

# **M7 – UF2**

## **JUNIT**

### **Pruebas Unitarias**

**Eduard Lara**

# INDICE

1. Introducció
2. Un exemple senzill
3. El framework JUnit
4. El TestRunner
5. Tèrminos
6. Pruebas de Excepciones
7. Objetos Mock

# 1. INTRODUCCION

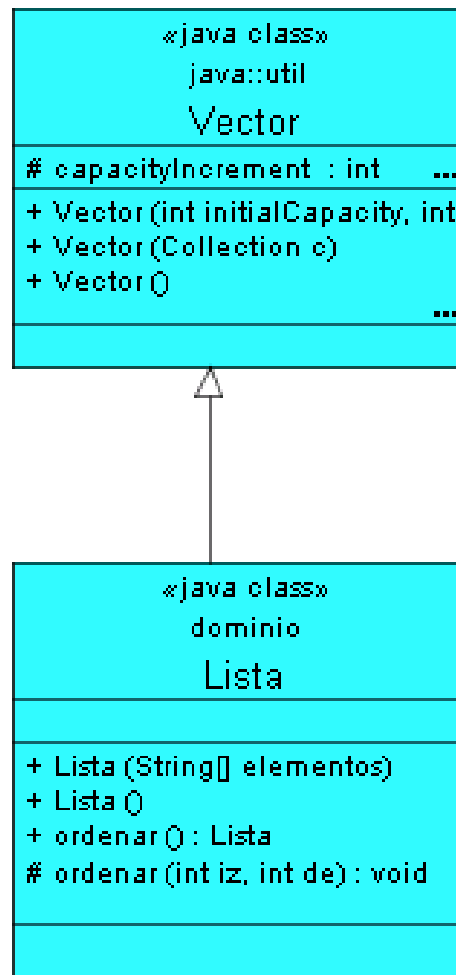
- JUnit es un “framework” para automatizar las pruebas de programas Java
- Escrito por Erich Gamma y Kent Beck
- Open Source, disponible en <http://www.junit.org>
- Adecuado para el Desarrollo dirigido por las pruebas (*Test-driven development*)
- Consta de un conjunto de clases que el programador puede utilizar para construir sus casos de prueba y ejecutarlos automáticamente
- Los casos de prueba son realmente programas Java. Quedan archivados y pueden ser reejecutados tantas veces como sea necesario

## 2. UN EJEMPLO SENCILLO

**Paso 1.** Representa una lista ordenable de forma creciente.

```
public class Lista extends Vector<String> {  
  
    public Lista(String [] elementos) {  
        for(String s:elementos)  
            this.add(s);  
    }  
    public Lista ordenar() {  
        for (int i=0; i<this.size()-1; i++) {  
            for (int j=i+1; j<this.size(); j++) {  
                if(get(i).compareTo(get(j))>1) {  
                    String aux = get(i);  
                    set(i,get(j));  
                    set(j, aux);  
                }  
            }  
        }  
        return this;  
    }  
}
```

## 2. UN EJEMPLO SENCILLO



Un posible caso de prueba es el siguiente:

```
String[] e3={"e", "d", "c", "b", "a"};
```

```
Lista reves=new Lista(e3);
```

```
Lista derecha=reves.ordenar();
```

...y el resultado esperado:

```
{"a", "b", "c", "d", "e"}
```

Si *derecha* es igual al resultado esperado, entonces el caso de prueba ha sido superado

## 2. UN EJEMPLO SENCILLO

**Paso 2.** Construyamos manualmente un objeto *expected* y comparémoslo con el obtenido:

```
public static void main(String args[]) {  
    String [] e3={"e", "d", "c", "b", "a"};  
    Lista revs = new Lista(e3);  
  
    Lista derecha = revs.ordenar();  
  
    String [] e2 = {"a", "b", "c", "d", "e"};  
    Lista expected=new Lista(e2);  
  
    if (derecha.equals(expected))  
        System.out.print("Resultado Correcto");  
    else  
        System.out.print("Resultado Incorrecto");  
}
```

# 3. FRAMEWORK JUNIT

El ejemplo anterior (*obtained* frente a *expected*) es una idea fundamental de JUnit

Ocurre que:

- JUnit nos va a permitir mantener de forma separada los casos de prueba
- JUnit permite ejecutarlos (y reejecutarlos) de forma automática
- Nos permite construir “árboles de casos de prueba” (*suites*)

### 3. FRAMEWORK JUNIT

Para el ejemplo anterior:

```
public void testOrdenarReves() {
```

```
    String[] ex={"a", "b", "c", "d", "e"};
```

```
    Lista expected=new Lista(ex);
```

Construcción manual del objeto  
esperado

```
    String[] e3={"e", "d", "c", "b", "a"};
```

```
    Lista listaAlReves=new Lista(e3);
```

Construcción manual del objeto obtenido haciendo uso de los métodos de la  
clase que estamos probando

```
    this.assertEquals(expected, listaAlReves.ordenar());
```

```
}
```

Comparación de ambos objetos haciendo uso de las funcionalidades  
suministradas por JUnit



## 3. FRAMEWORK JUNIT

Destaquemos algunos elementos:

```
public void testOrdenarReves() {  
    String[] ex={"a", "b", "c", "d", "e"};  
    Lista expected=new Lista(ex);  
  
    String[] e3={"e", "d", "c", "b", "a"};  
    Lista listaAlReves=new Lista(e3);  
  
    this.assertEquals(expected, listaAlReves.ordenar());  
}
```

No tiene método "assertEquals(...)"

Estamos probando la clase Lista

- Lista(String[])
- Lista()
- ordenar()
- ordenar(int, int)

```
java class  
dominio  
Lista  
  
+ Lista (String[] elementos)  
+ Lista ()  
+ ordenar () : Lista  
# ordenar (int iz, int de) : void
```

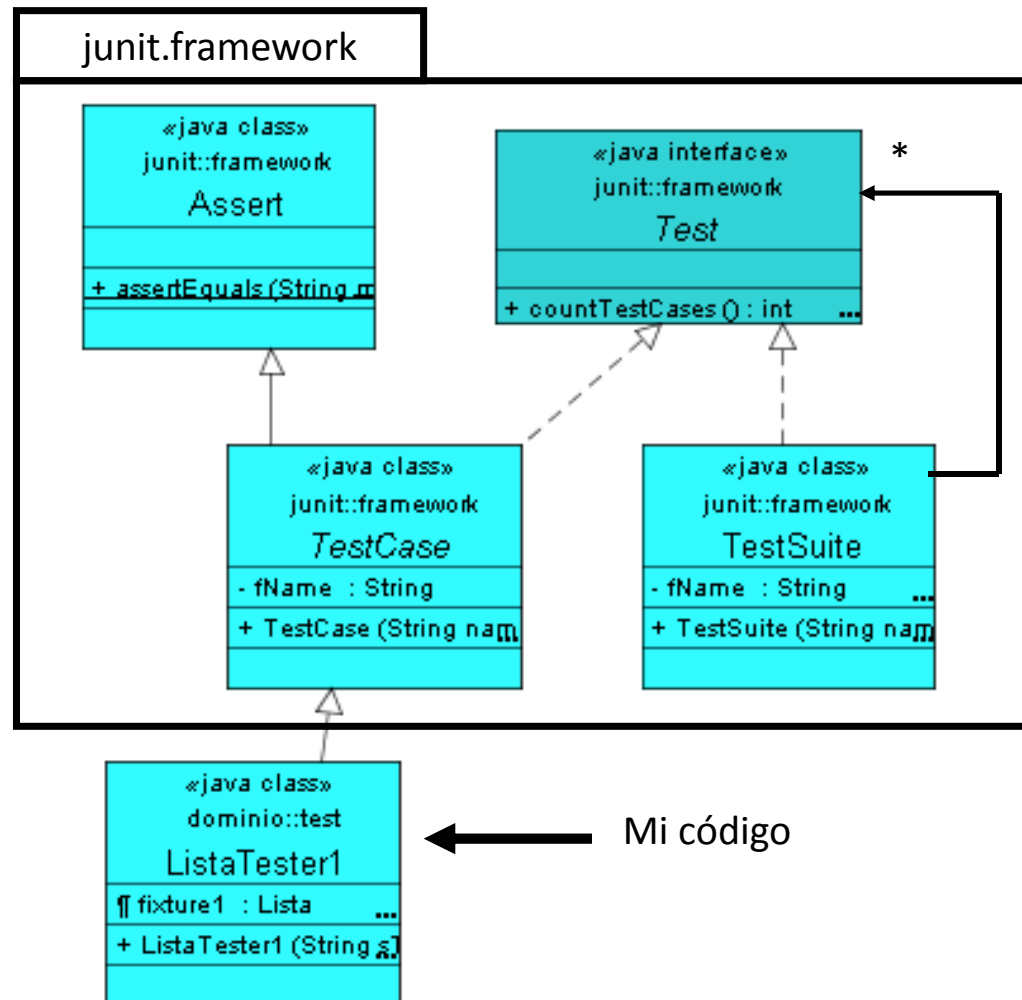
## 3. FRAMEWORK JUNIT

¿Dónde está el código anterior?

