

# Proves unitàries

## Introducció

Una prova unitària és una manera de provar el funcionament d'un mètode particular. Els mètodes es proven per separat. L'objectiu de les proves unitàries és aïllar cada part del programa i mostrar que el seu funcionament és correcte.

La idea és escriure casos de prova no trivials per a cada mètode i provar-los cada vegada que fem modificacions.

S'han desenvolupat eines que faciliten l'elaboració de proves unitàries en diferents llenguatges, **JUnit** o **TestNG** s'utilitza per a realitzar proves unitàries a Java.


Els conceptes fonamentals en aquestes eines són el **cas de prova** (test case) i la **suite de prova** (test suite).

Els casos de prova són classes o mòduls que disposen de mètodes per a provar els mètodes d'una classe o mòdul concret. Així, per a cada classe que vulguem provar definim la seua corresponent classe de cas de prova.

Mitjançant les suites podem organitzar els casos de prova, de manera que cada suite agrupa els casos de prova de mòduls que estan funcionalment relacionats.

Els casos de prova ens permeten automatitzar la realització de proves, s'escriuen una vegada i s'executen les vegades necessàries.

Veurem com es crea una unitat de prova mitjançant JUnit o TestNG amb NetBeans. Amb aqueixa unitat de prova provarem els mètodes d'una classe (proves unitàries). Les unitats de prova mantenen els valors de prova, no és necessari tornar a escriure-les i són fàcils d'executar.

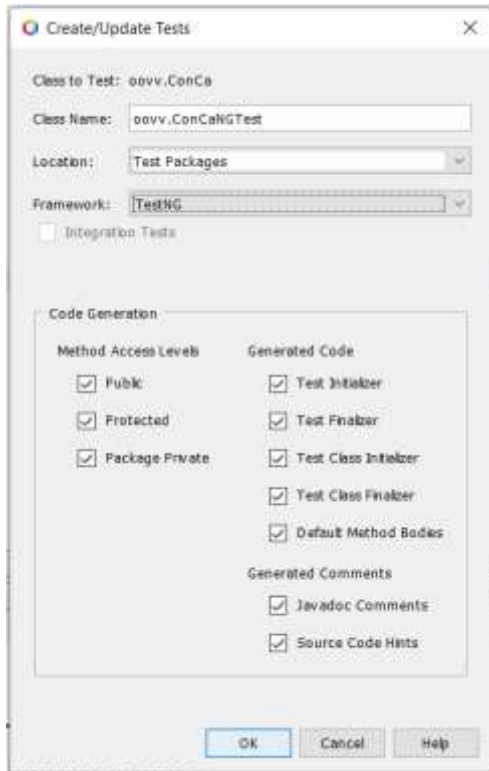
 El fet de que la implementació supere tots els casos de prova no vol dir que siga correcta, només vol dir que funciona correctament per als casos de prova que hem dissenyat

Un **Clean & Build** del projecte no inclou els paquets de prova en el fitxer .jar executable.



## Crear una classe de proves unitàries

Podem crear una classe de proves unitàries per a una classe fent clic en [Tools > Create/Update Test](#), també, és accessible des del menú contextual de la classe.



En el combobox [Framework](#) triem el framework per a realitza la classe de prova: [JUnit](#), [TestNG](#) o [JUnit4](#), i es defineix el paquet on es crearà la classe de prova.

**⚠ JUnit carrega les llibreries de JUnit 5.x, que no funcionen correctament amb projectes Ant, si estem usant un projecte Java amb Ant cal utilitzar JUnit4 (és el nostre cas).**

L'única diferencia entre JUnit 4.x i JUnit 5.x són les anotacions.

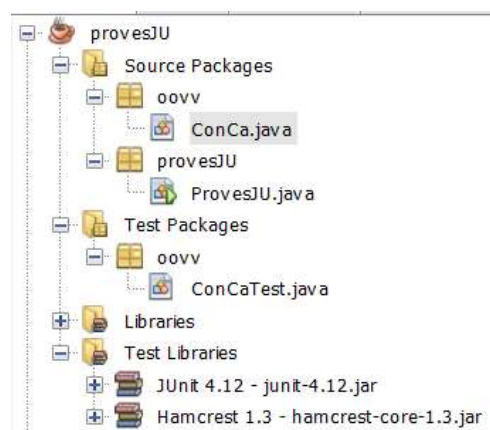
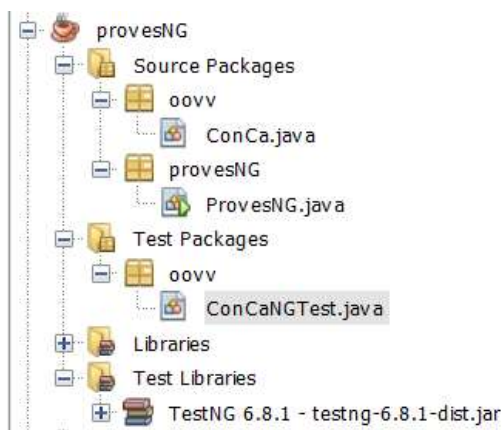
En la finestra tenim caselles per a seleccionar, quins mètodes es van provar, el codi i comentaris que generarà.

