EBook Gratis

APRENDIZAJE junit

Free unaffiliated eBook created from
Stack Overflow contributors.

#junit

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: junit

It is an unofficial and free junit ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official junit.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Junit

Observaciones

JUnit es un marco simple para escribir pruebas repetibles para el lenguaje de programación Java . Es una instancia de la arquitectura xUnit para marcos de prueba de unidades.

Las características principales consisten en:

- **Afirmaciones** , que le permiten personalizar cómo probar los valores en sus pruebas
- **Corredores de prueba** , que le permiten especificar cómo ejecutar las pruebas en su clase
- **Reglas** , que le permiten modificar con flexibilidad el comportamiento de las pruebas en su clase
- **Suites** , que le permiten construir juntos un conjunto de pruebas de muchas clases diferentes

Extensión útil para JUnit:

- AssertJ : Afirmaciones fluidas para java
- Mockito : marco de burla para java

Versiones

Versión	Fecha de lanzamiento
JUIT 5 Hito 2	2016-07-23
JUIT 5 Hito 1	2016-07-07
JUnit 4.12	2016-04-18
JUnit 4.11	2012-11-14
JUIT 4.10	2011-09-28
JUnit 4.9	2011-08-22
JUnit 4.8	2009-12-01
JUIT 4,7	2009-07-28
JUIT 4,6	2009-04-14

Examples

Instalación o configuración

Ya que JUnit es una biblioteca de Java, todo lo que tiene que hacer para instalarla es agregar algunos archivos JAR en la ruta de clases de su proyecto Java y estará listo para comenzar.

Puede descargar estos dos archivos JAR manualmente: junit.jar & hamcrest-core.jar .

Si está utilizando Maven, simplemente puede agregar una dependencia en su `pom.xml` :

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

O si está utilizando Gradle, agregue una dependencia en su `build.gradle` :

```
apply plugin: 'java'

dependencies {
    testCompile 'junit:junit:4.12'
}
```

Después de esto puedes crear tu primera clase de prueba:

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

import org.junit.Test;

public class MyTest {
    @Test
    public void onePlusOneShouldBeTwo() {
        int sum = 1 + 1;
        assertEquals(2, sum);
    }
}
```

y ejecutarlo desde la línea de comando:

- Windows `java -cp .;junit-X.YY.jar;hamcrest-core-XYjar org.junit.runner.JUnitCore MyTest`
- Linux o OsX `java -cp .:junit-X.YY.jar:hamcrest-core-XYjar org.junit.runner.JUnitCore MyTest`

o con Maven: `mvn test`

Ejemplo de prueba de unidad básica

Este ejemplo es una configuración básica para unittesting StringBuilder.toString () usando junit.

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

import org.junit.Test;
```

```
public class StringBuilderTest {

    @Test
    public void stringBuilderAppendShouldConcatinate()  {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("String");
        stringBuilder.append("Builder");
        stringBuilder.append("Test");

        assertEquals("StringBuilderTest", stringBuilder.toString());
    }

}
```

@Antes después

Se ejecutará un método anotado con `@Before` antes de cada ejecución de los métodos `@Test` . Análogo un `@After` método anotado es ejecutado después de cada `@Test` método. Esto se puede usar para establecer repetidamente una configuración de prueba y limpiar después de cada prueba. Así que las pruebas son independientes y el código de preparación no se copia dentro del método `@Test` .

Ejemplo:

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

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

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

public class DemoTest {

    private List<Integer> list;

    @Before
    public void setUp() {
        list = new ArrayList<>();
        list.add(3);
        list.add(1);
        list.add(4);
        list.add(1);
        list.add(5);
        list.add(9);
    }

    @After
    public void tearDown() {
        list.clear();
    }

    @Test
    public void shouldBeOkToAlterTestData() {
        list.remove(0); // Remove first element of list.
```

```
        assertEquals(5, list.size()); // Size is down to five
    }

    @Test
    public void shouldBeIndependentOfOtherTests() {
        assertEquals(6, list.size());
    }
}
```

Los métodos anotados con `@Before` o `@After` deben ser `public void` y con cero argumentos.

Atrapa la excepción esperada

Es posible atrapar fácilmente la excepción sin ningún bloque `try catch` .

