static Object[][] data() {

        Object[][] values = {
            { 0, 0, 0 },
            { 1, 1, 2 },
            { 2, 3, 5 },
            { 5, 4, 9 }
        };
        return values;
    }

    @Test(dataProvider = "numbers")
    public void test(int opA, int opB, int result) {
        assertEquals(result, opA + opB);
    }
}
```

- **Ejercicio 3**
 - Implementa un test que verifique que el valor absoluto de un número complejo (**método `abs()`**) se calcula correctamente
 - Define **varios número complejos** de ejemplo y verifica el cálculo para ellos
 - Usa tests parametrizados

- **Tests parametrizados**
 - Es posible definir parámetros en un archivo XML y utilizarlos en nuestra clase.
 - Click derecho sobre el proyecto > Run As... > Run configurations
 - Seleccionamos Suite y buscamos la configuración Sum2Test.xml
 - Click derecho sobre TestNG > New ConfigurationApply > Run



- Tests parametrizados

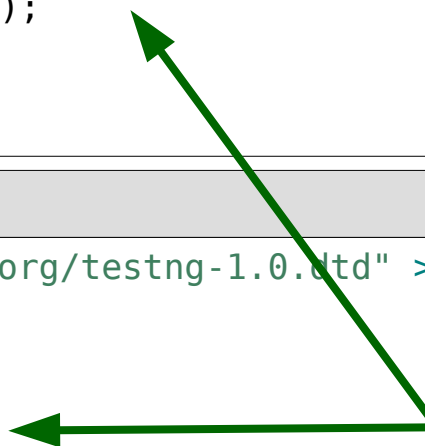
ejem1/src/test/java/es/codeurjc/test/ejem/Sum2Test.java

```
public class Sum2Test {  
  
    @Test  
    @Parameters({ "opA", "opB", "result" })  
    public void test(int opA, int opB, int result) {  
        assertEquals(opA + opB, result);  
    }  
  
}
```

ejem1/src/Sum2Test.xml

```
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >  
<suite name="Suite1" verbose="1" >  
    <parameter name="opA" value="2"/>  
    <parameter name="opB" value="4"/>  
    <parameter name="result" value="6"/>  
    <test name="Example">  
        <classes>  
            <class name="es.codeurjc.test.ejem.Sum2Test"/>  
        </classes>  
    </test>  
</suite>
```

Convierte String a Int



- Introducción
- Test y Suites
- Parámetros
- **Aserciones**
- Dobles
- Paralelismo
- Interceptores y Listeners
- Resultados
- Conclusiones

- Uno de los objetivos de las librerías de testing es que los **tests** sean **fáciles de escribir y de leer**
- Para ello, las **aserciones** no tienen que ser complejas
- La mayor parte de la innovación en librerías de testing ha ido en la **facilidad de escribir aserciones**, especialmente las específicas de un dominio concreto (**estructuras de datos, JSON, BBDD...**)

- Por ejemplo, al verificar que una **excepción** se lanza cuando debería hacerlo
- Los tests también tienen que verificar que el software **genera los errores esperados**
- El test debería fallar si no se genera la excepción o es de un tipo diferente al esperado

- Se puede implementar con **try/catch** y **fail()**
- Pero el test no es muy claro

ejem1/src/test/java/es/codeurjc/test/ejem/ExceptionTest.java

```
public class ExceptionTest {  
  
    @Test  
    public void arrayExceptionTest() {  
        int[] array = {3,4,2};  
        try {  
            int value = array[4];  
            fail("Array access out of range should "  
                + "throw ArrayIndexOutOfBoundsException");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            // Test should fail  
        }  
    }  
}
```

- TestNG permite especificar si se espera una excepción en el test
- El test queda mucho más **sencillo y conciso**

ejem1/src/test/java/es/codeurjc/test/ejem/ExceptionTest.java

```
public class ExceptionTest {  
  
    @Test(expectedExceptions=ArrayIndexOutOfBoundsException.class)  
    public void arrayExceptionTest2() {  
        int[] array = {3,4,2};  
        int value = array[4];  
    }  
}
```

• Ejercicio 4

- Implementar un test que verifique que el recíproco de `Zero Complex(0,0)` eleva una excepción `ArithmeticException` con el mensaje “division by zero”
- Como ahora mismo la clase `Complex` no está implementada de esa forma, el test debería fallar
- Modifica la clase `Complex` para que realmente eleve una excepción cuando se intente calcular el recíproco de `Complex(0,0)`
- Después de modificar el SUT, el test debería pasar

- **Mejorar los mensajes de error**
 - Cuanto más claros y más información aporten los mensajes en caso de fallo del test, más rápido se solucionará el bug que lo origina
 - Existen diversas formas de mejorar los mensajes:
 - Escribir el mensaje de la aserción en caso de fallo
 - Usar las aserciones más concretas
 - Usar *"matchers"*

- Escribir el mensaje de la aserción en caso de fallo

```
@Test
public void testSuma() {

    Calculadora calculadora = new Calculadora();
    int res = calculadora.suma(1, 1);
    assertEquals(2, res, "1+1 should be equal to 2");
}
```

- Es una tarea tediosa y repetitiva, y al final los mensajes quedan desactualizados
- Es la última opción cuando todas las demás no son válidas

- Usar las aserciones más concretas
 - Existe una aserción general **assertTrue** que debería evitarse si existe una aserción más específica
 - Si se usa **assertTrue**, en caso de fallo, JUnit sólo podrá mostrar la expresión y el resultado de la misma, pero no los valores individuales
 - No obstante, hay expresiones para las que no se dispone de **assert** específico

```
assertTrue(name.startsWith("ju"));
```

- Usar las aserciones más concretas

```
assertTrue(name.equals("juan"));
```



java.lang.AssertionError

Sólo muestra que
la aserción no es
correcta

```
assertEquals(name, "juan");
```



org.junit.ComparisonFailure:
expected:<[pepe]> but
was:<[juan]>

Muestra el valor de
cada operando

- Usar “*matchers*”
 - Un *matcher* es una función que verifica si un valor cumple una determinada propiedad:
 - Es un String y cumple una expresión regular
 - Es una lista y contiene algunos elementos
 - Es una lista vacía
 - Es igual que otro valor...
 - Se usan con la función **assertThat**
 - Además de mejorar el mensaje, mejora la legibilidad del test (similar a lenguaje natural)

- Ejemplos

```
assertThat(x, is(3));  
assertThat(x, is(not(4)));  
assertThat(name, equalTo("juan"));  
assertThat(name, startsWith("ju"));  
assertThat(name, containsString("ju"));  
assertThat(list, hasItem("elem"));
```

- De forma genérica