En la figura de la dreta s'usa el framework [TestNG](#) per a la classe `oovv.ConCa`, es crea la classe `oovv.ConCaNGTest.java` que es guarda en el paquet

Test Packages, afectarà els mètodes públics, protegits i els de paquet, generarà el codi d'inicialització i finalització del test i el cos de cada mètode, i generarà els comentaris, marca les opcions que vols.

Quan fem clic en [\[OK\]](#) es crea la unitat de prova en el paquet de proves. Si no està creat, es crea el paquet de prova i la llibreria de proves.

Es crea un mètode de prova per a cada mètode de la classe.



Si s'elegeix [JUnit4](#) les diferències són mínimes, en el codi generat canvien els noms de les anotacions.

## La classe de prova

La classe de l'exemple anterior és

```
public class ConCa {
    public String concatena(String uno, String dos){
        return uno + dos;
    }
}
```

La classe de prova que es crea per al framework [JUnit4](#) és

```
public class ConCaTest {
    public ConCaTest() {
    }
    @BeforeClass
    public static void setUpClass() {
    }
    @AfterClass
    public static void tearDownClass() {
    }
    @Before
    public void setUp() {
    }
    @After
    public void tearDown() {
    }
    /**
     * Test of concatena method, of class ConCa.
     */
    @Test
    public void testConcatena() {
        System.out.println("concatena");
        String uno = "";
        String dos = "";
        ConCa instance = new ConCa();
        String expectedResult = "";
        String result = instance.concatena(uno, dos);
        assertEquals(expectedResult, result);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }
}
```

La classe de prova que es crea per al framework [TestNG](#) és

```
public class ConCaNGTest {
    public ConCaNGTest() {
    }
}
```



```

@BeforeClass
public static void setUpClass() throws Exception {
}
@AfterClass
public static void tearDownClass() throws Exception {
}
@BeforeMethod
public void setUpMethod() throws Exception {
}
@AfterMethod
public void tearDownMethod() throws Exception {
}
/**
 * Test of concatena method, of class ConCa.
 */
@Test
public void testConcatena() {
    System.out.println("concatena");
    String uno = "";
    String dos = "";
    ConCa instance = new ConCa();
    String expResult = "";
    String result = instance.concatena(uno, dos);
    assertEquals(result, expResult);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
}

```

Canvien les anotacions

JUnit 4.x	JUnit 5.x	TestNG
@Before	@BeforeAll	@BeforeClass
@After	@AfterAll	@AfterClass
@BeforeClass	@BeforeEach	@BeforeMethod
@AfterClass	@AfterEach	@AfterMethod

Però el mètode de prova és el mateix

```

@Test
public void testConcatena() {
    System.out.println("concatena");
    String uno = "";
    String dos = "";
    ConCa instance = new ConCa();
    String expResult = "";
    String result = instance.concatena(uno, dos);
    assertEquals(result, expResult);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}

```



⚠ la prova falla, cal llevar el fail del mètode de prova i escriure el codi de la prova a realitzar usant assercions

Una asserció és una comparació del valor que esperem amb el resultat que realment s'obté, si no coincideixen, llavors, l'asserció falla.

Per defecte, es crea un mètode de prova per a cada mètode de la classe, depén de les caselles que s'han marcat.

Encara que, només tinguem un únic mètode, podem crear tots els mètodes de prova que vulguem.

El test quedaria com segueix

```
public void testConcatena() {
    System.out.println("concatena");
    String uno = "hola";
    String dos = "mundo";
    ConCa instance = new ConCa();
    String expectedResult = "holamundo";
    String result = instance.concatena(uno, dos);
    assertEquals(expectedResult, result);
}
```

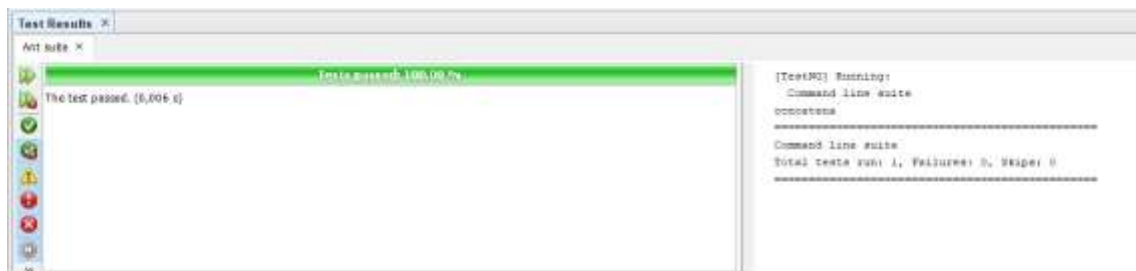
es pot escurçar, i deixar-ho com segueix

```
public void testConcatena() {
    System.out.println("concatena");
    ConCa instance = new ConCa();
    String result = instance.concatena("hola", "mundo");
    assertEquals("holamundo", result);
}
```

## Executar els tests

Per a executar els tests, obrim el menú contextual de la classe de proves i triem [Test File](#).

S'obri la finestra [Test Results](#) on es visualitza la realització dels diferents tests de la classe. Si tot va bé s'indica que el 100% de les proves han sigut correcta, tot verda.