```
public class ListTest {
  private final List<Object> list = new ArrayList<>();

  @Test(expected = IndexOutOfBoundsException.class)
  public void testIndexOutOfBoundsException() {
    list.get(0);
  }
}
```

El ejemplo anterior debería ser suficiente para casos más simples, cuando no desea o necesita verificar el mensaje transmitido por la excepción lanzada.

Si desea verificar la información sobre la excepción, puede usar el bloque try / catch:

```
@Test
public void testIndexOutOfBoundsException() {
    try {
        list.get(0);
        Assert.fail("Should throw IndexOutOfBoundException");
    } catch (IndexOutOfBoundsException ex) {
        Assert.assertEquals("Index: 0, Size: 0", ex.getMessage());
    }
}
```

Para este ejemplo, debe tener en cuenta que siempre debe agregar `Assert.fail()` para asegurarse de que la prueba no se `Assert.fail()` cuando no se lance una excepción.

Para casos más elaborados, JUnit tiene la `ExpectedException @Rule` , que también puede probar esta información y se utiliza de la siguiente manera:

```
public class SimpleExpectedExceptionTest {
     @Rule
     public ExpectedException expectedException = ExpectedException.none();

     @Test
     public void throwsNothing() {
         // no exception expected, none thrown: passes.
     }
```

```
    @Test
    public void throwsExceptionWithSpecificType() {
        expectedException.expect(NullPointerException.class);

        throw new NullPointerException();
    }

    @Test
    public void throwsExceptionWithSpecificTypeAndMessage() {
        expectedException.expect(IllegalArgumentException.class);
        expectedException.expectMessage("Wanted a donut.");

        throw new IllegalArgumentException("Wanted a donut.");
    }
}
```

Pruebas de excepciones en JUnit5

Para lograr lo mismo en JUnit 5, utiliza un mecanismo completamente nuevo :

El metodo probado

```
public class Calculator {
    public double divide(double a, double b) {
        if (b == 0.0) {
            throw new IllegalArgumentException("Divider must not be 0");
        }
        return a/b;
    }
}
```

El metodo de prueba

```
public class CalculatorTest {
    @Test
    void triangularMinus5() { // The test method does not have to be public in JUnit5
        Calculator calc = new Calculator();

        IllegalArgumentException thrown = assertThrows(
            IllegalArgumentException.class,
            () -> calculator.divide(42.0, 0.0));
        // If the exception has not been thrown, the above test has failed.

        // And now you may further inspect the returned exception...
        // ...e.g. like this:
        assertEquals("Divider must not be 0", thrown.getMessage());
}
```

Ignorando las pruebas

Para ignorar una prueba, simplemente agregue la anotación `@Ignore` a la prueba y, opcionalmente, proporcione un parámetro a la anotación con el motivo.

```
@Ignore("Calculator add not implemented yet.")
@Test
public void testPlus() {
    assertEquals(5, calculator.add(2,3));
}
```

En comparación con comentar la prueba o eliminar la anotación `@Test` , el corredor de la prueba seguirá informando esta prueba y notará que se ignoró.

También es posible ignorar un caso de prueba condicionalmente utilizando *supuestos de* JUnit. Un ejemplo de caso de uso sería ejecutar el caso de prueba solo después de que un desarrollador arregle un determinado error. Ejemplo:

```
import org.junit.Assume;
import org.junit.Assert;
...

@Test
public void testForBug1234() {

    Assume.assumeTrue(isBugFixed(1234));//will not run this test until the bug 1234 is fixed

    Assert.assertEquals(5, calculator.add(2,3));
}
```

El corredor predeterminado trata las pruebas con suposiciones fallidas como ignoradas. Es posible que otros corredores se comporten de manera diferente, por ejemplo, trátelos como pasados.

JUnit - Ejemplos básicos de anotación.

Aquí hay algunas anotaciones básicas de JUnit que debes entender:

```
@BeforeClass – Run once before any of the test methods in the class, public static void
@AfterClass – Run once after all the tests in the class has been run, public static void
@Before – Run before @Test, public void
@After – Run after @Test, public void
@Test – This is the test method to run, public void
```

Lea Empezando con Junit en línea: https://riptutorial.com/es/junit/topic/1838/empezando-con-junit

Capítulo 2: Generar esqueleto de casos de prueba Junit para código existente

Introducción

A veces, necesita generar el esqueleto para los casos de prueba según las clases que tenga en su proyecto.

Examples

Genere el esqueleto de casos de prueba de Junit para el código existente en Eclipse

Aquí están los pasos para generar el esqueleto de prueba para el código existente:

1. Abra **Eclipse** y elija el proyecto para el que desea crear casos de prueba.
2. En el **Explorador de paquetes** , seleccione la clase java para la que desea generar la prueba Junit.
3. Ir a **Archivo** -> **Nuevo** -> **Casos de prueba de Junit**
4. Cambie la **carpeta de origen** para que apunte a la prueba utilizando Examinar (Nota: es mejor separar el código fuente del código de prueba)
5. Cambie el **paquete** según el paquete de destino que desee
6. En la **clase bajo prueba** , asegúrese de ingresar la clase para la que desea generar los casos de prueba.
7. Haga clic en **siguiente**
8. Seleccione los métodos que desea probar
9. Haga clic en **Finalizar**

Ahora, tendrá una clase Junit generada para probar la clase fuente que tiene

Lea Generar esqueleto de casos de prueba Junit para código existente en línea: https://riptutorial.com/es/junit/topic/8648/generar-esqueleto-de-casos-de-prueba-junit-para-codigo-existente

Capítulo 3: Ignorar los casos de prueba en Junit.

Introducción

A veces desea ignorar algunos de los casos de prueba que tiene en Junit. Por ejemplo, están parcialmente terminados y desea volver a ellos más tarde.

Examples

Ignorar caso de prueba en Junit

1. Ve al método de prueba que quieres ignorar.
2. Antes de la anotación `@Test` , ingrese `@Ignore`
3. opcional: puede agregar una descripción por qué está ignorando este método de prueba, algo como: `@Ignore ("ignoring this test case for now")`

un método de muestra sería:

```
@Ignore ("not going to test this method now")
@Test
public void test() {
     assertfalse(true);
}
```

Lea Ignorar los casos de prueba en Junit. en línea:
https://riptutorial.com/es/junit/topic/8640/ignorar-los-casos-de-prueba-en-junit-

Capítulo 4: Orden de ejecución de prueba

Sintaxis

- @FixMethodOrder // Ejecuta la prueba utilizando el clasificador de métodos predeterminado
- @FixMethodOrder (MethodSorters) // Ejecuta la prueba utilizando MethodSorter asociado con la enumeración de los MethodSorters.

Examples

Orden por defecto

Utilice la anotación - `@FixMethodOrder(MethodSorters.DEFAULT)` . Esto ejecuta todas las pruebas dentro de la clase en un orden determinista y algo predecible. La implementación contiene los nombres de los métodos y los compara. En el escenario de un empate, se ordena por orden lexicográfico.

Segmento de código a continuación tomado de JUnit Github - MethodSorter.java

```
public int compare(Method m1, Method m2) {
    int i1 = m1.getName().hashCode();
    int i2 = m2.getName().hashCode();
    if(i1 != i2) {
        return i1 < i2 ? -1 : 1;
    }
    return NAME_ASCENDING.compare(m1,m2);
}
```

Ejemplo

```
@FixMethodOrder(MethodSorters.DEFAULT)
public class OrderedTest {
    @Test
    public void testA() {}

    @Test
    public void testB() {}

    @Test
    public void testC() {}
}
```

Supongamos que los hashes para `testA` , `testB` y `testC` son 3, 2 y 1 respectivamente. Entonces la orden de ejecución es

1. testC
2. testB
3. testA

Supongamos que los hashes para todas las pruebas son iguales. Dado que todos los hashes son iguales, el orden de ejecución se basa en un orden lexicográfico. La orden de ejecución es

1. testA
2. testB
3. testC

Orden lexicografico

Utilice la anotación `@FixMethodOrder` con el clasificador de métodos `MethodSorters.NAME_ASCENDING` . Esto ejecutará todas las pruebas dentro de la clase en un orden determinista y predecible. La implementación compara los nombres de los métodos y, en el caso de un empate, compara los métodos ' `toString()` .

Segmento de código a continuación tomado de JUnit Github - MethodSorter.java