```
assertThat([value], [matcher statement]);
```

- Diferencias entre mensajes de error

```
assertTrue(name.startsWith("ju"));
```

java.lang.AssertionError

Sólo muestra que
la aserción no es
correcta

```
assertThat(name, startsWith("ju"));
```

java.lang.AssertionError:
Expected: a string starting with "ju"
but: was "pepe"

Muestra más
información del
error

Aserciones - Matchers

```
ejem2/src/test/java/es/codeurjc/test/ejem/HamcrestTest.java
```

```
package es.codeurjc.test.ejem;

import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.startsWith;
import static org.hamcrest.MatcherAssert.assertThat;

import org.testng.annotations.Test;

public class HamcrestTest {

    @Test
    public void testAssertThatEquals() {
        assertThat("pepe", equalTo("juan"));
    }

    @Test
    public void testAssertStartsWith() {
        assertThat("pepe", startsWith("ju"));
    }
}
```

- Los matchers que acabamos de ver pertenecen a la librería Hamcrest
- Paquete: org.hamcrest.CoreMatchers



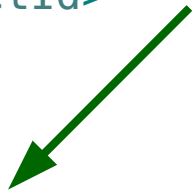
Java Hamcrest

- <http://hamcrest.org/JavaHamcrest/>
- <http://www.vogella.com/tutorials/Hamcrest/article.html>

Aserciones – Hamcrest

- Para poder usar la librería se incluye como dependencia en el pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.10</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest</artifactId>
    <version>2.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



- **Ejercicio 5**
 - Cambia las aserciones del Ejercicio 4 para que usen los matchers adecuados

- **Aspectos avanzados de Hamcrest**
 - Se pueden crear matchers personalizados para mejorar la legibilidad de los tests y los mensajes de error
 - Existen matchers creados por terceros
 - **Excel:**
 - <https://github.com/tobyweston/simple-excel>
 - **JSON:**
 - <https://github.com/hertzprung/hamcrest-json>
 - **XML:**
 - <https://github.com/davidehringer/xml-matchers>

- Aspectos avanzados de Hamcrest
 - Se pueden combinar varios matchers en expresiones

```
assertTrue(  
    responseString.contains("color") || responseString.contains("colour"));  
// ==> failure message:  
// java.lang.AssertionError:
```

```
assertThat(responseString,  
    anyOf(containsString("color"), containsString("colour")));  
// ==> failure message:  
// java.lang.AssertionError:  
// Expected: (a string containing "color" or a string containing "colour")  
//          got: "Please choose a font"
```

```
assertThat(responseString,  
    either(containsString("color")).or(containsString("colour")));
```


- La librería **AssertJ** permite escribir matchers con un enfoque diferente a Hamcrest
- Es más sencilla de usar porque se integra mucho mejor con los IDEs y el autocompletar

AssertJ

Fluent assertions for java

<http://joel-costigliola.github.io/assertj/index.html>

- Dispone de un método **assertThat(value)** que recibe el objeto a verificar
- Se encadenan aserciones que dependen de la clase del objeto pasado como parámetro

ejem2/src/test/java/es/codeurjc/test/ejem/AssertJTest.java

```
package es.codeurjc.test.ejem;

import static org.assertj.core.api.Assertions.*;
import static org.testng.Assert.*;
import org.testng.annotations.Test;
public class AssertJTest {

    @Test
    public void testAssertTrue() {
        assertThat("pepe").isEqualTo("juan");
    }
}
```

- Existen aserciones predefinidas para tipos primitivos, Strings, colecciones...

```
assertThat(fellowshipOfTheRing).hasSize(9)
                                .contains(frodo, sam)
                                .doesNotContain(sauron);
```

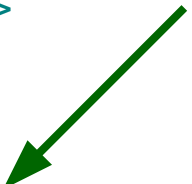
- Aserción de excepciones