Les fallades que s'han produïts s'indiquen en roig, (s'ha ja que el resultat és “hola/món”)





## Les anotacions

JUnit 4.x	JUnit 5.x	mètode
@Test	@Test	És el mètode de prova a realitzar
@BeforeClass	@BeforeAll	setUpClass s'executa una sola vegada abans de l'execució de les proves
@AfterClass	@AfterAll	tearDownClass s'executa una sola vegada abans de l'execució de les proves
@Before	@BeforeEach	setUp s'executa abans de cada mètode de prova
@After	@AfterEach	tearDown s'executa després de cada mètode de prova
@Ignore	@Ignore	Ignora el mètode de prova

TestNG	mètode
@Test	És el mètode de prova a realitzar
@BeforeClass	setUpClass s'executa una sola vegada abans de l'execució de les proves
@AfterClass	tearDownClass s'executa una sola vegada abans de l'execució de les proves
@BeforeMethod	setUpMethod s'executa abans de cada mètode de prova
@AfterMethod	tearDownMethod s'executa després de cada mètode de prova
@Ignore	Ignora el mètode de prova

Només s'executen els test que tinguen l'anotació @Test davant del codi del mètode de prova, si es lleva aquest mètode de prova no s'executa. També, es pot escriure l'anotació @Ignore davant de l'anotació @Test.

## Les assercions

Una asserció compara el valor esperat amb l'obtingut. Si l'asserció no és correcta es llança un AssertionError. El missatge escrit entre [ ] és optatiu, si es posa el missatge s'afegirà al AssertionError. Veurem els formats més habituals.

Un mètode de prova pot contenir més d'una asserció, però si es llança un AssertionError, llavors es finalitza el mètode i no es comproven les assercions posteriors.

El millor, és crear un mètode de prova per a cadascuna de les proves que volem realitzar, amb una única asserció.



## JUnit

Totes les assercions de JUnit les tens en

<http://junit.sourceforge.net/javadoc/org/junit/assert.html>

- `assertEquals([missatge], esperat, actual)` És correcta si el valor actual és igual al valor esperat, hi ha versions per a tipus de dades diferents
- `assertEquals([missatge], esperat, actual, tolerància)` És correcta si el valor actual és igual al valor esperat, és un format per a comprovar valors de tipus double, la tolerància indica el número de decimal a tindre en compte en la comparació
- `assertArrayEquals([missatge], esperada, actual)` És correcta si la matriu actual és igual a la matriu esperada
- `assertTrue([missatge], condició)` És correcta si la condició és vertadera
- `assertFalse([missatge], condició)` És correcta si la condició és falsa
- `assertNull([missatge], objecte)` És correcta si objecte és null
- `assertNotNull([missatge], objecte)` És correcta si objecte no és null
- `assertSame([missatge], esperat, actual)` És correcta si les referències apunten al mateix objecte
- `assertNotSame([missatge], esperat, actual)` És correcta si les referències no apunten al mateix objecte
- `fail(missatge)` Llança un `AssertionError`, això finalitza el mètode

## TestNG

Totes les assercions de TestNG les tens en

<https://www.javadoc.io/doc/org.testng/testng/6.8.17/org/testng/assert.html>

- `assertEquals(actual, esperat, [missatge])` És correcta si el valor actual és igual al valor esperat, hi ha versions per a tipus de dades diferents, també, inclou matrius i col·leccions
- `assertEquals(actual, esperat, tolerància, [missatge])` És correcta si el valor actual és igual al valor esperat, és un format per a comprovar valors de tipus double, la tolerància indica el número de decimal a tindre en compte en la comparació
- `assertEqualsNoOrder(actual, esperada, [missatge])` És correcta si les matrius actual i esperada contenen els mateixos elements sense importar l'ordre
- `assertTrue(condició, [missatge])` És correcta si la condició és vertadera
- `assertFalse(condició, [missatge])` És correcta si la condició és falsa
- `assertNull(objecte, [missatge])` És correcta si objecte és null



- `assertNotNull(objecte, [missatge])` És correcta si objecte no és null
- `assertSame(esperat, actual, [missatge])` És correcta si les referències apunten al mateix objecte
- `assertNotSame(esperat, actual, [missatge])` És correcta si les referències no apunten al mateix objecte
- `fail(missatge)` Llança un `AssertionError`, això finalitza el mètode

Com es pot apreciar els canvis d'un framework a un altre so mínims.

## Exemple

Aquesta és la classe `Calculador` que realitza operacions bàsiques.

```
public class Calculador {

    public double suma(double numero1, double numero2) {
        return numero1 + numero2;
    }

    public double resta(double numero1, double numero2) {
        return numero1 - numero2;
    }

    public double multiplica(double numero1, double numero2) {
        return numero1 * numero2;
    }

    public double divideix(double numero1, double numero2) {
        return numero1 / numero2;
    }

}
```