```
public int compare(Method m1, Method m2) {
    final int comparison = m1.getName().compareTo(m2.getName());
    if(comparison != 0) {
        return comparison;
    }
    return m1.toString().compareTo(m2.toString());
}
```

Ejemplo

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class OrderedTest {
    @Test
    public void testA() {}

    @Test
    public void testB() {}

    @Test
    public void testC() {}
}
```

La orden de ejecución es

1. testA
2. testB
3. testC

Lea Orden de ejecución de prueba en línea: https://riptutorial.com/es/junit/topic/5905/orden-de-ejecucion-de-prueba

Capítulo 5: Pruebas

Observaciones

Parámetro	Contexto	Detalles
@Antes de clase	Estático	Ejecutado cuando la clase se crea por primera vez
@Antes de	Ejemplo	Ejecutado antes de *cada prueba* en la clase.
@Prueba	Ejemplo	Se debe declarar cada método a probar.
@Después	Ejemplo	Ejecutado después de *cada prueba* en la clase.
@Después de clases	Estático	Ejecutado antes de la destrucción de la clase.

Ejemplo de formato de clase de prueba

```
public class TestFeatureA {

    @BeforeClass
    public static void setupClass() {}

    @Before
    public void setupTest() {}

    @Test
    public void testA() {}

    @Test
    public void testB() {}

    @After
    public void tearDownTest() {}

    @AfterClass
    public static void tearDownClass() {}

    }
}
```

Examples

Pruebas unitarias utilizando JUnit

Aquí tenemos una clase `Counter` con métodos `countNumbers()` y `hasNumbers()` .

```
public class Counter {

    /* To count the numbers in the input */
```

```
    public static int countNumbers(String input) {
        int count = 0;
        for (char letter : input.toCharArray()) {
            if (Character.isDigit(letter))
                count++;
        }
        return count;
    }

    /* To check whether the input has number*/
    public static boolean hasNumber(String input) {
        return input.matches(".*\\d.*");
    }
}
```

Para la prueba unitaria de esta clase, podemos usar el framework Junit. Agregue el `junit.jar` en la `junit.jar` de su clase de proyecto. Luego crea la clase de caso de prueba como se muestra a continuación:

```
import org.junit.Assert; // imports from the junit.jar
import org.junit.Test;

public class CounterTest {

    @Test // Test annotation makes this method as a test case
    public void countNumbersTest() {
        int expectedCount = 3;
        int actualCount = Counter.countNumbers("Hi 123");
        Assert.assertEquals(expectedCount, actualCount); //compares expected and actual value
    }

    @Test
    public void hasNumberTest() {
        boolean expectedValue = false;
        boolean actualValue = Counter.hasNumber("Hi there!");
        Assert.assertEquals(expectedValue, actualValue);
    }
}
```

En su IDE puede ejecutar esta clase como "Junit testcase" y ver el resultado en la GUI. En el símbolo del sistema, puede compilar y ejecutar el caso de prueba de la siguiente manera:

```
\> javac -cp ,;junit.jar CounterTest.java
\> java  -cp .;junit.jar org.junit.runner.JUnitCore CounterTest
```

El resultado de una ejecución de prueba exitosa debe ser similar a:

```
JUnit version 4.9b2
..
Time: 0.019

OK (2 tests)
```

En el caso de una falla de prueba se vería más como:

```
Time: 0.024
There was 1 failure:
1) CountNumbersTest(CounterTest)
java.lang.AssertionError: expected:<30> but was:<3>
... // truncated output
FAILURES!!!
Tests run: 2,  Failures: 1
```

Accesorios

De Wikipedia :

> Un dispositivo de prueba es algo que se usa para probar consistentemente algún elemento, dispositivo o pieza de software.

También puede mejorar la legibilidad de las pruebas al extraer el código común de inicialización / finalización de los métodos de prueba en sí.