```
assertThatThrownBy(() -> {
    //Code that launches exception
}).isInstanceOf(Exception.class).hasMessageContaining("..");
```

- Se pueden incluir aserciones personalizadas

- Para poder usar la librería se incluye como dependencia en el pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.10</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.11.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



- **Ejercicio 6**

- Convierte el Ejercicio 5 para que use la librería AssertJ
- Cambia los matchers de Hamcrest por aserciones de AssertJ
- Verifica la excepción con AssertJ

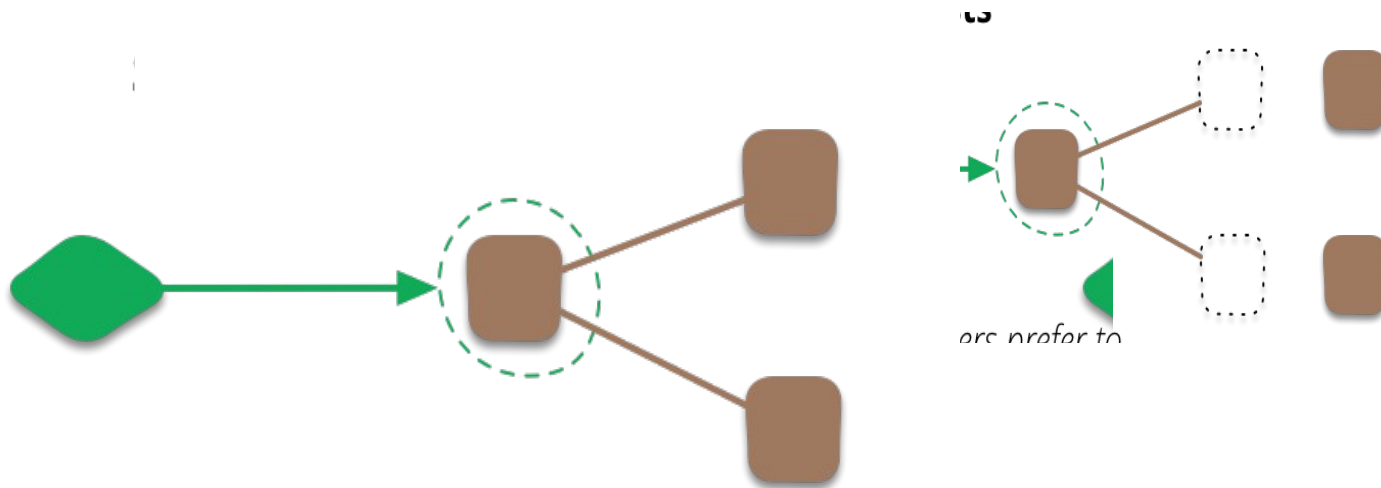
- Introducción
- Test y Suites
- Parámetros
- Aserciones
- **Dobles**
- Paralelismo
- Interceptores y Listeners
- Resultados
- Conclusiones

- Cualquier software tiene **clases que dependen de otras clases**
- Al implementar un **test** para probar una clase, a la clase se la conoce como **SUT** (***subject under test***) y a cada una de las clases de las que depende **DOC** (***depended-on component***)
- A veces, los DOCs hacen difícil **probar el SUT de forma aislada**

- **Limitaciones en un DOC para probar un SUT:**
 - **Lento:** Realiza cálculos extensivos, acceso a bases de datos o ficheros.
 - **No determinista:** Suministra al SUT valores de entrada no deterministas, descontrolados (aleatorio, hora, señal de sensor, ...).
 - **Difícil de automatizar:** Porque es una interfaz de usuario o porque se quieren simular comportamientos difíciles de conseguir (poco ancho de banda, fallo 500 es un servidor...)
 - **Preparación costosa:** Porque su construcción es costosa en tiempo, memoria, CPU.
 - **Acoplado a otros sistemas:** Es parte de otro componente (p.e. librería de acceso a otro servidor vía tcp/ip) o no debe cambiar su estado (p.e. un servicio de correo)
 - **No está disponible:** porque su implementación no está disponible

- Para solventar estos problemas se usan **dobles**
- Los **dobles** son objetos que se usan como **sustitutos de los DOCs reales** cuando no es conveniente usarlos
- El nombre de **doble** (*double*) proviene de los **dobles de las películas**, que se usan en las escenas en las que no es conveniente que participe el actor real
- Además, el uso de dobles permite una **adaptación del DOC a las necesidades del test**

- Los dobles permiten implementar tests solitarios o tests sociables pero con algunas dependencias sustituidas por dobles



Test unitario sociable con dependencias transitivas sustituidas por dobles

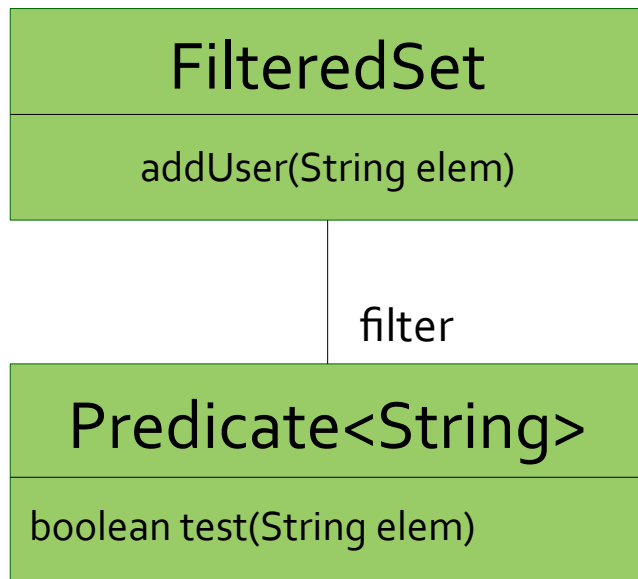
- Los **dobles** son objetos que **sustituyen** las dependencias reales de una clase
- Un doble de un **DOC** tiene que comportarse a ojos del **SUT** como el **DOC** original.
- En un lenguaje tipado como Java, además **debe tener el mismo tipo**.
- En Java los dobles de los **DOCs** **se implementan**:
 - Si el **SUT** depende de una **interfaz**, creando una nueva **implementación** para el doble que implemente la interfaz
 - Si no, con **librerías especiales** que permiten redefinir el comportamiento de una clase durante la ejecución de un test

- **Tipos de dobles:**

- **Dummy:** DOC que no se va a ser usado por el SUT en el test, pero se utiliza para que el código compile
- **Fake:** Objeto que se comporta como el real, pero con una implementación más simple y más rápida para tests (p.e. Base de datos en memoria)
- **Spy:** Proxy de un DOC real que permite verificar en el test qué métodos han sido llamados, con qué parámetros y valor devuelto
- **Stubs:** Objetos cuyo comportamiento se define en el test con respuestas predefinidas para sus métodos.
- **Mocks:** Iguales a los stubs, pero que además se verifica que los métodos especificados han sido llamados

- **Cómo se usa un mock o un stub en un test**
 - Se define una **variable** para el DOC
 - Se crea el **mock o stub** y se define el comportamiento de aquellos métodos que vayan a ser usados por el SUT durante el test
 - Se **pasa el doble al SUT** en el constructor o en un método
 - Se **ejercita** el SUT (para que use el DOC)
 - El test verifica que los resultados de ejercitar el **SUT son los esperados**
 - Si el test **verifica también que los métodos del doble han sido llamados**, el doble será un mock. Si no, será un stub.

- Ejemplo



```
ejem3/src/main/java/es/codeurjc/test/ejem/FilteredSet.java
public class FilteredSet extends HashSet<String> {

    private Predicate<String> filter;

    public FilteredSet(Predicate<String> filter){
        this.filter = filter;
    }

    @Override
    public boolean add(String element) {
        if(filter.test(element)) {
            return super.add(element);
        } else {
            return false;
        }
    }
}
```

ejem3/src/test/java/es/codeurjc/test/ejem/FilteredSetTest.java

```
public class FilteredSetTest {

    @Test
    public void notAllowedElementTest() {

        Predicate<String> filter = elem -> false;

        FilteredSet fSet = new FilteredSet(filter);
        boolean changed = fSet.add("elem");

        assertThat(changed).isFalse();
        assertThat(fSet.size()).isEqualTo(0);
    }

    @Test
    public void allowedElementTest() {

        Predicate<String> filter = elem -> true;

        FilteredSet fSet = new FilteredSet(filter);
        boolean changed = fSet.add("elem");

        assertThat(changed).isTrue();
        assertThat(fSet.size()).isEqualTo(1);
    }
}
```

Doble que
sustituye al DOC

Es un **stub** porque
no se verifica que
se llame al método
test

No hace falta librería porque
el DOC implementa el
interfaz Predicate

- **Librería de dobles en Java**

- En Java existen varias librerías para la creación de dobles: Mockito, JMockit, EasyMock, PowerMock...
- Nosotros usaremos **Mockito** por ser la más extendida



<http://mockito.org/>

- Mockito es una librería para construir **stubs, mocks y spies**
- Para sustituir el comportamiento de métodos estáticos y constructores se usa la librería **PowerMock**
- Para hacer un **mock o un stub** de una clase Java se ejecuta el método estático **mock(...)** pasando la clase o interfaz

```
import static org.mockito.Mockito.*;  
  
User user = mock(User.class);
```

- Se define el comportamiento de los métodos del mock
- Métodos sin parámetros que devuelven un valor

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

...
User user = mock(User.class);

when(user.getName())
    .thenReturn("Pepe").thenReturn("Juan");

when(user.getAndIncAge())
    .thenThrow(new RuntimeException("Max age"));
```