Obri la finestra de [Create/Update tests](#) per a la classe `Calculador`, en [Framework](#) elegeix [TestNG](#), marca únicament [Default Method Bodies](#) en la part [Generated Code](#). fes clic en [\[OK\]](#) per a crear la classe de prova.

```
public class CalculadorNGTest {
    public CalculadorNGTest() {
    }
    /**
     * Test of suma method, of class Calculador.
     */
    @Test
    public void testSuma() {
        System.out.println("suma");
        double numero1 = 0.0;
        double numero2 = 0.0;
        Calculador instance = new Calculador();
        double expResult = 0.0;
```





```

    double result = instance.suma(numero1, numero2);
    assertEquals(result, expectedResult, 0.0);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
/**
 * Test of resta method, of class Calculador.
 */
@Test
public void testResta() {
    System.out.println("resta");
    double numero1 = 0.0;
    double numero2 = 0.0;
    Calculador instance = new Calculador();
    double expectedResult = 0.0;
    double result = instance.resta(numero1, numero2);
    assertEquals(result, expectedResult, 0.0);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
/**
 * Test of multiplica method, of class Calculador.
 */
@Test
public void testMultiplica() {
    System.out.println("multiplica");
    double numero1 = 0.0;
    double numero2 = 0.0;
    Calculador instance = new Calculador();
    double expectedResult = 0.0;
    double result = instance.multiplica(numero1, numero2);
    assertEquals(result, expectedResult, 0.0);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
/**
 * Test of divideix method, of class Calculador.
 */
@Test
public void testDivideix() {
    System.out.println("divideix");
    double numero1 = 0.0;
    double numero2 = 0.0;
    Calculador instance = new Calculador();
    double expectedResult = 0.0;
    double result = instance.divideix(numero1, numero2);
    assertEquals(result, expectedResult, 0.0);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
}

```

El primer que cal fer, és llevar o comentar el fail, després cal modificar els test dels mètodes per a definir les proves que volem aplicar.



Comenta les anotacions de @Test dels mètodes menys el de suma, així sols s'executa el test sobre suma.

Realitzarem dues proves sobre la suma

- Sumar 1.00000001 més 1.1 i comprovar que dóna 2.1 amb una tolerància 0.0
- Sumar 0.998 més 0.997 i comprovar que dóna 2.0 amb una tolerància de 0.01.

```
@Test
public void testSuma() {
    System.out.println("suma");
    double numero1 = 1.00000001;
    double numero2 = 1.1;
    Calculador instance = new Calculador();           // crea la instància de la classe
    double expResult = 2.1;
    double result = instance.suma(numero1, numero2); // executa el mètode
    assertEquals(expResult, result, 0.0);           // si el resultat no es igual a l'esperat falla
    numero1 = 0.998;
    numero2 = 0.997;
    expResult = 2.0;
    result = instance.suma(numero1, numero2);
    assertEquals(expResult, result, 0.01, "tolerància 0.01"); // la tolerància és 0.01
}
```

Executem el test



La prova testSuma ha fallat, s'esperava el valor 2.10000001 i s'ha rebut el valor 2.1, la prova testSuma que ha fallat es la escrita en la línia 32 de CalculadorNGTest. Si una assertió falla llavors el mètode de prova finalitza, aquí falla la primera assertió, per tant no es comprova la segona.

És millor crear diversos mètodes de prova sobre un mateix mètode de la classe, així si falla un se segueix amb els següents

```
@Test
public void testSuma() {
    System.out.println("suma sense tolerància");
    double numero1 = 1.00000001;
    double numero2 = 1.1;
    Calculador instance = new Calculador();           // crea la instància de la classe
    double expResult = 2.1;
    double result = instance.suma(numero1, numero2); // executa el mètode
    assertEquals(expResult, result, 0.0);           // si el resultat no es igual a l'esperat falla
}
```



```

}
@Test
public void testSuma2() {
    System.out.println("suma amb tolerància");
    double numero1 = 0.998;
    double numero2 = 0.997;
    Calculador instance = new Calculador(); // crea la instància de la classe
    double expResult = 2.0;
    double result = instance.suma(numero1, numero2);
    assertEquals(expResult, result, 0.01, "tolerància 0.01"); // la tolerància és 0.01
}

```

Executem el test



S'han passat el 50% de les proves, la prova testSuma1 ha fallat i la prova testSuma2 no. La prova testSuma1 ha fallat, s'esperava el valor 2.10000001 i s'ha rebut el valor 2.1, testSuma1 falla perquè no té tolerància, la prova testSuma2 no falla, perquè, el valor rebut menys l'esperat ( $2.10000001 - 2.1 = 0.00000001$ ) està dins de la tolerància (0.01).

El següent test és per a la divisió, dividirem 0.0 per 0.0 i esperem un 0.0, però això falla.

```

@Test
public void testDivideix() {
    System.out.println("divideix");
    double numero1 = 0.0;
    double numero2 = 0.0;
    Calculador instance = new Calculador();
    double expResult = 0.0;
    double result = instance.divideix(numero1, numero2);
    assertEquals(result, expResult, 0.0);
}

```

La finestra de test és la següent:



En el test hi ha tres proves, dos per a la suma i una per a la divisió, i dos fallen per això apareix que s'ha passat el 33.33% de les proves. En dividir per 0 s'obté NaN (Not a Number, significa un resultat impossible de calcular)