- En una clase *ListaTester*, creada ex profeso para realizar las pruebas de *Lista*
- *ListaTester* especializa a la clase *TestCase* definida en JUnit
- En *TestCase* está definido el método *assertEquals* antes mencionado, y muchos otros más

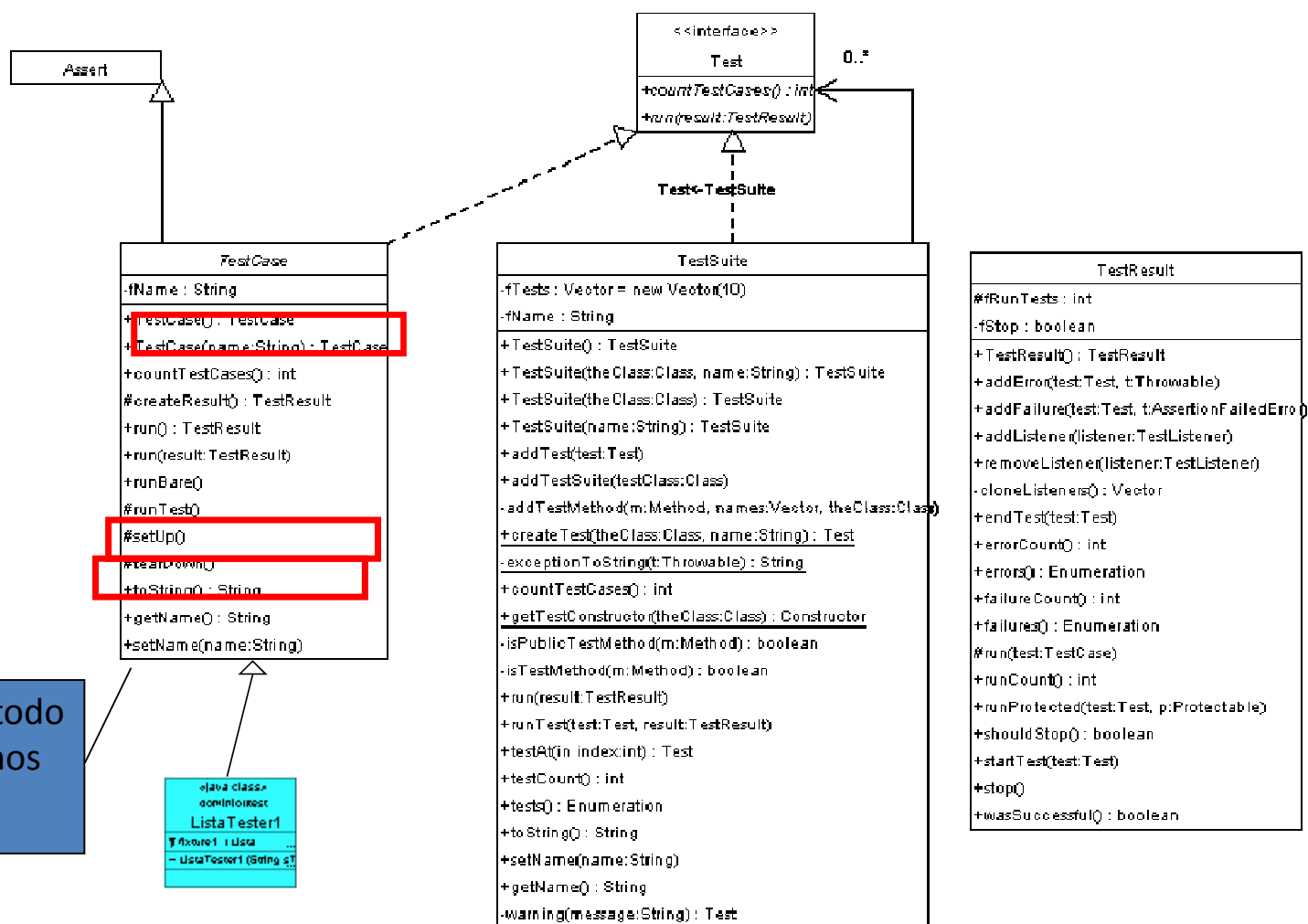
# 3. FRAMEWORK JUNIT

## Clases fundamentales



# 3. FRAMEWORK JUNIT

## Clases fundamentales



Ahí es donde utilizamos el método *assertEquals* que mencionamos antes

# 3. FRAMEWORK JUNIT

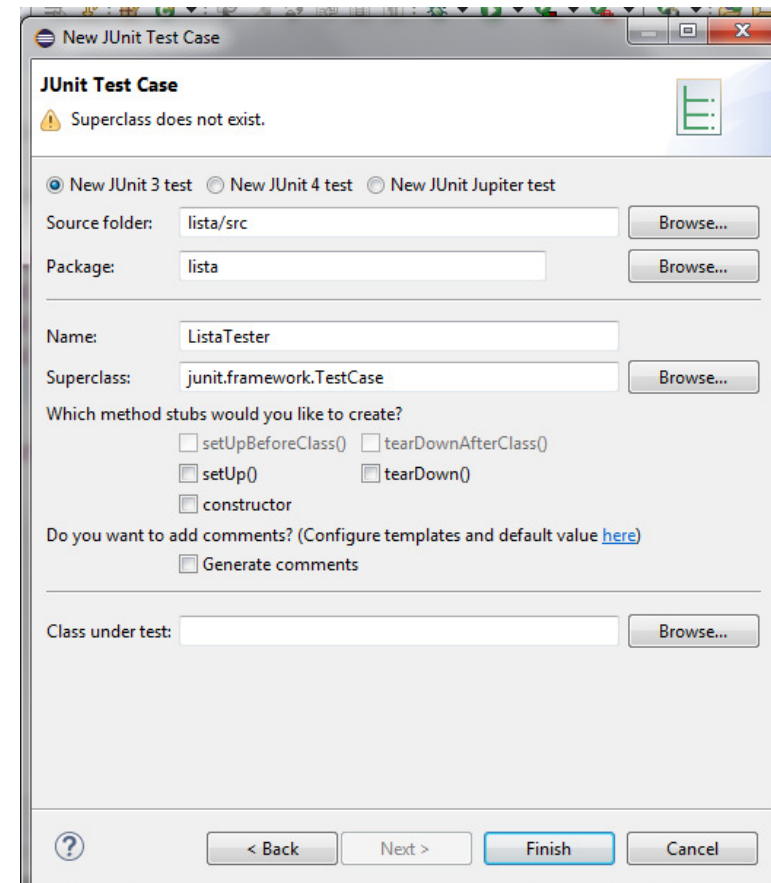
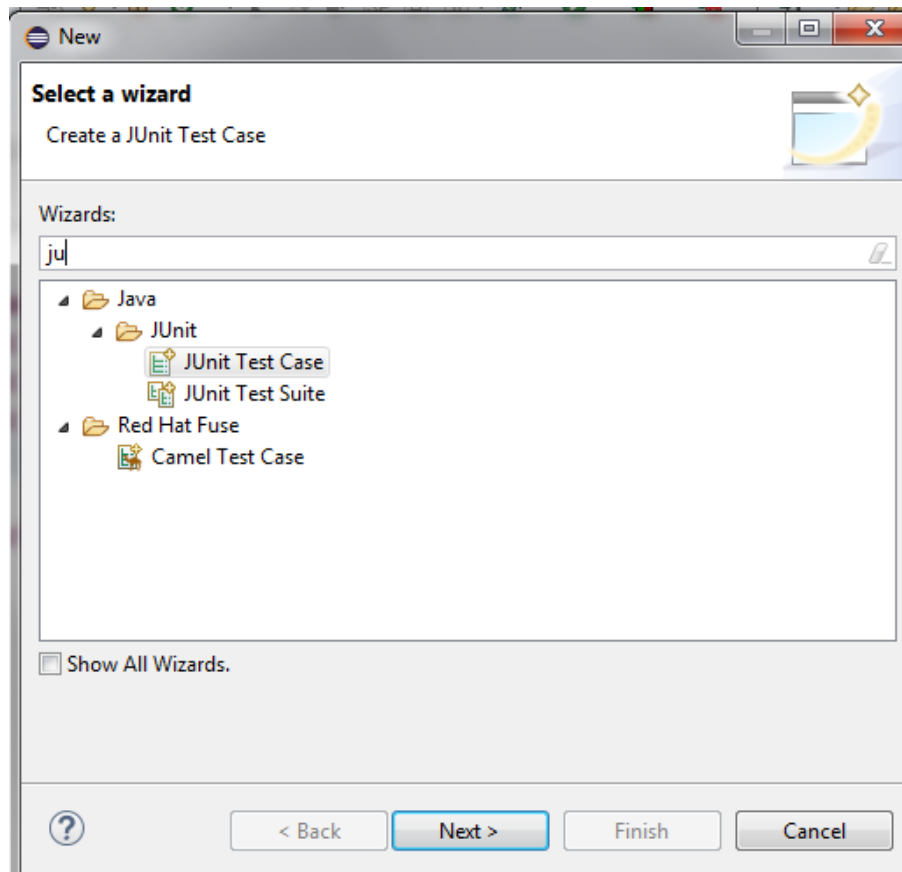
## Clases fundamentales Assert