- Se define el comportamiento de los métodos del mock
- Métodos sin parámetros que devuelven **void**

```
import static org.mockito.Mockito.*;  
  
...  
  
User user = mock(User.class);  
  
doThrow(new RuntimeException())  
    .when(user).saveToDisk();
```

- Se define el comportamiento de los métodos del mock
 - Si el método recibe parámetros, se indica el valor del parámetro o un *matcher*
 - Cuando el método recibe parámetros, se pueden configurar respuestas diferentes para cada valor

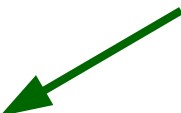
```
Manager manager = mock(Manager.class);  
  
when(manager.get("lang")).thenReturn("es");  
  
when(manager.get(anyString()))  
    .thenThrow(new RuntimeException("Not value"));
```

- Se define el comportamiento de los métodos del mock
- Los métodos pueden devolver otros mocks

```
Chat chat = mock(Chat.class);  
User user = mock(User.class);  
  
when(chat.getUser("Juan")).thenReturn(null);  
  
when(chat.getUser(anyString())).thenReturn(user);
```

- El test puede verificar **qué métodos han sido invocados** (y los parámetros usados)
- **Es una aserción del test.** El test fallará si los métodos no han sido invocados

```
User user = mock(User.class);  
when(user.getName()).thenReturn("Pepe");  
  
Chat chat = mock(Chat.class);  
when(chat.getUser(anyString())).thenReturn(user);  
...  
  
verify(user).getName();  
verify(chat).getUser("Juan");
```



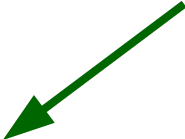
- Se puede verificar el **número de invocaciones** en función de los parámetros

```
verify(chat).getUser("Juan");  
  
verify(chat, times(1)).getUser("Pepe");  
verify(chat, times(2)).getUser("Juan");  
  
verify(chat, never()).getUser(anyString());  
  
verify(chat, atLeastOnce()).getUser("Bob");  
verify(chat, atLeast(2)).getUser("Susan");  
verify(chat, atMost(5)).getUser("MrX2");
```

- En Mockito los **mocks y stubs** se crean y configuran de la misma forma.
- Se considera **mock** si se entre las aserciones del test se **verifican las llamadas a métodos**
- Se considera **stub** si las aserciones del test **no verifican** que los métodos hayan sido llamados

- Para poder usar la librería Mockito se añade una dependencia al **pom.xml**

```
<dependencies>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.10</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.13.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



Dobles

ejem3/src/test/java/es/codeurjc/test/ejem/FilteredSetTest.java

```
@Test
public void notAllowedElementTest() {

    Predicate<String> filter = elem -> false;

    FilteredSet fSet = new FilteredSet(filter);
    boolean changed = fSet.add("elem");

    assertThat(changed).isFalse();
    assertThat(fSet.size()).isEqualTo(0);
}
```

ejem3/src/test/java/es/codeurjc/test/ejem/FilteredSetMockTest.java

```
@Test
public void notAllowedElementTest() {

    Predicate<String> filter = mock(Predicate.class);
    when(filter.test(any())).thenReturn(false);

    FilteredSet fSet = new FilteredSet(filter);
    boolean changed = fSet.add("elem");

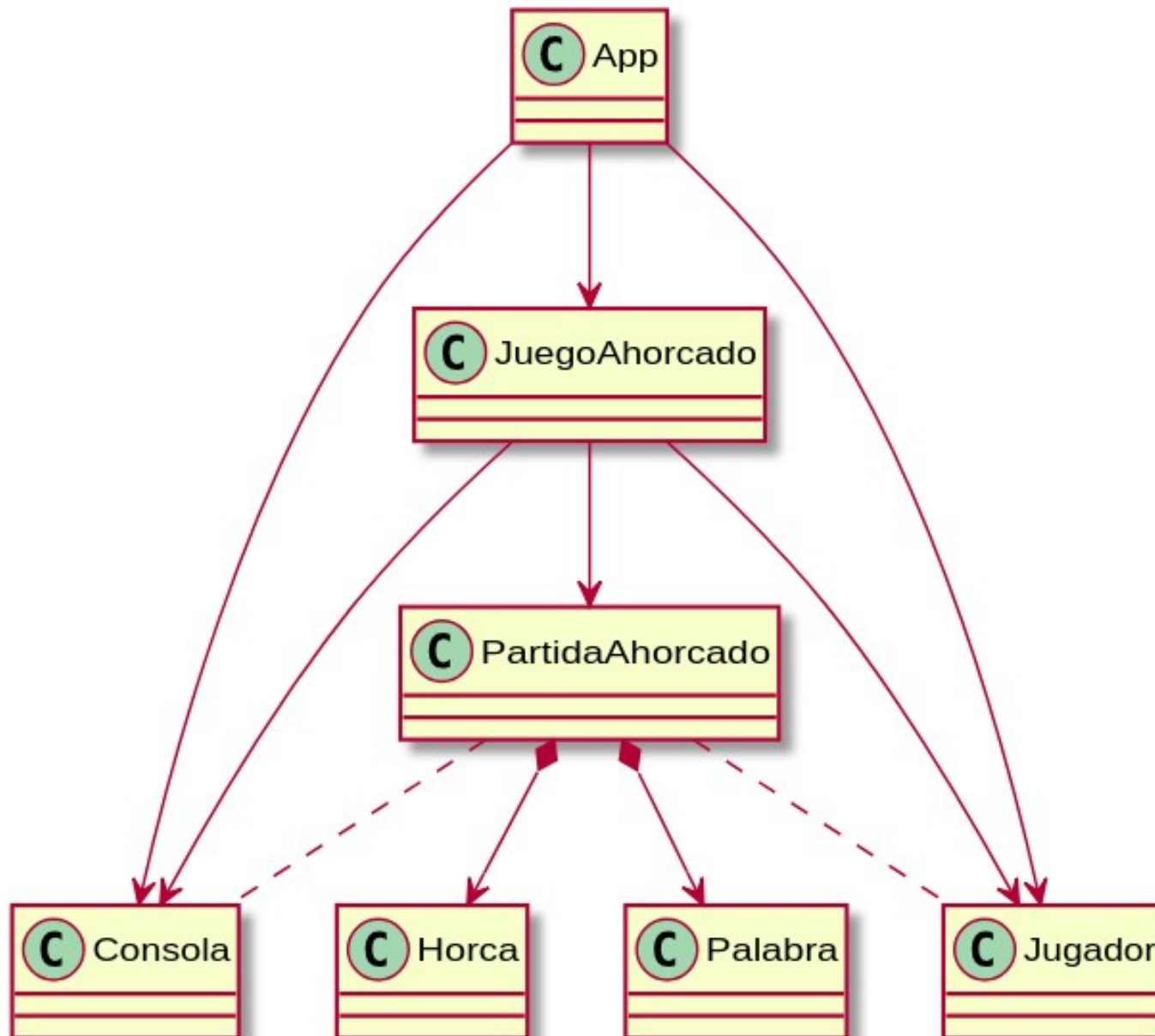
    verify(filter).test("elem");

