JUnit 4

Pruebas Unitarias

Ejercicio Resuelto

Desarrollar un método estático que tome un array de enteros como argumento y devuelva el mayor valor encontrado en el array

Ejercicio Resuelto

- Partimos de esta versión:

```
public class Mayor {

    /** Devuelve el elemento de mayor valor de una lista ...9 lines */
    public static int obt_mayorNumero(int lista[]) {
        int indice, max = Integer.MAX_VALUE;
        for (indice = 0; indice < lista.length - 1; indice++) {
            if (lista[indice] > max) {
                max = lista[indice];
            }
        }
        return max;
    }
```

Ejercicio Resuelto

- ¿Qué pruebas se te ocurren para el método **obt_mayorNumero?**
  - ¿Qué ocurre para un *array con valores cualesquiera (el caso* normal)?
    - [3, 7, 9, 8] > 9
  - ¿Qué ocurre si el mayor elemento se encuentra al principio, en medio o al final del *array?*
    - [9, 7, 8] > 9; [7, 9, 8] > 9; [7, 8, 9] > 9
  - ¿Y si el mayor elemento se encuentra duplicado en el *array?*
    - [9, 7, 9, 8] > 9
  - ¿Y si sólo hay un elemento en el *array?*
    - [7] > 7
  - ¿Y si el *array está compuesto por elementos negativos?*
    - [-4, -6, -7, -22] > -4
  - ¿Y si el array es null?
    - Disparará una excepción

Ejercicio Resuelto

- Preparo las pruebas:

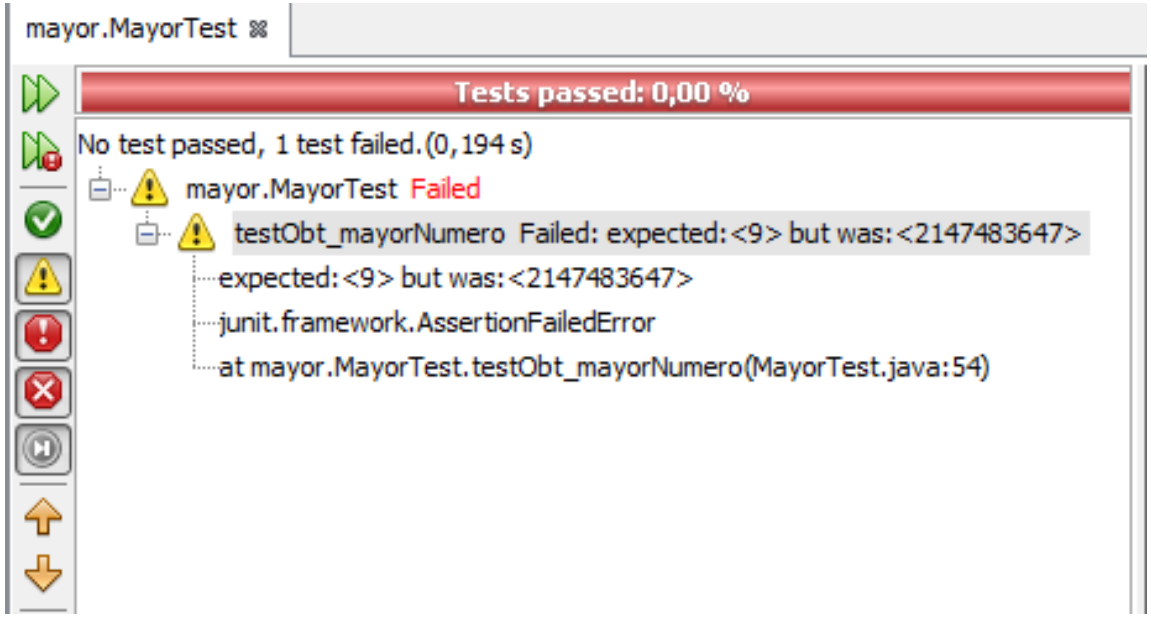
```
@Test
public void testObt_mayorNumero() {
    System.out.println("obt_mayorNumero");

    //si lista es null disparará una NullPointerExcepcion
    try {
        assertEquals(0, Mayor.obt_mayorNumero(null));
        fail("deberia haber lanzado una NullPointerException");
    } catch (NullPointerException e) {
    }

    assertEquals(9, Mayor.obt_mayorNumero(new int[]{3, 7, 9, 8}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{9, 7, 8}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{7, 9, 8}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{7, 8, 9}));
    assertEquals(9, Mayor.obt_mayorNumero(new int[]{9, 7, 9, 8}));
    assertEquals(7, Mayor.obt_mayorNumero(new int[]{7}));
    assertEquals(-4, Mayor.obt_mayorNumero(new int[]{-4, -6, -7, -22}));
}
```

Ejercicio Resuelto

- No las pasamos:



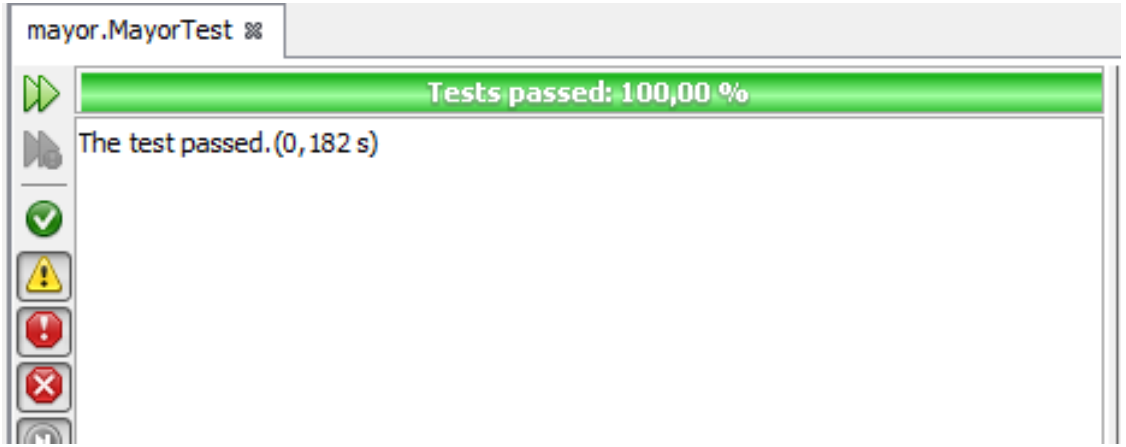
Ejercicio Resuelto

- Reviso el código:

```
public static int obt_mayorNumero(int lista[]) {
    //   int indice, max = Integer.MAX_VALUE;
    int indice, max = Integer.MIN_VALUE;
    // for (indice = 0; indice < lista.length - 1; indice++) {
    for (indice = 0; indice < lista.length; indice++) {
        if (lista[indice] > max) {
            max = lista[indice];
        }
    }
    return max;
}
```

Ejercicio Resuelto

- Ahora pasamos las pruebas:



Ejercicio Propuesto

- Aunque dentro de un método de prueba podemos poner tantos assert como queramos **es recomendable crear un método de prueba diferente por cada caso de prueba** que tengamos
  - Modifica el método testObt_mayorNumero() que contiene muchos casos de prueba creando varios métodos de prueba distintos: testObt_mayorNumero1(), testObt_mayorNumero2(), …
  - De esta forma cuando se presenten los resultados de las pruebas podremos ver exactamente qué caso de prueba es el que ha fallado

Anotaciones JUnit

Anotaciones JUnit

- En versiones anteriores de Junit no existían
  - Se han incluido en la versión 4
- Se trata de palabras clave que se colocan antes de los métodos de test e indican a las librerías Junit instrucciones concretas:
- @RunWith, @Before , @After , @Test

@Test

- La anotación @Test identifica el método que sigue como método de prueba o método test

```
@Test
public void method()
```

```
@Test  (expected  =  Exception.class)
```

Falla si el método no lanza la excepción esperada

```
@Test(timeout=100)
```

Falla si el método tarda más de 100 milisegundos

@Before

- @Before
  - Si anotamos a un método con esta etiqueta el código será ejecutado antes de cada método de prueba
  - Puede haber varios métodos con esta anotación
  - Se usa para preparar el entorno de test
    - Por ejemplo, leer datos de entrada, inicializar la clase, etc

@Before

- ¿Cómo podríamos utilizar @Before en Clase Calculadora?
  - Fíjate que en todos los métodos de test repetimos la instrucción de creación del objeto calcu

    calcu=new Calculadora()
  - Podríamos crear el método creaCalculadora() con la etiqueta @Before
- Creo una clase nueva CalculadoraTest2

```
import org.junit.Test;
import org.junit.After;
import org.junit.Before;

public class CalculadoraTest2 {

    private Calculadora calcu;
    private int resultado;

    @Before
    public void creaCalculadora(){
        calcu=new Calculadora(20,10);

    }

    @Test
    public void testSuma() {
         resultado= calcu.suma();
        assertEquals(30,resultado);
    }
```

@After

- @After
  - Si anotamos un método con esta etiqueta el código será ejecutado después de cada método de prueba
  - Podemos tener varios métodos con esta anotación
  - Se usa para limpiar el entorno de test
    - Por ejemplo borrar datos temporales, restaurar valores por defecto,…
    - Se puede usar también para ahorrar memoria limpiando estructuras de memoria pesadas

@After

- En la clase CalculadoraTest2 vamos a añadir un método que limpie los objetos creados y se ejecute después de las pruebas
- El método se llamará borraCalculadora() y se ejecutará al final de las pruebas de la clase Calculadoratest2

```
import org.junit.Test;
import org.junit.After;
import org.junit.Before;

public class CalculadoraTest2 {

    private Calculadora calcu;
    private int resultado;

    @Before
    public void creaCalculadora(){
        calcu=new Calculadora(20,10);

    }

    @After
    public void borraCalculadora(){
        calcu= null;

    }
```

@BeforeClass y @AfterClass

- @BeforeClass
  - Solo puede haber un método con esta etiqueta
  - El método marcado con esta anotación es invocado una vez antes de ejecutar todas las pruebas
  - Se usa para ejecutar actividades intensivas como conectar a una base de datos
- @AfterClass
  - Solo puede haber un método con esta anotación
  - Este método será invocado una sola vez cuando finalicen todas las pruebas
  - Se usa para actividades de limpieza, como por ejemplo, desconectar de la base de datos
- Los métodos marcados @BeforeClass y @AfterClass necesitan ser definidos como static

@BeforeClass y @AfterClass

- Los métodos anotados como @BeforeClass y @AfterClass deben ser **static** y por tanto los atributos a los que acceden también
- Creamos CalculadoraTest3 y añadimos métodos creaCalculadora y borraCalculadora con @BeforeClass y@AfterClass
- Ojo son de tipo static!!

```
import static org.junit.Assert.*;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

public class CalculadoraTest3 {

    private static Calculadora calcu;
    private int resultado;

    @BeforeClass
    public static void creaCalculadora(){
        calcu=new Calculadora(20,10);

    }

    @AfterClass
    public static void borraCalculadora(){
        calcu= null;

    }
```

@Ignore

- @Ignore Ignora el método de test
- Es útil cuando el código a probar ha cambiado y el caso de uso no ha sido todavía adaptado
- O si el tiempo de ejecución del método de test es demasiado largo para ser incluido