Assert
<pre>#Assert() : Assert +assertTrue(message:String, in condition:boolean) +assertTrue(in condition:boolean) +assertFalse(message:String, in condition:boolean) +assertFalse(in condition:boolean) +fail(message:String) +fail() +assertEquals(message:String, expected:Object, actual:Object) +assertEquals(expected:Object, actual:Object) +assertEquals(message:String, expected:String, actual:String) +assertEquals(expected:String, actual:String) +assertEquals(message:String, in expected:double, in actual:double, in delta:double) +assertEquals(in expected:double, in actual:double, in delta:double) +assertEquals(message:String, in expected:float, in actual:float, in delta:float) +assertEquals(in expected:float, in actual:float, in delta:float) +assertEquals(message:String, in expected:long, in actual:long) +assertEquals(in expected:long, in actual:long) +assertEquals(message:String, in expected:boolean, in actual:boolean) +assertEquals(in expected:boolean, in actual:boolean) +assertEquals(message:String, in expected:byte, in actual:byte) +assertEquals(in expected:byte, in actual:byte) +assertEquals(message:String, in expected:char, in actual:char)</pre>

```
+assertEquals(in expected:char, in actual:char)
+assertEquals(message:String, expected:short, actual:short)
+assertEquals(expected:short, actual:short)
+assertEquals(message:String, in expected:int, in actual:int)
+assertEquals(in expected:int, in actual:int)
+assertNotNull(object:Object)
+assertNotNull(message:String, object:Object)
+assertNull(object:Object)
+assertNull(message:String, object:Object)
+assertSame(message:String, expected:Object, actual:Object)
+assertSame(expected:Object, actual:Object)
+assertNotSame(message:String, expected:Object, actual:Object)
+assertNotSame(expected:Object, actual:Object)
-failSame(message:String)
-failNotSame(message:String, expected:Object, actual:Object)
-failNotEquals(message:String, expected:Object, actual:Object)
~format(message:String, expected:Object, actual:Object) : String
```

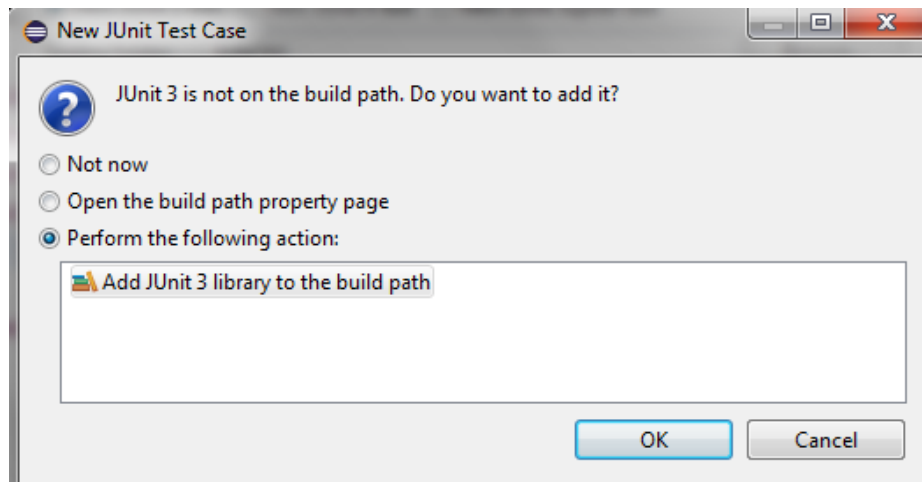
# 3. FRAMEWORK JUNIT

**Paso 3.** Crearemos la clase Test para testear la función ordenar:



# 3. FRAMEWORK JUNIT

**Paso 4.** Indicamos que utilizaremos la librería junit3:



```
ListaTester.java
1 package lista;
2
3 import junit.framework.TestCase;
4
5 public class ListaTester extends TestCase {
6
7 }
8
```

## 3. FRAMEWORK JUNIT

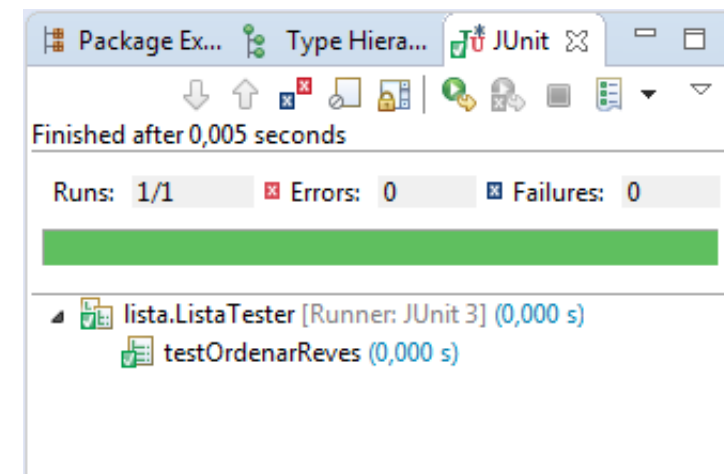
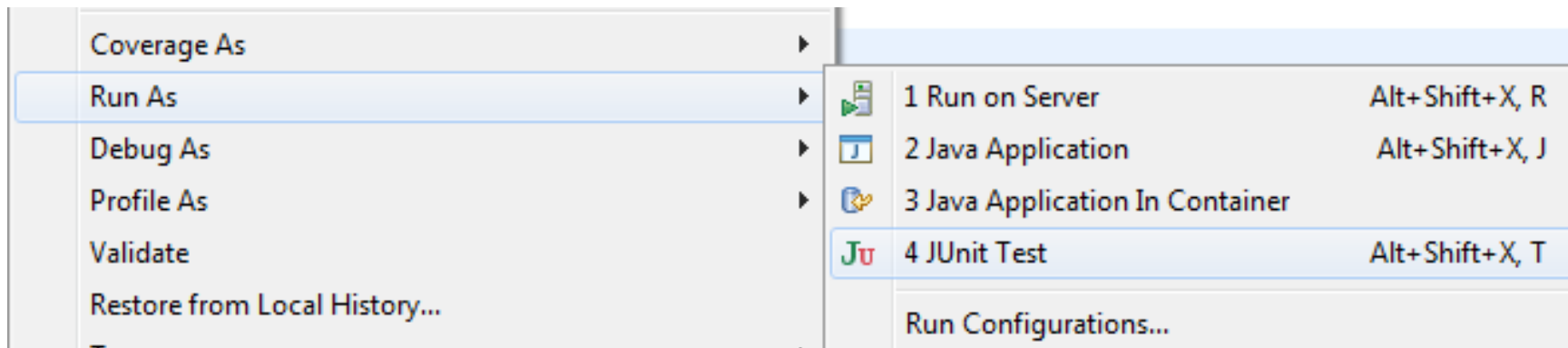
**Paso 5.** Completamos el código de testeo:

```
ListaTester.java ✖
1 package lista;
2
3 import junit.framework.TestCase;
4
5 public class ListaTester extends TestCase {
6     public ListaTester(String sTestName) {
7         super(sTestName);
8     }
9     public void testOrdenarReves() {
10         String[] ex={"a", "b", "c", "d", "e"};
11         Lista expected=new Lista(ex);
12
13         String[] e3={"e", "d", "c", "b", "a"};
14         Lista listaAlReves=new Lista(e3);
15
16         this.assertEquals(expected, listaAlReves.ordenar());
17     }
18 }
```