    assertThat(changed).isFalse();
    assertThat(fSet.size()).isEqualTo(0);
}
```

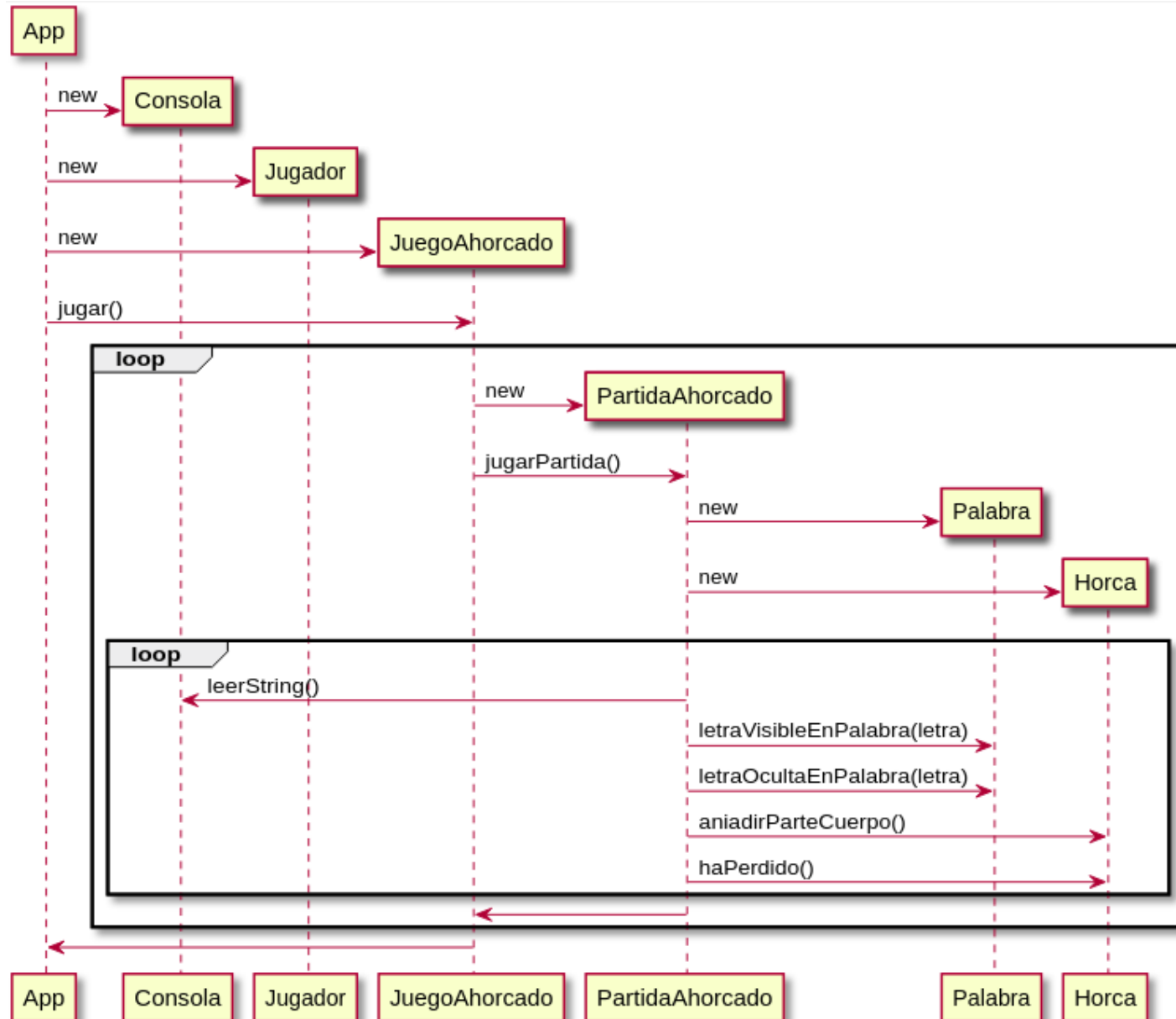
**Doble que
sustituye al DOC**

Es un **mock** porque
se verifica que se
llame al método
test

Ejemplo: Juego del Ahorcado



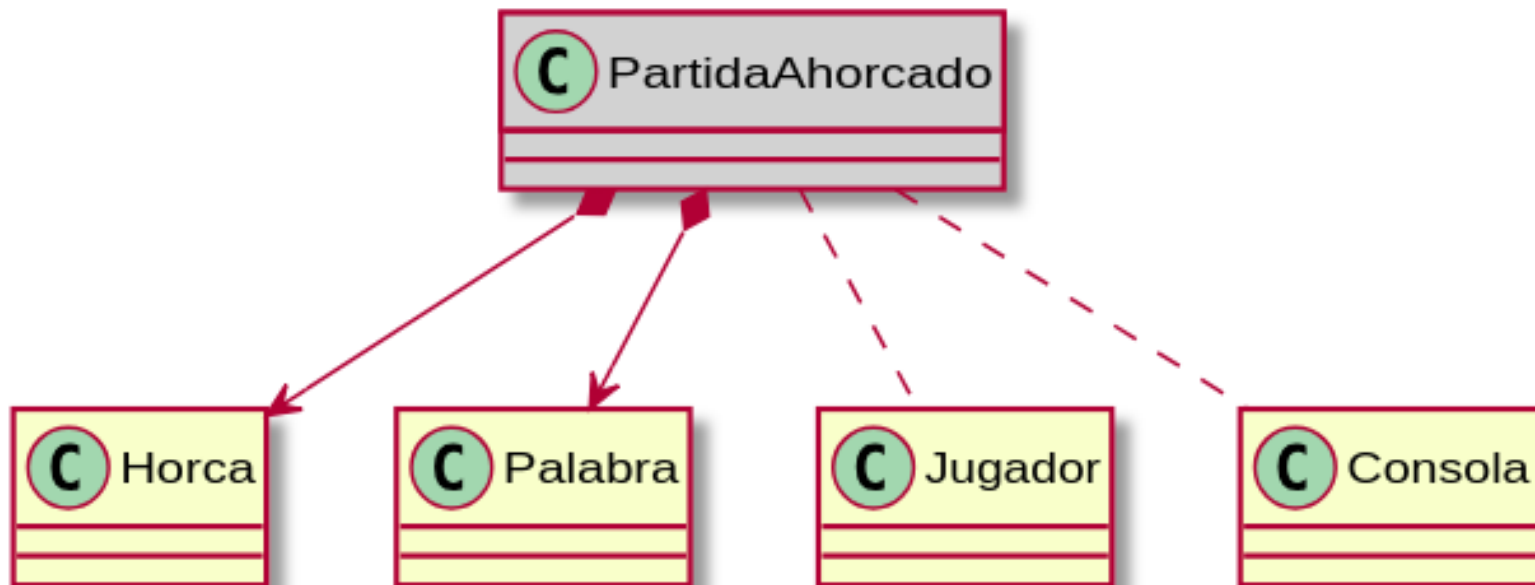
Ejemplo: Juego del Ahorcado



Ejemplo: Juego del Ahorcado

- **Test unitario**

- **SUT:** PartidaAhorcado
- **DOCs:** Consola, Palabra, Jugador y Horca



- **Test unitario sociable de PartidaAhorcado**

Creamos un **mock de la consola** definiendo las entradas del usuario