Afegim els test següents

```
@Test
public void testResta() {
    System.out.println("resta");
    Calculador instance = new Calculador();
    double expectedResult = 0.0;
    double result = instance.resta(12.33, 12.333);
    assertEquals(result, expectedResult, 0.01);
}
```

```
@Test
public void testMultiplica() {
    System.out.println("multiplica");
    Calculador instance = new Calculador();
    double expectedResult = 0.0;
    double result = instance.multiplica(33.25, -0);
    assertEquals(result, expectedResult, 0.0);
}
```

Els dos test són correctes, la finestra de test és la següent:



## Exemple

Classe per a provar les assercions amb matrius.

```
public class Separa {
    public String[] separa(String text, String separador) {
        if (text == null || separador == null) {
            return null;
        }
        return text.split(separador);
    }
}
```

Creem la classe de proves amb TestNG i s'obté

```
public class SeparaNGTest {
```



```

public SeparaNGTestO {
}

/**
 * Test of separa method, of class Separa.
 */
@Test
public void testSeparaO {
    System.out.println("separa");
    String text = "";
    String separador = "";
    Separaa instance = new SeparaO;
    String[] expResult = null;
    String[] result = instance.separa(text, separador);
    assertEquals(result, expResult);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
}

```

Prova de si el mètode rep un null, llavors retorna un null, s'usa un assertNull.

```

@Test
public void testSepara1O {
    System.out.println("separa");
    String text = null;
    String separador = "/";
    Separa instance = new SeparaO;
    assertNull(instance.separa(text, separador));
}

```

Prova de si el mètode separa la cadena "Hola que tal Pascual" en les paraules "hola", "que", "tal" amb el separador espai en blanc, la prova falla

```

@Test
public void testSepara2O {
    System.out.println("separa");
    Separa instance = new SeparaO;
    String text = "Hola que tal Pascual";
    String[] expResult = new String[]{"hola", "que", "tal"};
    String[] result = instance.separa(text, " ");
    assertEquals(result, expResult);
}

```



canviem la línia

```
String[] expResult = new String[]{"hola", "que", "tal"};
```

a

```
String[] expResult = new String[]{"hola", "que", "tal", "Pascual"};
```

falla ja que la paraula “hola” en el text té majúscula



Prova de si el mètode separa la cadena "per una mirada, un mon, per un somriure, un cel" amb el separador “un” en les paraules "per", "a mirada", "mon, per", "somriure", "cel", la prova funciona

```
@Test
public void testSepara3() {
    System.out.println("separa");
    String text = "per una mirada, un mon, per un somriure, un cel";
    String separador = "un";
    Separa instance = new Separa();
    String[] expResult = new String[]{"per", "a mirada", "mon, per", "somriure", "cel"};
    String[] result = instance.separa(text, separador);
    assertEquals(result, expResult);
}
```

## Excepcions

En alguns casos de prova, el que s'espera com a eixida no és que el mètode ens retorne un determinat valor, sinó que es produïska una excepció.

Per a comprovar que l'excepció s'ha llançat podem optar per dos mètodes diferents

- Indicar l'excepció esperada en l'anotació @Test
- Utilitzar el mètode fail

### Indicar l'excepció esperada en @Test

Per a JUnit s'escriu

```
@Test(expected = java.lang.IllegalArgumentException.class)
```

Per a TestNG s'escriu



```
@Test(expectedExceptions = {java.lang.IllegalArgumentException.class})
```

El codi següent usa la notació de JUnit

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    Object o = emptyList.get(0);           // llança l'excepció ja que la llista està buida
}
```

dóna el test com a bo.

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    emptyList.add("uno");
    Object obj = emptyList.get(0);        // no llança l'excepció
}
```

dóna el test com a dolent, ja que no llança l'excepció.

Cal controlar un sol llançament en cada test.

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    Object o = emptyList.get(0);           // llança l'excepció ja que la llista està buida
    emptyList.add("uno");
    Object obj = emptyList.get(0);        // no llança l'excepció
}
```

dóna per bo el test, ja que s'ha llançat l'excepció.

## Utilitzar el mètode fail

El mètode fail ens permet indicar que hi ha una fallada en la prova, finalitza la prova quan s'executa. En aquest cas el que fem és cridar al mètode a provar dins d'un bloc try catch que captura l'excepció esperada, si no salta l'excepció, això vol dir que la prova ha fallat, per tant s'executa el fail.

```
@Test
public void testCalculaHoresExtras() {
    try {
        Empleat.calculaHoresExtras(-1.0, 20.0);
        fail("S'esperava l'excepció HoresException ");
    } catch (HoresException e) { }
    try {
        Empleat.calculaHoresExtras(10, -20.0);
        fail("S'esperava l'excepció PreuException");
    } catch (PreuException e) { }
}
```



El mètode calculaHoresExtras té dos paràmetres, el nombre d'hores i el preu per hora. Si el nombre d'hores és incorrecte llança l'excepció HoresException, si el preu de l'hora és incorrecte llança l'excepció PreuException.

Quan el mètode que estiguem provant puga llançar una excepció marcada (checked) que hem de capturar de manera obligatòria, també podem utilitzar fail dins del bloc catch per a notificar de la fallada en cas que es llance l'excepció de forma no esperada:

```
public void testCalculaSalari() {
    float resultatReal;
    try {
        resultatReal = Empleat.calculaSalari(2000.0, 8.0);
        double resultatEsperat = 1360.0;
        assertEquals(resultatEsperat, resultatReal, 0.01);
    } catch (MIException e) {
        fail("Llançada excepció no esperada MIException");
    }
}
```

Si el mètode provat llança una excepció no marcada (unchecked) que no hem controlat, llavors ho indica com un error no com una fallada de la prova.

## Exemple

La classe Matricad conté uns mètodes que treballen sobre una llista de cadenes.