## 4. EL TESTRUNNER

**Paso 6.** Completamos el código de testeo:



## 4. EL TESTRUNNER

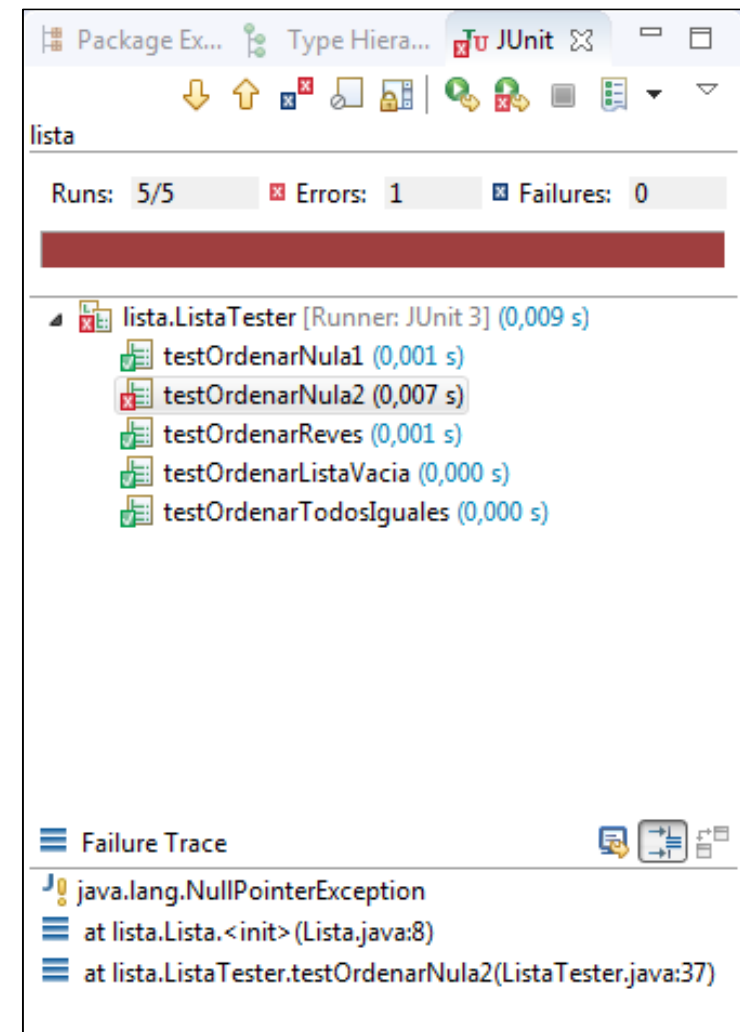
### Paso 7. Completamos el código de testeo:

```
public void testOrdenarTodosIguales() {  
    String[] e2={"a", "a", "a", "a", "a"};  
    Lista listaTodosIguales=new Lista(e2);  
  
    String[] ex={"a", "a", "a", "a", "a"};  
    Lista expected=new Lista(ex);  
  
    this.assertEquals(expected, listaTodosIguales.ordenar());  
}  
  
public void testOrdenarNula1() {  
    Lista listaNula1=null;  
    this.assertNull(listaNula1);  
}
```

```
public void testOrdenarNula2() {  
    String[] e4=null;  
    Lista listaNula2=new Lista(e4);  
    String[] ex=null;  
    Lista expected=new Lista(ex);  
    this.assertEquals(expected, listaNula2.ordenar());  
}  
  
public void testOrdenarListaVacía() {  
    String[] e5={};  
    Lista listaVacía=new Lista(e5);  
    String[] ex={};  
    Lista expected=new Lista(ex);  
    this.assertEquals(expected, listaVacía.ordenar());  
}
```

## 4. EL TESTRUNNER

**Paso 8.** Completamos el código de testeo:

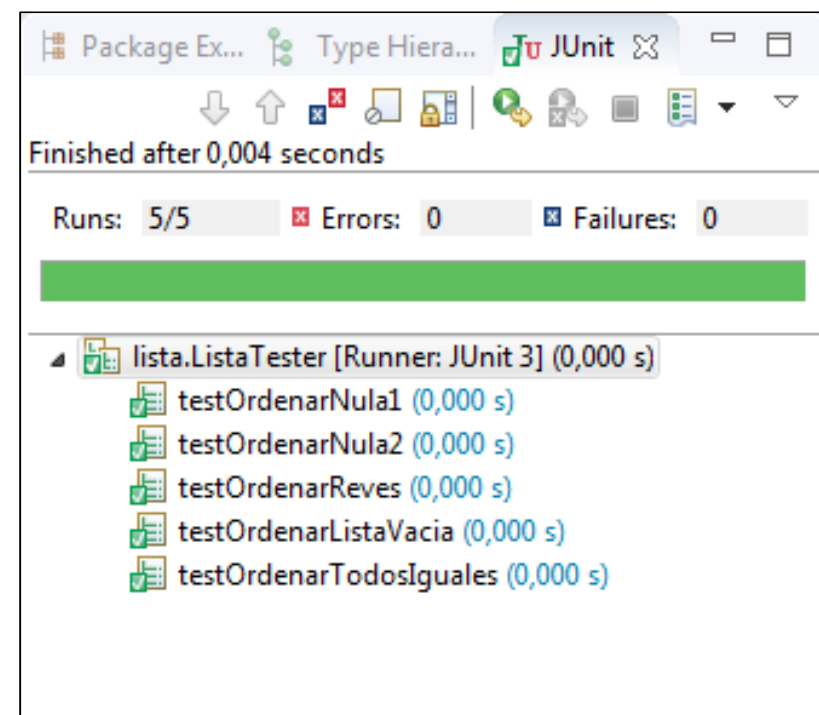


## 4. EL TESTRUNNER

**Paso 9.** Una vez que la clase *Lista* ha sido corregida...

```
public class Lista extends Vector<String> {

    public Lista(String [] elementos) {
        if (elementos!=null)
            for(String s:elementos)
                this.add(s);
    }
    public Lista ordenar() {
        for (int i=0; i<this.size()-1; i++) {
            for (int j=i+1; j<this.size(); j++) {
                if(get(i).compareTo(get(j))>=1) {
                    String aux = get(i);
                    set(i,get(j));
                    set(j, aux);
                }
            }
        }
        return this;
    }
}
```