```
@Test
public void testPartidaGanada() {
    // Given
    Consola consola = mock(Consola.class);
    when(consola.leerString()).thenReturn("T", "E", "S", "T");
    Jugador jugador = new Jugador(consola);
    PartidaAhorcado partida = new PartidaAhorcado("TEST");
    // When
    Resultado resultado = partida.jugarPartida(jugador);
    // Then
    assertThat(resultado).isEqualTo(Resultado.PALABRA_ACERTADA);
    verify(consola, times(3)).leerString();
}
```

- Test unitario solitario de PartidaAhorcado

```
@Test
public void testPartidaGanada() {

    //Given
    Consola consola = mock(Consola.class);
    when(consola.leerString()).thenReturn("A");

    Jugador jugador = mock(Jugador.class);
    when(jugador.getNombre()).thenReturn("X");
    when(jugador.getConsola()).thenReturn(consola);

    Horca horca = mock(Horca.class);
    when(horca.haPerdido()).thenReturn(false);

    Palabra palabra = mock(Palabra.class);
    when(palabra.letraOcultaEnPalabra(any())).thenReturn(true);
    when(palabra.palabraCompleta()).thenReturn(false, false, false, true);

    PartidaAhorcado partida = new PartidaAhorcado(palabra, horca);

    //When
    Resultado resultado = partida.jugarPartida(jugador);

    //Then
    assertThat(resultado).isEqualTo(Resultado.PALABRA_ACERTADA);

    verify(consola, times(4)).leerString();
    verify(horca, never()).aniadirParteCuerpo();
    verify(horca, never()).haPerdido();
    verify(palabra, times(4)).palabraCompleta();
}
```

- **Test unitarios sociables vs solitarios**
 - Hay que evitar que los tests sean frágiles
 - Si cada vez que refactorizamos tenemos que reimplementar los tests, no nos ayudan, nos entorpecen
 - Los tests sociables son menos frágiles y favorecen la refactorización
 - Es imprescindible sustituir por dobles los DOCs costosos (E/S, algoritmos...), no deterministas, difíciles de controlar (Consola)...

- **Mocks vs Stubs**

- La **verificación de las llamadas** a métodos de los dobles puede dar lugar a **tests más frágiles** si realmente depende de detalles de implementación de la clase
- Es **adecuado verificar** cuando el uso de un doble forma parte de los **requisitos del SUT** (p.e. enviar una alarma, guardar un objeto, etc.) que no se puede verificar observando el estado del SUT

- Introducción
- Test y Suites
- Parámetros
- Aserciones
- Dobles
- **Paralelismo**
- Interceptores y Listeners
- Resultados
- Conclusiones

- Podemos especificar en TestNG como ejecutar nuestros test en diferentes Threads, de forma que se ejecuten en paralelo.

ejem1/src/test/java/es/codeurjc/test/ejem/ParallelTest.java

```
public class ParallelTest {  
  
    @Test(threadPoolSize = 3, invocationCount = 10, timeout = 10000)  
    public void testSuma() {  
  
        int randomInt = new Random().nextInt();  
  
        // Arrange / Given  
        Calculadora calculadora = new Calculadora();  
  
        // Act / When  
        int res = calculadora.suma(randomInt, randomInt);  
  
        // Assert / Then  
        assertEquals(2*randomInt, res, 0);  
    }  
}
```

Se ejecutaran **10** test, hasta **3** test al mismo tiempo.

```
[RemoteTestNG] detected TestNG version 6.10.0  
[TestNG] Running:  
[ThreadUtil] Starting executor timeout:10000ms  
workers:10 threadPoolSize:3  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma  
PASSED: testSuma
```

```
=====  
Default test  
Tests run: 10, Failures: 0, Skips: 0  
=====
```

- También podemos paralelizar los test parametrizados de forma sencilla.

ejem1/src/test/java/es/codeurjc/test/ejem/Parallel2Test.java

```
public class Parallel2Test {

    @DataProvider(name = "numbers", parallel = true)
    public static Object[][] data() {

        Object[][] values = {
            { 0, 0, 0 },
            { 1, 1, 2 },
            { 2, 3, 5 },
            { 5, 4, 9 }
        };
        return values;
    }

    @Test(dataProvider = "numbers")
    public void test(int opA, int opB, int result) {
        System.out.println("Current Thread: "+Thread.currentThread().getId());
        assertEquals(opA + opB, result);
    }

}
```

Por defecto usará un pool de Threads de tamaño 10

- En la configuración de los suites que definimos en XML podemos establecer paralelismo a diferentes niveles:
- **De método:** Cada método se ejecutará en un Thread distinto

```
<suite name="My suite" parallel="methods" thread-count="5">
```

- **De clase:** Todos los métodos de una clase se ejecutaran en el mismo Thread. Habrá un Thread por clase.