Cuando se puede ejecutar una inicialización común una vez en lugar de antes de cada prueba, esto también puede reducir la cantidad de tiempo que se tarda en ejecutar las pruebas.

El siguiente ejemplo está diseñado para mostrar las opciones principales proporcionadas por JUnit. Supongamos una clase `Foo` que es caro inicializar:

```
public class Foo {
    public Foo() {
        // expensive initialization
    }

    public void cleanUp() {
        // cleans up resources
    }
}
```

Otra clase de `Bar` tiene una referencia a `Foo` :

```
public class Bar {
    private Foo foo;

    public Bar(Foo foo) {
        this.foo = foo;
    }

    public void cleanUp() {
        // cleans up resources
    }
}
```

Las siguientes pruebas esperan un contexto inicial de una Lista que contenga una única `Bar` .

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

import java.util.Arrays;
```

```
import java.util.List;

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

public class FixturesTest {

    private static Foo referenceFoo;

    private List<Bar> testContext;

    @BeforeClass
    public static void setupOnce() {
        // Called once before any tests have run
        referenceFoo = new Foo();
    }

    @Before
    public void setup() {
        // Called before each test is run
        testContext = Arrays.asList(new Bar(referenceFoo));
    }

    @Test
    public void testSingle() {
        assertEquals("Wrong test context size", 1, testContext.size());
        Bar baz = testContext.get(0);
        assertEquals(referenceFoo, baz.getFoo());
    }

    @Test
    public void testMultiple() {
        testContext.add(new Bar(referenceFoo));
        assertEquals("Wrong test context size", 2, testContext.size());
        for (Bar baz : testContext) {
            assertEquals(referenceFoo, baz.getFoo());
        }
    }

    @After
    public void tearDown() {
        // Called after each test is run
        for (Bar baz : testContext) {
            baz.cleanUp();
        }
    }

    @AfterClass
    public void tearDownOnce() {
        // Called once after all tests have run
        referenceFoo.cleanUp();
    }
}
```

En el ejemplo de la `@BeforeClass` método anotado `setupOnce()` se utiliza para crear la `Foo` objeto, que es caro para inicializar. Es importante que no se modifique por ninguna de las pruebas, de lo contrario, el resultado de la ejecución de la prueba podría depender del orden de ejecución de las

pruebas individuales. La idea es que cada prueba sea independiente y pruebe una pequeña característica.

El `@Before` método anotado `setup()` configura el contexto de prueba. El contexto puede modificarse durante la ejecución de la prueba, por lo que debe inicializarse antes de cada prueba. El efecto equivalente podría lograrse incluyendo el código contenido en este método al comienzo de cada método de prueba.

El `@After` anotada método `tearDown()` limpia los recursos dentro del contexto de prueba. Se llama después de cada invocación de prueba y, como tal, a menudo se utiliza para liberar recursos asignados en un método anotado `@Before` .

El `@AfterClass` método anotado `tearDownOnce()` limpia los recursos una vez que todas las pruebas se han ejecutado. Estos métodos se utilizan normalmente para liberar recursos asignados durante la inicialización o en un método anotado `@BeforeClass` . Dicho esto, probablemente es mejor evitar los recursos externos en las pruebas unitarias para que las pruebas no dependan de nada fuera de la clase de prueba.

Pruebas unitarias utilizando teorías.

Desde JavaDoc

> Theory Theory Runner permite probar cierta funcionalidad contra un subconjunto de un conjunto infinito de puntos de datos.

Teorias corrientes

```
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;
import org.junit.runner.RunWith;

@RunWith(Theories.class)
public class FixturesTest {

    @Theory
    public void theory(){
        //...some asserts
    }
}
```

Los métodos anotados con `@Theory` serán leídos como teorías por el corredor de teorías.

@DataPoint anotación

```
@RunWith(Theories.class)
public class FixturesTest {

    @DataPoint
    public static String dataPoint1 = "str1";
    @DataPoint
    public static String dataPoint2 = "str2";
    @DataPoint
```

```
    public static int intDataPoint = 2;