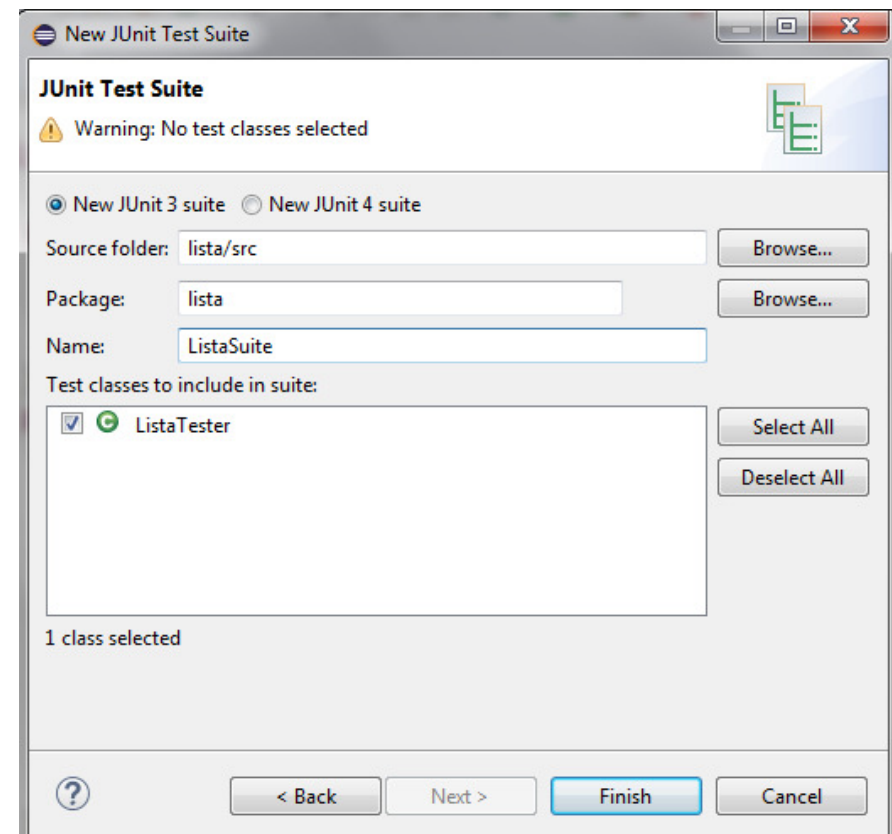
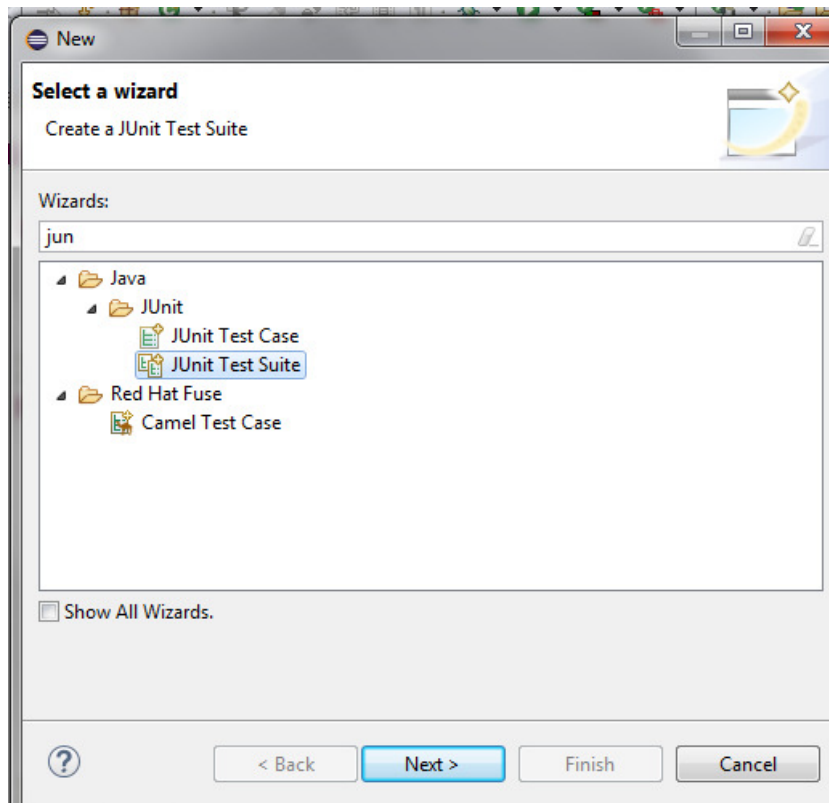


## 4. EL TESTRUNNER

- Es importante notar que todos los métodos *test* que vamos implementando se quedan guardados en *ListaTester*
- Si añadimos, borramos o modificamos el código de *Lista*, los casos de prueba habidos en *ListaTester* siguen disponibles y pueden volver a ser ejecutados
- Se aconseja reejecutarlos cada vez que se modifique el código

## 5. TERMINOS: TESTSUITE

**Paso 10.** En ocasiones es bueno agrupar casos de prueba: por ejemplo, tener un grupo de pruebas en el que ponemos las pruebas realizadas a listas vacías y nulas



## 5. TERMINOS: TESTSUITE

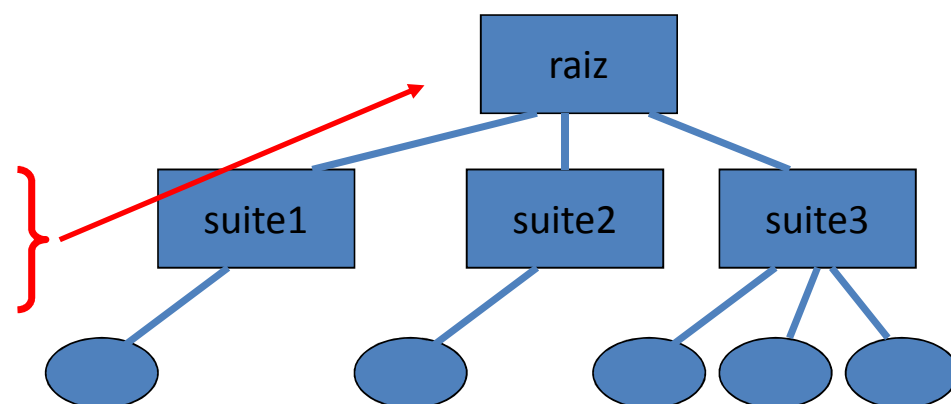
**Paso 11.** En ocasiones es bueno agrupar casos de prueba: por ejemplo, tener un grupo de pruebas en el que ponemos las pruebas realizadas a listas vacías y nulas

```
public class ListaSuite {

    public static Test suite() {
        TestSuite raiz=new TestSuite("raíz");
        TestSuite suite1=new TestSuite("Iguales");
        suite1.addTest(new ListaTester("testOrdenarTodosIguales"))}
        TestSuite suite2=new TestSuite("Al revés");
        suite2.addTest(new ListaTester("testOrdenarReves"));
        TestSuite suite3=new TestSuite("Nulas o vacías");
        suite3.addTest(new ListaTester("testOrdenarNula1"));
        suite3.addTest(new ListaTester("testOrdenarNula2"));
        suite3.addTest(new ListaTester("testOrdenarListaVacía"));
        raiz.addTest(suite1);
        raiz.addTest(suite2);
        raiz.addTest(suite3);
        return raiz;
    }
}
```