```
public class Matricad {
    private java.util.ArrayList<String> cadenes;           // referència a la llista de cadenes, un camp
    /**
     * Constructor de Matricad.
     * @param dada matriu amb les cadenes per a la llista
     */
    public Matricad(String[] dada) {
        if ((dada == null) || (dada.length == 0)) {        // Verifiquem que la llista tinga valors
            throw new IllegalArgumentException();
        }
        this.cadenes = new java.util.ArrayList<>();
        for (String element : dada) {
            cadenes.add(element);
        }
    }
    /**
     * @return la cadena que té més caràcters. La primera si hi ha diverses amb la mateixa longitud
     */
    public String getMaxCad() {
        String max = "";
        for (String element : cadenes) {
            if (element.length() > max.length()) {
                max = element;
            }
        }
        return max;
    }
}
```





```

}
/**
 * @return la suma total de caràcters de totes les cadenes.
 */
public int getSumaCaracters() {
    int total = 0;
    for (String d : cadenes) {
        total += d.length();
    }
    return total;
}
/**
 * Retorna l'índex de la cadena buscada.
 *
 * @param unaCadena Cadena buscada
 * @return Retorna la posició d'un element dins de l'array
 * @throws java.util.NoSuchElementException Si l'element no existeix en la llista
 */
public int getIndexDe(String unaCadena)
    throws java.util.NoSuchElementException {
    if (unaCadena == null) { // Comprovem que l'argument siga vàlid
        throw new IllegalArgumentException();
    }
    int pos = 0;
    for (String d : cadenes) { // Recorrem la informació fins a trobar l'element
        if (d.equals(unaCadena)) {
            return pos;
        }
        pos++;
    }
    throw new java.util.NoSuchElementException(unaCadena); // L'element no existeix, llancem l'excepció
}
}

```

Quan es crea la unitat de proves, es marquen totes les opcions. Per a **TestNG** es genera la classe de proves següent

```

public class MatriCadNGTest {
    public MatriCadNGTest() {
    }
    @BeforeClass
    public static void setUpClass() throws Exception { // s'executa una única vegada a l'inici de la prova
    }
    @AfterClass
    public static void tearDownClass() throws Exception { // s'executa una única vegada al final de la prova
    }
    @BeforeMethod
    public void setUpMethod() throws Exception { // s'executa cada vegada a l'inici del mètode
    }
    @AfterMethod
    public void tearDownMethod() throws Exception { // s'executa cada vegada al final del mètode
    }
    /**
     * Test of getMaxCad method, of class MatriCad.

```



```

*/
@Test
public void testGetMaxCad() {
    System.out.println("getMaxCad");
    MatriCad instance = null;
    String expectedResult = "";
    String result = instance.getMaxCad();
    assertEquals(result, expectedResult);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
/**
 * Test of getMaxCaracters method, of class MatriCad.
 */
@Test
public void testGetSumaCaracters() {
    System.out.println("getSumaCaracters");
    MatriCad instance = null;
    int expectedResult = 0;
    int result = instance.getSumaCaracters();
    assertEquals(result, expectedResult);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
/**
 * Test of getIndexDe method, of class MatriCad.
 */
@Test
public void testGetIndexDe() {
    System.out.println("getIndexDe");
    String unaCadena = "";
    MatriCad instance = null;
    int expectedResult = 0;
    int result = instance.getIndexDe(unaCadena);
    assertEquals(result, expectedResult);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
}
}

```

Modifiquem la classe de proves, afegim el camp estàtic cadenes que contindrà una matriu de cadenes. El camp és estàtic perquè es va a usar en el mètode setUpClass que és estàtic.

S'eliminen els mètodes tearDownClass, setUpMethod i tearDownMethod, ja que no anem a usar-los.

Anem a fer tres proves al constructor, amb la matriu cadenes, un null i una matriu buida, s'ha de llançar l'excepció. La primera prova fallarà.

```

public class MatriCadNGTest {
    public MatriCadNGTest() {
    }
    static String[] cadenes; //matriu de cadenes
}

```



```

@BeforeClass
public static void setUpClass() throws Exception {
    cadenes = new String[]{"hui", "és", "dilluns", "i", "no", "m'agrada", "gens"}; // carrega la matriu de cadenes
}
/**
 * Test del constructor de la classe MatriCad. comprova que es llança
 * IllegalArgumentException si es crea amb una matriu de cadenes
 */
@Test(expectedExceptions = {java.lang.IllegalArgumentException.class})
public void crea0() {
    System.out.println("constructor amb la matriu cadenes");
    MatriCad instance = new MatriCad(cadenes); // falla, ja que no llança l'excepció
}
/**
 * Test del constructor de la classe MatriCad. comprova que es llança
 * IllegalArgumentException si es crea amb un null
 */
@Test(expectedExceptions = {java.lang.IllegalArgumentException.class})
public void crea1() {
    System.out.println("constructor amb un null");
    MatriCad instance = new MatriCad(null);
}
/**
 * Test del constructor de la classe MatriCad. comprova que es llança
 * IllegalArgumentException si es crea amb una matriu buida
 */
@Test(expectedExceptions = {java.lang.IllegalArgumentException.class})
public void crea2() {
    System.out.println("constructor amb una matriu buida");
    String[] cads = {};
    MatriCad instance = new MatriCad(cads);
}
}

```



La comprovació de la creació de Matricad amb un null o amb una matriu buida usant el fail és la següent