```
<suite name="My suite" parallel="classes" thread-count="5">
```

- Introducción
- Test y Suites
- Parámetros
- Aserciones
- Dobles
- Paralelismo
- **Interceptores y Listeners**
- Resultados
- Conclusiones

- En TestNG podemos crear interceptores, clases que nos permiten modificar el orden y filtrar los test que se van a ejecutar.
- Los interceptores solo actuarán sobre test que no tengan dependencias o se ejecuten en un orden por defecto.
- Definimos una clase que extienda la interfaz ***IMethodInterceptor***.
- Implementamos el método ***intercept(List<IMethodInstance> methods, ITestContext context)***:
 - Recibimos una lista de Test
 - Creamos una nueva lista de test (distinto orden y/o filtrando algunos)
 - Devolvemos la lista de Test
- Creamos un archivo XML que utilice nuestro nuevo interceptor.

Interceptores

ejem4/src/test/java/es/codeurjc/test/ejem/AdditionTest.java

```
package es.codeurjc.test.ejem;

import static org.testng.Assert.assertEquals;
import org.testng.annotations.Test;

public class AdditionTest {

    @Test(priority=1, groups="smoke")
    public void example1Test() {
        System.out.println("example1Test()");
        assertEquals(1+1, 2);
    }

    @Test(priority=2, groups="UI")
    public void example2Test() {
        System.out.println("example2Test()");
        assertEquals(2+2, 4);
    }

    @Test(priority=3, groups="regression")
    public void example3Test() {
        System.out.println("example3Test()");
        assertEquals(3+3, 6);
    }
}
```

ejem4/src/test/java/es/codeurjc/test/ejem/SubstractTest.java

```
package es.codeurjc.test.ejem;

import static org.testng.Assert.assertEquals;
import org.testng.annotations.Test;

public class SubstractTest {

    @Test(priority=1, groups="smoke")
    public void example1Test() {
        System.out.println("example1Test()");
        assertEquals(1-1, 0);
    }

    @Test(priority=2, groups="UI")
    public void example2Test() {
        System.out.println("example2Test()");
        assertEquals(2-2, 0);
    }

    @Test(priority=3, groups="regression")
    public void example3Test() {
        System.out.println("example3Test()");
        assertEquals(3-3, 0);
    }
}
```


Interceptores

ejem4/src/test/java/es/codeurjc/test/ejem/CalculatorInterceptor.java

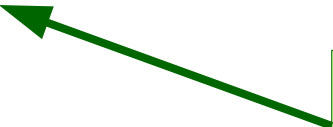
```
package es.codeurjc.test.ejem;

import java.util.ArrayList;
import java.util.List;

import org.testng.IMethodInstance;
import org.testng.IMethodInterceptor;
import org.testng.ITestContext;
import org.testng.annotations.Test;

public class CalculatorInterceptor implements IMethodInterceptor{

    @Override
    public List<IMethodInstance> intercept(List<IMethodInstance> methods, ITestContext context) {
        List<IMethodInstance> result = new ArrayList<IMethodInstance>();
        for (IMethodInstance method : methods) {
            Test testMethod = method.getMethod()
                .getConstructorOrMethod().getMethod().getAnnotation(Test.class);
            if (testMethod.priority() <= 2) {
                result.add(method);
            }
        }
        return result;
    }
}
```



Filtramos los test con prioridad
igual o menor que 2

Interceptores

ejem4/interceptor-config.xml

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Suite1" verbose="1">
  <listeners>
    <listener class-name="es.codeurjc.test.ejem.CalculatorInterceptor" />
  </listeners>
  <test name="CalculatorTest">
    <classes>
      <class name="es.codeurjc.test.ejem.AdditionTest" />
      <class name="es.codeurjc.test.ejem.SubtractTest" />
    </classes>
  </test>
</suite>
```

```
[RemoteTestNG] detected TestNG version 6.10.0
[TestNG] Running:
  /home/ejem4/interceptor-config.xml
```

```
example1Test()
example1Test()
example2Test()
example2Test()
```

```
=====
Suite1
Total tests run: 4, Failures: 0, Skips: 0
=====
```

- **Ejercicio 7**

- A partir del ejemplo anterior, crea un nuevo interceptor que asegure que los test de grupo UI se ejecuten los últimos.
- Notas:
 - De forma previa, hay que eliminar las prioridades fijadas.
 - Solo ordena los test dentro de una misma clase.

- Los Listener son clases que puede modificar el comportamiento de TestNG.
- Los interceptores son un tipo concreto de listeners.
- TestNG define múltiples interfaces para crearlos:
 - `IAnnotationTransformer`
 - `IAnnotationTransformer2`
 - `IHookable`
 - `IInvokedMethodListener`
 - **`IMethodInterceptor`**
 - `IReporter`
 - `ISuiteListener`
 - `ITestListener`

Listeners

```
ejem4/src/test/java/es/codeurjc/test/ejem/MyListener.java
```

```
package es.codeurjc.test.ejem;

import org.testng.ITestContext;
import org.testng.ITestListener;
import org.testng.ITestResult;

public class MyListener implements ITestListener{

    @Override
    public void onTestStart(ITestResult result) {
        System.out.println("Test start!");
    }

    @Override
    public void onTestSuccess(ITestResult result) {
        System.out.println("Test success!");
    }

    // Other methods

}
```

- Además de definirse en un fichero de configuración, pueden añadirse como anotación sobre una clase:

ejem4/src/test/java/es/codeurjc/test/ejem/Addition2Test.java

```
@Listeners({ MyListener.class })  
public class Addition2Test {  
    ...  
}
```

```
Test start!  
example1Test()  
Test success!  
Test start!  
example2Test()  
Test success!  
Test start!  
example3Test()  
Test success!
```

- Introducción
- Test y Suites
- Parámetros
- Aserciones
- Dobles
- Paralelismo
- Interceptores
- **Resultados**
- Conclusiones

- TestNG genera los resultados de los test en diferentes formatos para su consulta en la carpeta *test-output*.
- **Web:** abriendo el fichero HTML *test-output/index.html* en un navegador, nos ofrece una interfaz web dónde ver el resultado de los test.
- **JUnitReports:** genera ficheros XML en la carpeta *test-output/junitreports/* por cada clase de test en formato Junit.
- **TestNG Report:** genera un fichero XML con información adicional de TestNG que los JUnitReports no tienen.