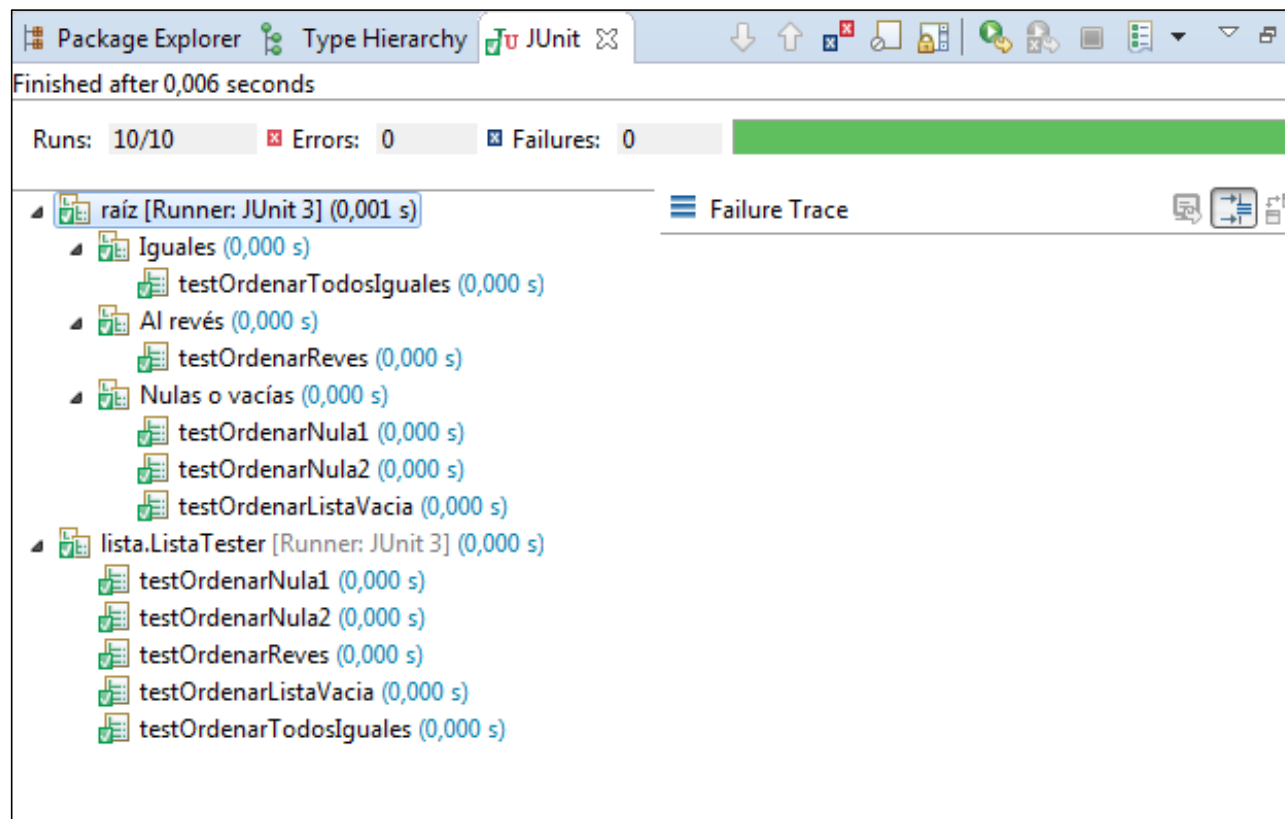
```
public class ListaSuite {

    public static Test suite() {
        TestSuite suite = new TestSuite(ListaSuite.class.getName());
        //$JUnit-BEGIN$
        suite.addTestSuite(ListaTester.class);
        //$JUnit-END$
        return suite;
    }
}
```



## 5. TERMINOS: TESTSUITE

**Paso 12.** Reejecutamos y nos ejecuta todo el suite de pruebas





## 5. TERMINOS

En muchos casos, los mismos objetos pueden ser utilizados para múltiples pruebas. Supongamos que añadimos a Lista un método *toString():String*

```
public String toString()  
{  
    String s="";  
    for (int i=0; i<size(); i++)  
        s+=" " + elementAt(i);  
    return s;  
}
```

- Nos interesará probar el `toString()` con la lista nula, la lista vacía, etc.

## 5. TERMINOS

```
public void testOrdenarReves() {  
    String[] ex={"a", "b", "c", "d", "e"};  
    Lista expected=new Lista(ex);  
    String[] e3={"e", "d", "c", "b", "a"};  
    Lista listaAlReves=new Lista(e3);  
    this.assertEquals(expected, listaAlReves.ordenar());  
}
```

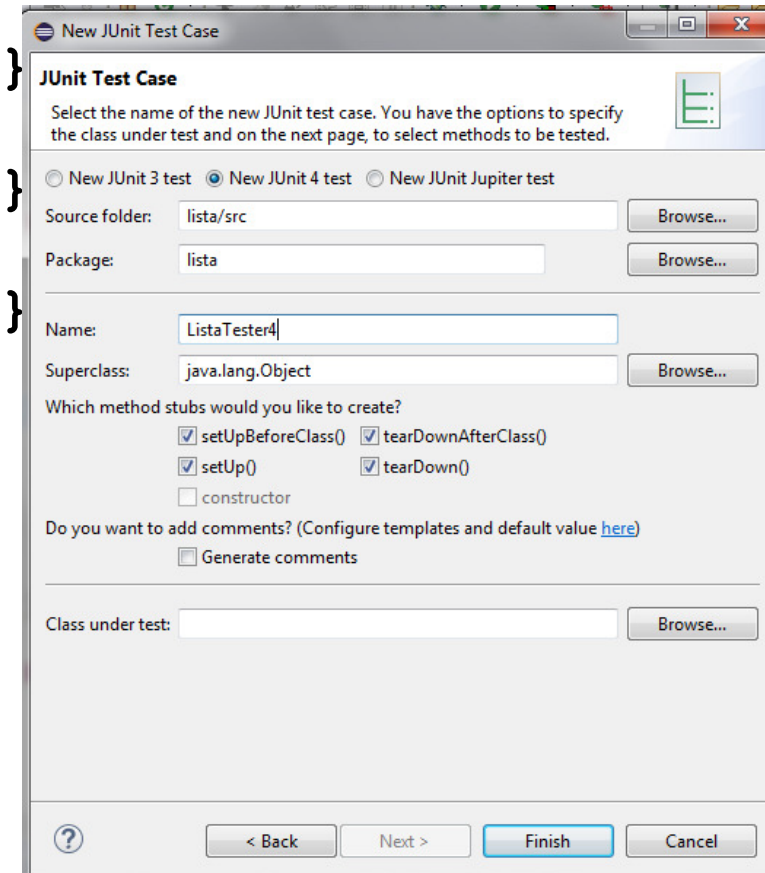
```
public void testToStringListaAlReves() {  
    String expected="a b c d e";  
    String[] e3={"e", "d", "c", "b", "a"};  
    Lista listaAlReves=new Lista(e3);  
    listaAlReves.ordenar();  
    this.assertEquals(expected, listaAlReves.ordenar());  
}
```

## 5. TERMINOS: FIXTURE

- En casos como el anterior creamos *fixtures* ( $\approx$  elementos fijos)
- Son variables de instancia de la clase de Test
- Se les asigna valor en el método *setUp()*, heredado de *TestCase*
- Se liberan en *tearDown()*
- *setUp* y *tearDown* se ejecutan antes y después de cada el *TestRunner* llame a cada método *test*

## 5. TERMINOS: FIXTURE

```
public void setUp() {  
    String[] e1={"a", "a", "a", "a", "a"};  
    listaTodosIguales=new Lista(e1);  
    String[] e2={"a", "b", "c", "d", "e"};  
    listaOrdenada=new Lista(e2);  
    String[] e3={"e", "d", "c", "b", "a"};  
    listaAlReves=new Lista(e3);  
    listaNula1=null;  
    String[] e4=null;  
    listaNula2=new Lista(e4);  
    String[] e5={};  
    listaVacua=new Lista(e5);  
}
```



## 6. PRUEBAS DE EXCEPCIONES (FAIL)

- Igual que es necesario comprobar cómo se comporta el programa en situaciones idóneas, es también importante probarlo en situaciones en que se producen errores.
- Es decir, que a veces el comportamiento correcto de nuestro programa consisten en se produzca un error

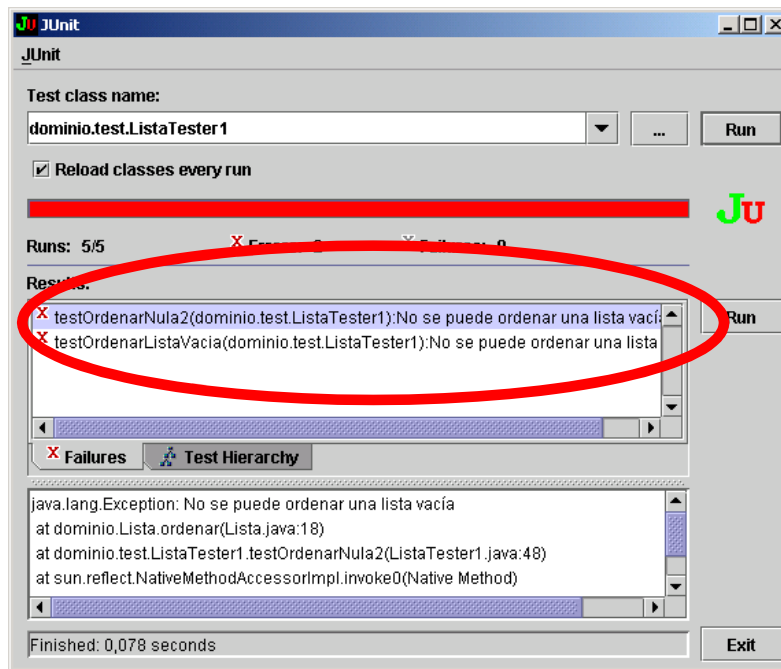
## 6. PRUEBAS DE EXCEPCIONES(FAIL)