```

@Test
public void crea3() {
    try {
        System.out.println("constructor amb null");
        MatriCad instance = new MatriCad (null); // llança l'excepció i salta al catch i no executa el fail
        fail("no ha saltat l'excepció per a null");
    } catch (IllegalArgumentException e) {
    }
}
@Test

```



```

public void crea40 {
    try {
        System.out.println("constructor amb una matriu buida");
        String[] mat = {" "}; // la matriu conté una cadena buida, la matriu no està buida
        MatriCadinstance = new MatriCad (mat); // no llança l'excepció
        fail("no ha saltat l'excepció per a la matriu buida"); // executa el fail
    } catch (IllegalArgumentException e) {
    }
}
}

```

La segona prova falla.



Provem els mètodes getMaxCad i getSumaCaracters, la primera prova falla ja que “dilluns” no és la paraula més llarga.

```

public class MatriCadNGTest {
    public MatriCadNGTest() {
    }
    static String[] cadenes; //matriu de cadenes
    @BeforeClass
    public static void setUpClass() throws Exception {
        cadenes = new String[]{"hui", "és", "dilluns", "i", "no", "m'agrada", "gens"}; // carrega la matriu de cadenes
    }
    /**
     * Test of getMaxCad method, of class MatriCad. comprova que "dilluns" és la paraula més llarga
     */
    @Test
    public void testGetMaxCad() {
        System.out.println("getMaxCad");
        MatriCad instance = new MatriCad(cadenes);
        String expResult = "dilluns";
        String result = instance.getMaxCad();
        assertEquals(result, expResult);
    }
    /**
     * Test of getSumaCaracters method, of class MatriCad. comprova que les cadenes de la matriu
     * sumen 27 caràcters, es fixa un temps límit de 25 mil·lisegons per a l'execució del test
     */
    @Test(timeout = 25)
    public void testGetSumaCaracters() {
        System.out.println("getSumaCaracters");
        MatriCad instance = new MatriCad(cadenes);
        int expResult = 27;
        int result = instance.getSumaCaracters();
        assertEquals(expResult, result);
    }
}

```



En la prova de `getSumaCaracters`, si el temps d'execució excedeix de 25 mil·lisegons, llavors la prova falla.



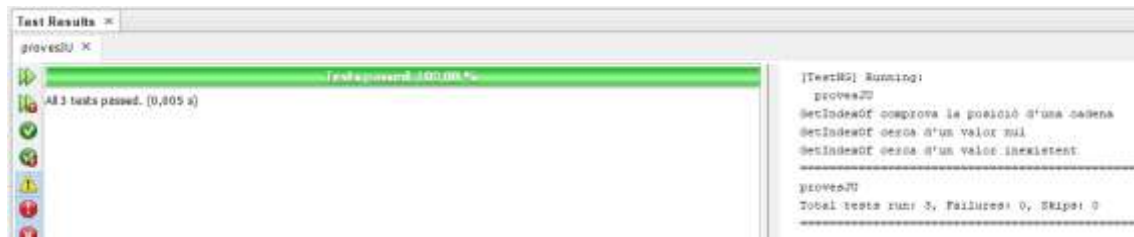
Anem a fer tres proves sobre el mètode `getIndexDe`, en la primera es comprova que la posició és la correcta, en les dos següents s'espera una excepció.

```
public class MatriCadNGTest {
    public MatriCadNGTest() {
    }
    static String[] cadenes; //matriu de cadenes
    @BeforeClass
    public static void setUpClass() throws Exception {
        cadenes = new String[]{"hui", "és", "dilluns", "i", "no", "m'agrada", "gens"}; // carrega la matriu de cadenes
    }
    /**
     * Test of getIndexDe method, of class MatriCad. comprova que la posició de "dilluns" és la 2
     */
    @Test
    public void testGetIndexDe() {
        System.out.println("GetIndexOf comprova la posició d'una cadena");
        MatriCad instance = new MatriCad(cadenes);
        int expectedResult = 2;
        int result = instance.getIndexDe("dilluns");
        assertEquals(expectedResult, result);
    }
    /**
     * Test of getIndexDe method, of class MatriCad. comprova que es llança
     * IllegalArgumentException si es cerca l'index de null
     */
    @Test(expectedExceptions = {java.lang.IllegalArgumentException.class})
    public void testGetIndexDe1() {
        System.out.println("GetIndexOf cerca d'un valor nul");
        MatriCad instance = new MatriCad(cadenes);
        instance.getIndexDe(null);
    }
    /**
     * Test of getIndexDe method, of class MatriCad. comprova que es llança
     * NoSuchElementException si es cerca un valor que no està en la matriu
     */
    @Test(expectedExceptions = {java.util.NoSuchElementException.class})
    public void testGetIndexDe2() {
        System.out.println("GetIndexOf cerca d'un valor inexistent");
        MatriCad instance = new MatriCad(cadenes);
        instance.getIndexDe("lunes");
    }
}
```



```
}
```

Els tres funcionen correctament



## Exercici

Classe per a provar les assercions amb valors booleans.