Resultados

- Web Report

Test results

1 suite, 9 failed tests

All suites

Default suite

Info

- testng-customsuite.xml
- 1 test
- 2 groups
- Times
- Reporter output
- Ignored methods
- Chronological view

Results

- 45 methods, 9 failed, 36 passed
- Failed methods (hide)
 - test
 - testAssertEquals
 - testAssertEquals
 - testAssertStartsWith
 - testAssertStartsWith
 - testAssertThatEquals
 - testAssertThatStartsWith
 - testAssertTrue
 - testAssertTrue
- Passed methods (hide)
 - arrayExceptionTest
 - arrayExceptionTest2
 - test(2, 3, 5)
 - test(2, 3, 5)
 - test(0, 0, 0)
 - test(0, 0, 0)
 - test(1, 1, 2)
 - test(5, 4, 9)
 - test(5, 4, 9)
 - test(1, 1, 2)
 - test1
 - test2
 - testResta
 - testSuma

```
org.testng.TestNGException:
Parameter 'opA' is required by @Test on method test but has not been marked @Optional or defined
in /tmp/testng-eclipse--891014528/testng-customsuite.xml
    at org.testng.internal.Parameters.createParameters(Parameters.java:199)
    at org.testng.internal.Parameters.createParameters(Parameters.java:443)
    at org.testng.internal.Parameters.handleParameters(Parameters.java:568)
    at org.testng.internal.Invoker.handleParameters(Invoker.java:1293)
    at org.testng.internal.Invoker.createParameters(Invoker.java:1020)
    at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:1110)
    at org.testng.internal.TestMethodWorker.invokeTestMethods(TestMethodWorker.java:129)
    at org.testng.internal.TestMethodWorker.run(TestMethodWorker.java:112)
    at org.testng.TestRunner.privateRun(TestRunner.java:756)
    at org.testng.TestRunner.run(TestRunner.java:610)
    at org.testng.SuiteRunner.runTest(SuiteRunner.java:387)
    at org.testng.SuiteRunner.runSequentially(SuiteRunner.java:382)
    at org.testng.SuiteRunner.privateRun(SuiteRunner.java:340)
    at org.testng.SuiteRunner.run(SuiteRunner.java:289)
    at org.testng.SuiteRunnerWorker.runSuite(SuiteRunnerWorker.java:52)
    at org.testng.SuiteRunnerWorker.run(SuiteRunnerWorker.java:86)
    at org.testng.TestNG.runSuitesSequentially(TestNG.java:1293)
    at org.testng.TestNG.runSuitesLocally(TestNG.java:1218)
    at org.testng.TestNG.runSuites(TestNG.java:1133)
    at org.testng.TestNG.run(TestNG.java:1104)
    at org.testng.remote.AbstractRemoteTestNG.run(AbstractRemoteTestNG.java:115)
    at org.testng.remote.RemoteTestNG.initAndRun(RemoteTestNG.java:251)
    at org.testng.remote.RemoteTestNG.main(RemoteTestNG.java:77)
```

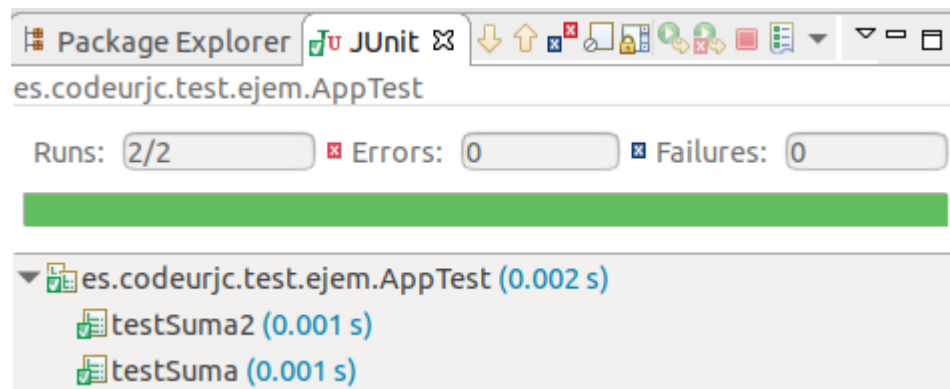
es.codeurjc.test.ejem.MatchersTest

```
testAssertEquals
java.lang.AssertionError: expected [juan] but found [pepe]
    at org.testng.Assert.fail(Assert.java:94)
    at org.testng.Assert.failNotEquals(Assert.java:513)
    at org.testng.Assert.assertEqualsImpl(Assert.java:135)
    at org.testng.Assert.assertEquals(Assert.java:116)
    at org.testng.Assert.assertEquals(Assert.java:190)
    at org.testng.Assert.assertEquals(Assert.java:200)
    at es.codeurjc.test.ejem.MatchersTest.testAssertEquals(MatchersTest.java:20)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
```

- JUnitReport (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by org.testng.reporters.JUnitReportReporter -->
<testsuite name="es.codeurjc.test.ejem.AppTest" tests="2" time="0.002" ignored="0"
failures="0" timestamp="24 Feb 2020 11:33:01 GMT" skipped="0" hostname="michel" errors="0">
  <testcase name="testSuma2" time="0.001" classname="es.codeurjc.test.ejem.AppTest"/>
  <testcase name="testSuma" time="0.001" classname="es.codeurjc.test.ejem.AppTest"/>
</testsuite> <!-- es.codeurjc.test.ejem.AppTest -->
```

- JUnitReport (Eclipse)



- TestNG Report

```
<?xml version="1.0" encoding="UTF-8"?>
<testng-results skipped="0" failed="9" ignored="0" total="45" passed="36">
  <reporter-output>
  </reporter-output>
  <suite name="Default suite" duration-ms="119" started-at="2020-02-24T11:33:01Z" finished-
at="2020-02-24T11:33:01Z">
    <groups>
      <group name="suma">
        <method signature="GroupsTest.testSuma()[pri:0,
instance:es.codeurjc.test.ejem.GroupsTest@548b7f67]" name="testSuma"
class="es.codeurjc.test.ejem.GroupsTest"/>
        <method signature="GroupsTest.testSumaYResta()[pri:0,
instance:es.codeurjc.test.ejem.GroupsTest@548b7f67]" name="testSumaYResta"
class="es.codeurjc.test.ejem.GroupsTest"/>
      </group> <!-- suma -->
      <group name="resta">
        <method signature="GroupsTest.testResta()[pri:0,
instance:es.codeurjc.test.ejem.GroupsTest@548b7f67]" name="testResta"
class="es.codeurjc.test.ejem.GroupsTest"/>
        <method signature="GroupsTest.testSumaYResta()[pri:0,
instance:es.codeurjc.test.ejem.GroupsTest@548b7f67]" name="testSumaYResta"
class="es.codeurjc.test.ejem.GroupsTest"/>
      </group> <!-- resta -->
    </groups>
    ...
  </suite>
</testng-results>
```

- Introducción
- Test y Suites
- Parámetros
- Aserciones
- Dobles
- Paralelismo
- Interceptores
- Resultados
- **Conclusiones**

- Existen **frameworks** que facilitan la implementación de pruebas automáticas
- Son **muy similares** independientemente del **lenguaje de programación** usado
- Se diferencian tres partes en un test: **Given – When – Then** o **Arrange – Act - Assert**
- Las **aserciones** son una parte importante de las librerías porque hacen el **test más legible**
- Los **dobles** de las dependencias **facilitan** la creación de pruebas automáticas del SUT