→

Podemos desear que *ordenar()* dé un error cuando la lista esté vacía:

```
public Lista ordenar() throws Exception {  
    if (size()==0)  
        throw new Exception("No se puede ordenar una lista vacía");  
    ordenar(0, size()-1);  
    return this;  
}
```

## 6. PRUEBAS DE EXCEPCIONES (FAIL)



```
public void testOrdenarNula2()  
    throws Exception {  
    String[] ex=null;  
    Lista expected=new Lista(ex);  
    this.assertEquals(expected,  
        listaNula2.ordenar());  
}  
  
public void testOrdenarListaVacía()  
    throws Exception {  
    String[] ex={};  
    Lista expected=new Lista(ex);  
    this.assertEquals(expected,  
        listaVacía.ordenar());  
}
```

## 6. PRUEBAS DE EXCEPCIONES(FAIL)

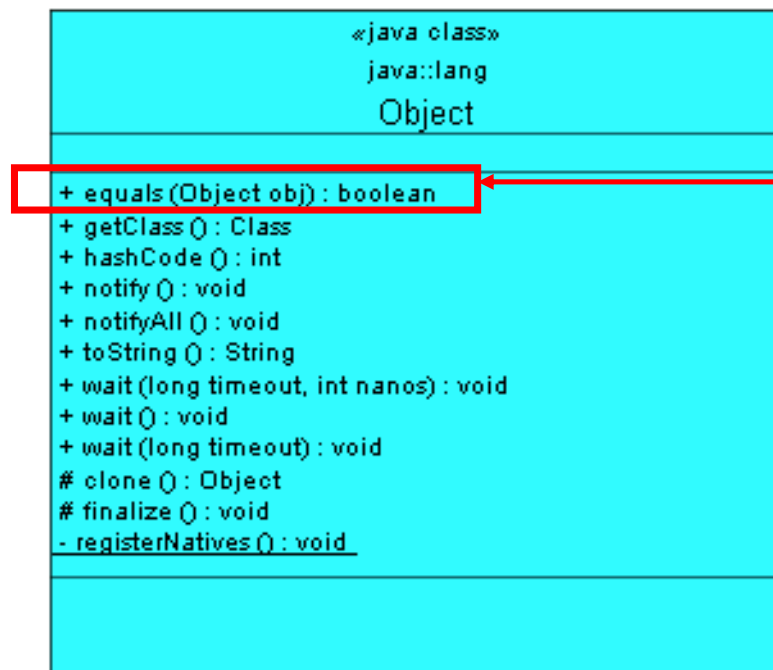
### Modificamos los dos métodos *test*

```
public void testOrdenarNula2() throws Exception {
    try
    {
        String[] ex=null;
        Lista expected=new Lista(ex);
        this.assertEquals(expected, listaNula2.ordenar());
        fail("Debería haberse lanzado una excepción");
    }
    catch (Exception e)
    {
        // Capturamos la excepción para que el caso no falle
    }
}
```



# Redefinición del método *equals*

Todas las clases Java son especializaciones de *Object* →

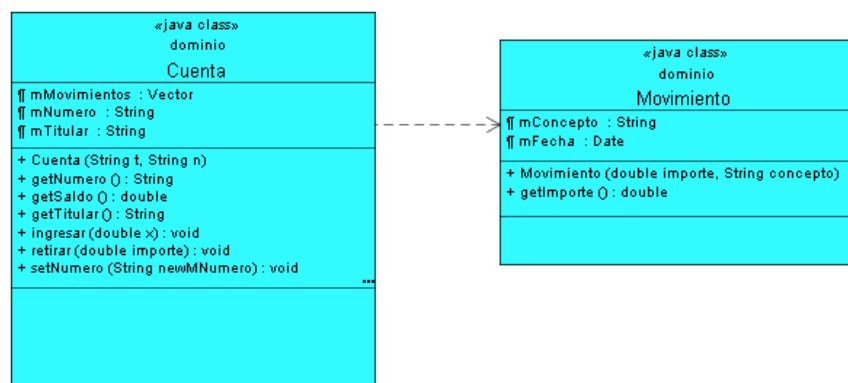


Llamado por los  
*assertEquals(...)* definidos  
en *Assert*

# Redefinición del método *equals*

Por tanto, en muchos casos tendremos que redefinir *equals(Object):boolean* en la clase que estamos probando

# Ejemplo “equals” (I)



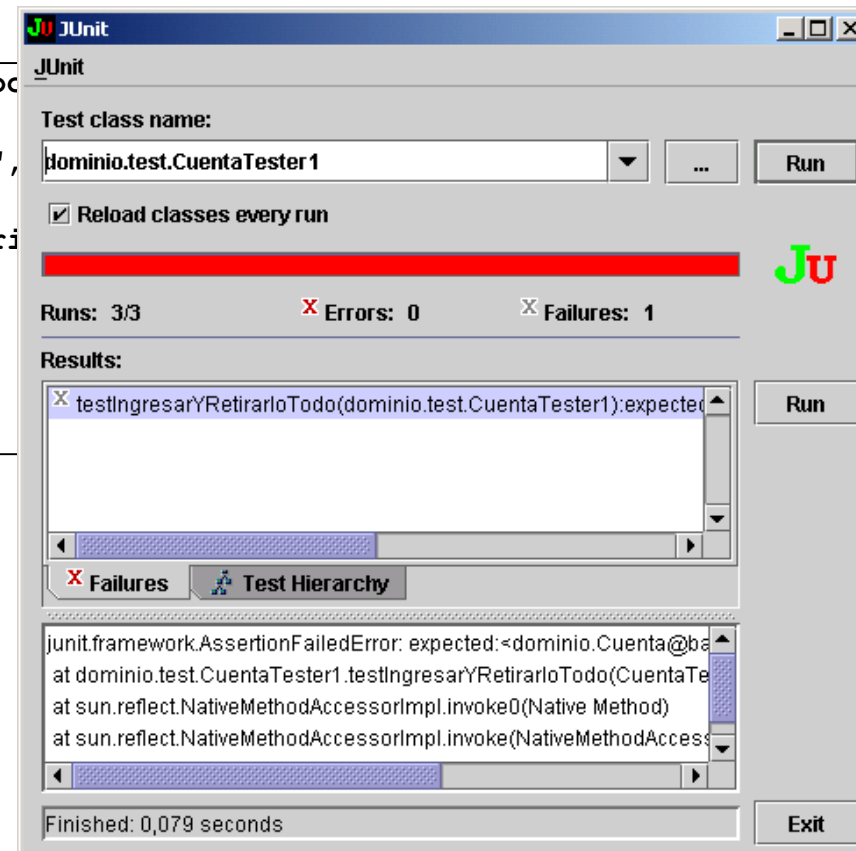
¿Cuándo son dos cuentas son iguales?

- a) Los saldos son los mismos
- b) Tienen el mismo nº de movimientos
- c) Opción b y todos son iguales
- d) ...