```
public class Vocals {  
    /**  
     * Comprova si la lletra és una vocal. el conjunt de vocals és "aeiouAEIOU"  
     * @param lletra cadena de text amb la lletra a comprovar  
     * @return <code>true</code> si és una vocal<br><code>false</code> si és no una vocal  
     */  
    public boolean esVocal(String lletra) { // rep una lletra  
        if (lletra.length() != 1) { // si lletra no té longitud 1  
            return false; // no és una lletra  
        }  
        String vocals = "aeiouAEIOU"; // cadenes amb les vocals  
        for (int i = 0; i < vocals.length(); i++) { // recorre les vocals  
            if (vocals.substring(i, i+1).equals(letra)) { // si la subcadena de 1 caràcter és la lletra  
                return true; // és una vocal  
            }  
        }  
        return false;  
    }  
}
```

Usant [TestNG](#) s'obté el mètode de prova següent, però com el mètode esVocal retorna un valor booleà, en lloc d'usar assertEquals usaràs assertTrue i assertFalse per a comprovar si s'obté un vertader o fals com a resposta.

assertTrue(result) espera que result siga vertader i assertFalse(result) que siga fals.

```
@Test  
public void testEsVocal() {  
    System.out.println("esVocal");  
    String lletra = "";  
    Vocals instance = new Vocals();  
    boolean expResult = false;  
    boolean result = instance.esVocal(letra);  
    assertEquals(result, expResult);  
    // TODO review the generated test code and remove the default call to fail.  
    fail("The test case is a prototype.");  
}
```



```
}
```

Crea el mètode `testEsVocalA` que comprova que la lletra “A” retorna un vertader.

Crea el mètode `testEsVocala` que comprova que la lletra “a” retorna un vertader.

Crea el mètode `testEsVocalB` que comprova que la lletra “B” retorna un fals.

Crea el mètode `testEsCad` que comprova que una cadena de text retorna un fals. En el `assertFalse(result, missatge)` afegeix un missatge que mostra el valor de la cadena (on es mostra el missatge).

## Exercici

Crea la classe `Empleat` amb els mètodes

```
public double calculaSalariBrut( String tipusEmpleat, double vendesMes, int horesExtra)
```

El salari base és de 1000 euros si l'empleat és de tipus “venedor”, i de 1500 euros si és de tipus “encarregat”. S'obté una prima de 100 euros si les vendes del mes són majors o iguals que 1000 euros, i de 200 euros són major de 2500 euros. Per cada hora extra es paga 18.57 euros. El salari brut és igual al salari base més la prima més les hores extres. Si tipus d'empleat és null o no és “venedor” ni “encarregat” el mètode llança una excepció de tipus `MaException` amb el missatge “*el tipus de vendedor no és correcte*”. Si `ventesMes` o `horesExtra` prenen valors negatius el mètode llança una excepció de tipus `MaException` amb el missatge “*el valor no pot ser negatiu*”.

```
public double calculaSalariNet(double salariBrut)
```

Si el salari brut va

- de 0 a 999 euros no s'aplicarà cap retenció
- de 1000 a 1500 euros se'ls aplicarà un 16% de retenció
- de 1501 euros i superiors se'ls aplicarà un 20% de retenció

El mètode retorna `salariBrut * (1 - retenció)`

Si salari brut és negatiu el mètode llança una excepció de tipus `MaException` amb el missatge “*el valor no pot ser negatiu*”.

Realitza les proves següents,

Per al mètode `calculaSalariNet`

- entrada = 2000, eixida = 1600
- entrada = 500, eixida = 500
- entrada = 0, eixida = 0



- entrada = 1500, eixida = 1200
- entrada = 1499.99, eixida = 1259.99
- entrada = 1251, eixida = 1050.845
- entrada = 999.99, eixida = 999.99
- entrada = -1, eixida = MaException

Per al mètode calculaSalariBrut

- entrada = "venedor", 2000.0, 8, eixida = 1360
- entrada = "venedor", 1500.55, 3, eixida = 1155.7
- entrada = "venedor", 2500.0, 10, eixida = 1285
- entrada = "venedor", 2500.01, 10, eixida = 1285.7
- entrada = "encarregat", 2500.0, 10, eixida = 1285
- entrada = "encarregat", 999.99, 3, eixida = 1555.7
- entrada = "venedor", -1, 0, eixida = MaException
- entrada = "venedor", 2125.7, -1, eixida = MaException
- entrada = "Encarregat", 999.99, 3, eixida = MaException
- entrada = "vendedor", 999.99, 3, eixida = MaException
- entrada = null, 0, 0, eixida = MaException

Menys les tres primeres, les altres fallen, corregeix-les.