    @Theory
    public void theoryMethod(String dataPoint, int intData){
        //...some asserts
    }
}
```

Cada campo anotado con `@DataPoint` se usará como un parámetro de método de un tipo dado en las teorías. En el ejemplo anterior, `theoryMethod` se ejecutará dos veces con los siguientes parámetros: `["str1", 2] , ["str2", 2]`

@DataPoints anotación @RunWith (Theories.class) clase pública FixturesTest {

```
    @DataPoints
    public static String[] dataPoints = new String[]{"str1", "str2"};
    @DataPoints
    public static int[] dataPoints = new int[]{1, 2};

    @Theory
    public void theoryMethod(String dataPoint, ){
        //...some asserts
    }
}
```

Cada elemento de la matriz anotada con la anotación `@DataPoints` se usará como un parámetro de método de un tipo dado en las teorías. En el ejemplo anterior, `theoryMethod` se ejecutará cuatro veces con los siguientes parámetros: `["str1", 1], ["str2", 1], ["str1", 2], ["str2", 2]`

Medición del desempeño

Si necesita verificar si su método de prueba demora demasiado en ejecutarse, puede hacerlo mencionando el tiempo de ejecución esperado usando la propiedad de tiempo de espera de la anotación @Test. Si la ejecución de la prueba lleva más de esa cantidad de milisegundos, el método de prueba falla.

```
public class StringConcatenationTest {

    private static final int TIMES = 10_000;

    // timeout in milliseconds
    @Test(timeout = 20)
    public void testString(){

        String res = "";

        for (int i = 0; i < TIMES; i++) {
            res += i;
        }

        System.out.println(res.length());
    }

    @Test(timeout = 20)
    public void testStringBuilder(){
```

```
        StringBuilder res = new StringBuilder();

        for (int i = 0; i < TIMES; i++) {
            res.append(i);
        }

        System.out.println(res.length());
    }

    @Test(timeout = 20)
    public void testStringBuffer(){

        StringBuffer res = new StringBufferr();

        for (int i = 0; i < TIMES; i++) {
            res.append(i);
        }

        System.out.println(res.length());
    }

}
```

En la mayoría de los casos sin JVM, la `testString` de `testString` fallará. Pero `testStringBuffer` y `testStringBuilder` deberían pasar esta prueba con éxito.

Lea Pruebas en línea: https://riptutorial.com/es/junit/topic/5414/pruebas

Capítulo 6: Pruebas con proveedores de datos

Examples

Instalación y uso

Instalación:

Para utilizar DataProviders, necesita junit-dataprovider .jar:

Github

Descarga directa

Hamcrest-core-1.3.jar:

Github

Descarga directa

Y agrega ambos .jar a tu proyecto.

Uso:

Agregue esta `import` a su código:

```
import com.tngtech.java.junit.dataprovider.DataProvider;
import com.tngtech.java.junit.dataprovider.DataProviderRunner;
import com.tngtech.java.junit.dataprovider.UseDataProvider;
```

Antes de la declaración de su clase:

```
@RunWith(DataProviderRunner.class)
```

Así que se ve así:

```
@RunWith(DataProviderRunner.class)
public class example {
    //code
}
```

Cómo crear DataProviders:

Antes de la función que desee que sea un proveedor de datos, agregue este decorador:

```
@DataProvider
```

Así se vería así:

```
@DataProvider
public static Object[][] testExampleProvider() {
    return new Object[][]{
        {"param1", "param2", number1}
        {"param1", "param2", number1}
        //You can put as many parameters as you want
    };
}
```

Cómo utilizar DataProviders:

Antes de cualquier función que desee que obtenga los parámetros que devolvemos del proveedor de datos, agregue este decorador:

```
@UseDataProvider("testExampleProvider")
```

Entonces tu función para probar se ve así:

```
@Test
@UseDataProvider("testExampleProvider")
public void testAccount(String param1, String param2, int number) {
    //System.out.println("exampleOfDataProviders");
    //assertEquals(...);
    //assertEquals(...);
}
```

Lea Pruebas con proveedores de datos en línea:
https://riptutorial.com/es/junit/topic/7469/pruebas-con-proveedores-de-datos

Capítulo 7: Pruebas de paramaterización

Introducción

A veces, tiene una prueba que necesita ejecutar varias veces, cada vez con datos diferentes. La parametrización de la prueba le permite hacer esto de una manera fácil y fácil de mantener.

Sintaxis

- @RunWith (Parameterized.class) // anotación para la clase de prueba

  @ Parámetros // anotación para los datos

Observaciones

Un beneficio de usar parámetros es que si falla un conjunto de datos, la ejecución simplemente se moverá al siguiente conjunto de datos en lugar de detener toda la prueba.

Examples

Usando un constructor

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.is;
import java.util.*;
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class SimpleParmeterizedTest {
    @Parameters
    public static Collection<Object[]> data(){
        return Arrays.asList(new Object[][]{
                {5, false}, {6, true}, {8, true}, {11, false}
        });
    }

    private int input;
    private boolean expected;

    public SimpleParmeterizedTest(int input, boolean expected){
        this.input = input;
        this.expected = expected;
    }

    @Test
    public void testIsEven(){
        assertThat(isEven(input), is(expected));
```

```
    }
}
```

En datos (), usted suministra los datos que se utilizarán en las pruebas. Junit recorrerá los datos y ejecutará la prueba con cada conjunto de datos.

Lea Pruebas de paramaterización en línea: https://riptutorial.com/es/junit/topic/9425/pruebas-de-paramaterizacion

Capítulo 8: Reglas de prueba personalizadas

Observaciones

Hay beneficios para cualquiera. Extender `ExternalResource` es conveniente, especialmente si solo requerimos un `before()` para configurar algo.

Sin embargo, debemos tener en cuenta que, dado que el método `before()` se ejecuta fuera del `try...finally` , cualquier código que se requiera para limpiar `after()` no se ejecutará si hay un error durante el Ejecución de `before()` .

Así es como se ve dentro de `ExternalResource` :

```
before();
try {
    base.evaluate();
} finally {
    after();
}
```

Obviamente, si se lanza alguna excepción en la prueba en sí, o por otra regla anidada, aún se ejecutará el siguiente.

Examples

Custom @TestRule por implementación

Esto es especialmente útil si tenemos una clase que queremos extender en la regla. Vea el ejemplo a continuación para un método más conveniente.

```
import org.junit.rules.TestRule;
import org.junit.runners.model.Statement;

public class AwesomeTestRule implements TextRule {

    @Override
    public Statement apply(Statement base, Description description) {
        return new AwesomeStatement(base);
    }

    private static class AwesomeStatement extends Statement {

        private Statement base;

        public AwesomeStatement(Statement base) {
            this.base = base;
        }

        @Override
        public void evaluate() throws Throwable {
            try {
```

```
                // do your magic
                base.evaluate(); // this will call Junit to run an individual test
            } finally {
                // undo the magic, if required
            }
        }
    }

}
```

Custom @TestRule por extensión

JUnit tiene una implementación abstracta de `@TestRule` que le permite escribir una regla de una manera más sencilla. Esto se llama `ExternalResource` y proporciona dos métodos protegidos que pueden extenderse de la siguiente manera:

```
public class AwesomeTestRule extends ExternalResource {

    @Override
    protected void before() {
        // do your magic
    }

    @Override
    protected void after() {
        // undo your magic, if needed
    }

}
```

Lea Reglas de prueba personalizadas en línea: https://riptutorial.com/es/junit/topic/9808/reglas-de-prueba-personalizadas

Creditos

S. No	Capítulos	Contributors
1	Empezando con Junit	acdcjunior, Andrii Abramov, Ashish Bhavsar, cheffe, Community , Daniel Käfer, Honza Zidek, Lachezar Balev, NamshubWriter, nishizawa23, Roland Weisleder, Rufi, Simulant, Squidward, svgameren, t0mppa, Tim Tong, Tomasz Bawor
2	Generar esqueleto de casos de prueba Junit para código existente	Hamzawey
3	Ignorar los casos de prueba en Junit.	Hamzawey
4	Orden de ejecución de prueba	Tim Tong
5	Pruebas	Andrii Abramov, gar, Ram, Robert, Sergii Bishyr, Squidward
6	Pruebas con proveedores de datos	Manuel
7	Pruebas de paramaterización	Hai Vu, selotape, user7491506
8	Reglas de prueba personalizadas	pablisco