# Ejemplo “equals” (II)

```
public void testIngresarYRetirarloTodo()
{
    Cuenta expected=new Cuenta("Pepe",
    Cuenta obtained=new Cuenta("Macari
    obtained.ingresar(1000.0);
    obtained.retirar(1000.0);

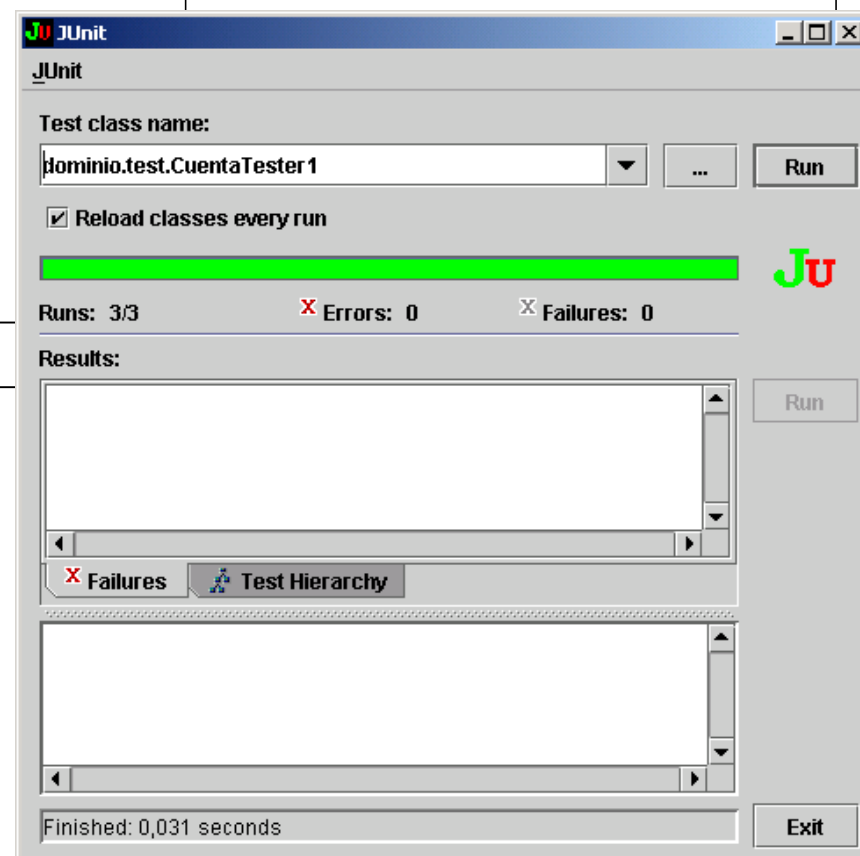
    assertEquals(expected, obtained);
}
```



# Ejemplo “equals” (y III)

Si redefinimos *equals(Object): boolean* en Cuenta de ese modo...

```
public void testIngresarYRetirarloTodo() throws
{
    Cuenta expected=new Cuenta("Pepe", 123);
    public boolean equals(Object o){
        if (!Cuenta.class.isInstance(o))
            return false;
        Cuenta obtained=new Cuenta("Macario", "123456");
        Cuenta c=(Cuenta) o;
        obtained.ingresar(1000);
        obtained.retirar(1000);
        return getSaldo()==c.getSaldo();
    }
    assertEquals(expected, obtained);
}
```



## **assertTrue(boolean)**

```
public void testIngresar()
{
    Cuenta obtained=new Cuenta("Pepe", "123");
    obtained.ingresar(100.0); obtained.ingresar(200.0);
    obtained.ingresar(300.0);
    assertTrue(obtained.getSaldo()==600.0);
}
```

## **assertNull(Object)**

```
public void testNull()
{
    Cuenta c=null;
    assertNull(c);
}
```

# Otros métodos *assert*X

```
assertSame(Object, Object)/assertNotSame(Object, Object)
    public void testDiferentesReferencias() throws Exception
    {
        Cuenta cuenta1=new Cuenta("Macario", "123456");
        cuenta1.ingresar(1000.0);
        cuenta1.retirar(1000.0);

        Cuenta cuenta2=new Cuenta("Macario", "123456");
        cuenta2.ingresar(1000.0);
        cuenta2.retirar(1000.0);

        assertEquals(cuenta1, cuenta2);
        assertNotSame(cuenta1, cuenta2);
    }
```

# Clases de prueba abstractas

**Se pueden posponer las pruebas hasta que se tengan especializaciones concretas de la clase abstracta**

**Pero también puede construirse una clase de Test abstracta**



# Clases de prueba abstractas

```
public abstract class TarjetaTester1 extends TestCase
{
    public TarjetaTester1(String sTestName)
    {
        super(sTestName);
    }

    public abstract Tarjeta getTarjeta();
    public abstract Tarjeta prepararTarjetaEsperada();

    public void testRetirar()
    {
        Tarjeta obtained=getTarjeta();
        obtained.retirar(100.0);

        Tarjeta expected=prepararTarjetaEsperada();
        assertEquals(expected, obtained);
    }
}
```

## 7. OBJETOS MOCK

Basados en JUnit

Sustituyen a clases complejas, dispositivos, etc.

Ejemplos: servlets, páginas jsp, bases de datos...

# 7. OBJETOS MOCK

```
public class temperature extends HttpServlet
{
    private static final String CONTENT_TYPE = "text/html";

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException
    {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        String str f=request.getParameter("Fahrenheit");

        try {
            int temp_f=Integer.parseInt(str f);
            double temp_c=(temp_f-32)*5/9.0;
            out.println("Fahrenheit: " + temp_f + ", Celsius: " + temp_c);
        }
        catch (NumberFormatException e) {
            out.println("Invalid temperature: " + str f);
        }
    }
}
```

# 7. OBJETOS MOCK

```
import com.mockobjects.servlet.*;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class TemperatureTester extends TestCase
{
    public TemperatureTester()
    {
    }

    public void test_bad_parameter() throws Exception {
        temperature s = new temperature();
        MockHttpServletRequest request=new MockHttpServletRequest();
        MockHttpServletResponse response=new MockHttpServletResponse();
        request.setupAddParameter("Fahrenheit", "boo!");
        response.setExpectedContentType("text/html");
        s.doGet(request, response);
        response.verify();
        assertTrue(response.getOutputStreamContents().startsWith("Invalid temperature"));
    }
    ...
}
```

El *MockHttpServletRequest* y el *MockHttpServletResponse* son objetos *HttpServletRequest* y *HttpServletResponse*, ya que el servlet que estamos probando trabaja con objetos de estos tipos

# 7. OBJETOS MOCK



# 7. OBJETOS MOCK

```
...
request.setupAddParameter("Fahrenheit", "boo!");
response.setExpectedContentType("text/html");
s.doGet(request, response);
response.verify();
...
```

Operaciones específicas para probar

```
«java class»
com::mockobjects::servlet
MockHttpServletRequest

- myParameters : Dictionary
+ MockHttpServletRequest(): void
+ addActualParameter(String param0, String param1): void
+ addActualParameter(String param0, String[] param1): void
+ getAttribute(String param0): Object
+ getAttributeNames(): Enumeration
+ getAuthType(): String
+ getCharacterEncoding(): String
+ getRemoteAddr(): String
+ getRemoteHost(): String
+ getRemoteUser(): String
+ getRequestDispatcher(String param0): RequestDispatcher
+ getRequestURI(): String
+ getRequestedSessionId(): String
+ getScheme(): String
+ getServerName(): String
+ getServerPort(): int
+ getServletPath(): String
+ getSession(boolean param0): HttpSession
+ getSession(): HttpSession
+ getUserPrincipal(): Principal
+ isRequestedSessionIdFromCookie(): boolean
+ isRequestedSessionIdFromURL(): boolean
+ isRequestedSessionIdFromUrl(): boolean
+ isRequestedSessionIdValid(): boolean
+ isSecure(): boolean
+ isUserInRole(String param0): boolean
+ removeAttribute(String param0): void
+ setAttribute(String param0, Object param1): void
+ setNoActualParameters(): void
+ setupAddParameter(String param0, String param1): void
+ setupAddParameter(String param0, String[] param1): void
+ setupNoParameters(): void
+ setupPathInfo(String param0): void
```

## 7. OBJETOS MOCK

**De forma general, todos los objetos Mock comparten la misma estructura:**

- Especializan a la clase que se usa realmente (implementan por tanto todas sus posibles operaciones abstractas)
- Contienen un conjunto de operaciones adicionales *addExpected...* o *setupExpected...*, que van indicando al objeto el estado en que quedará tras ejecutar la operación de “dominio”
- Pueden implementar la interfaz *Verifiable* (método *verify()*)

## 7. OBJETOS MOCK

**Difíciles de usar (poca documentación)**

**Descargas y más información en [www.mockobjects.com](http://www.mockobjects.com)**



# CONCLUSIONES

- Marco de pruebas semiautomático
- Automatiza las pruebas de regresión
- Los casos de prueba documentan el propio código fuente
- Adecuado para Desarrollo dirigido por las pruebas
- Extensible (p.ej.: Mock), abierto, gratuito