UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

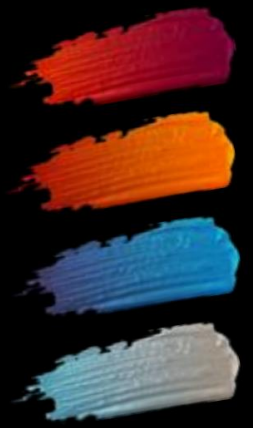
SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

JUnit: test parametrizzati

Azzurra Ragone

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 | Fax: +39.080.5442536
serlab.di.uniba.it



Cosa sono i test parametrici?

I test parametrizzati consentono di eseguire un test più volte con argomenti diversi.

Test parametrizzati

Dichiarato Appena Piace metodi @Test regolari ma con l'annotazione @ParameterizedTest.

Devi dichiarare almeno una fonte che fornirà gli argomenti per ogni invocazione e quindi consumare gli argomenti nel metodo di prova.

Test parametrizzati

Esempio molto semplice

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue( condition: argument > 0 && argument < 4);
}
```

Test parametrizzati: sorgenti nulle e vuote

Gestire i valori **Null** ed **Empty** forniti ai nostri test parametrizzati

Controllare **i casi limite** e verificare il corretto comportamento del nostro software quando riceve input ***errati***

- @NullSource: fornisce un **singolo argomento nullo** per metodo @ParameterizedTest annotato.
- @EmptySource: fornisce un **singolo argomento vuoto** al metodo @ParameterizedTest annotato per i parametri di quanto segue tipi: java.lang.String, java.util.List, java.util.Set, java.util.Map, array primitivi (ad esempio, int[], char[][], ecc.), array di oggetti (ad esempio, String[], Integer[][], ecc.).

Test parametrizzati: sorgenti nulle e vuote

Gestire i valori **Null** ed **Empty** forniti ai nostri test parametrizzati

Controllare **i casi limite** e verificare il corretto comportamento del nostro software quando riceve input ***errati***

- @NullSource
- @EmptySource •

@NullAndEmptySource: un'annotazione composta che combina le funzionalità di @NullSource e @EmptySource.

Test parametrizzati: sorgenti nulle e vuote

Gestire i valori **Null** ed **Empty** forniti ai nostri test parametrizzati

```
@ParameterizedTest
@NullSource
void checkNull(String value) {
    assertEquals( expected: null,  actual: value);
}

@ParameterizedTest
@EmptySource
void checkEmpty(String value) {
    assertEquals( expected: "",  actual: value);
}
```

Test parametrizzati: sorgenti nulle e vuote

Gestire i valori **Null** e **vuoti** forniti ai nostri test parametrizzati

```
@ParameterizedTest
@NullAndEmptySource
void checkNullAndEmpty(String value) {
    assertTrue( condition: value == null || value.isEmpty());
}
```

Test parametrizzati: sorgenti nulle e vuote

- @NullSource: fornisce un **singolo argomento nullo** al metodo annotato @ParameterizedTest.
- @EmptySource: fornisce un **singolo argomento vuoto** al metodo @ParameterizedTest annotato

```
@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue( condition: text == null || text.trim().isEmpty());
}
```

Quante volte viene eseguito il metodo di prova?

Test parametrizzati: sorgenti nulle e vuote

- @NullAndEmptySource: un'annotazione composta che combina il funzionalità di @NullSource e @EmptySource.

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = { " ", "   ", "\t", "\n" })
void nullEmptyAndBlankStrings2(String text) {
    assertTrue( condition: text == null || text.trim().isEmpty());
}
```

Test parametrizzati: @MethodSource

- @MethodSource consente di fare riferimento a uno o più metodi di fabbrica
- Ogni **metodo factory** genera un **flusso di argomenti**

  (ad esempio, Stream<Argomenti>)
- Ogni insieme di argomenti all'interno del flusso saranno gli argomenti fisici

  per singole invocazioni del @ParameterizedTest annotato

  metodo
- Un "flusso" è qualsiasi cosa che JUnit può convertire in modo affidabile in un flusso, come

  Stream, DoubleStream, LongStream, IntStream, Collection, Iterator, Itera

  bile, un array di oggetti o un array di primitive.

Test parametrizzati: @MethodSource

- Se hai bisogno di un solo parametro, puoi restituire un flusso di istanze del tipo di parametro

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull( actual: argument);
}

static Stream<String> stringProvider() {
    return Stream.of( ...values: "apple", "banana");
}
```

Metodo factory che genera il flusso di argomenti (stringa) per il test parametrizzato

Test parametrizzati: @MethodSource

- Flussi per tipi primitivi (DoubleStream, IntStream, e LongStream) sono supportati

```
@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
    assertNotEquals( unexpected: 9,  actual: argument);
}

static IntStream range() {
    return IntStream.range(0, 20).skip( n: 10);
}
```

Metodo factory che genera un flusso di Int

Test parametrizzati: @MethodSource

Se un metodo di test parametrizzato ha più parametri, è necessario restituire una raccolta, un flusso o un array di istanze di argomenti o array di oggetti

```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals( expected: 5,  actual: str.length());
    assertTrue( condition: num >=1 && num <=2);
    assertEquals( expected: 2,  actual: list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
            ...values: arguments( ...arguments: "apple", 1, Arrays.asList("a", "b")),
            arguments( ...arguments: "lemon", 2, Arrays.asList("x", "y"))
    );
}
```

arguments(Object…)
è un metodo Factory

Test parametrizzati: @CsvSource

Per esprimere elenchi di argomenti come valori separati da virgole (csv)

Ogni stringa fornita tramite l'attributo value in @CsvSource rappresenta un

Record CSV e risultati in una chiamata del test parametrizzato.

```
@ParameterizedTest
@CsvSource({
        "apple,         1",
        "banana,        2",
        "'lemon, lime', 0xF1",
        "strawberry,    700_000"
})
void testWithCsvSource1(String fruit, int rank) {
    assertNotNull( actual: fruit);
    assertNotEquals( unexpected: 0,  actual: rank);
}
```

Virgoletta singola (') come carattere di virgoletta

Test parametrizzati: @CsvSource

Per esprimere elenchi di argomenti come valori separati da virgole (csv)

Per impostazione predefinita, gli spazi iniziali e finali in una colonna CSV vengono tagliati.

Questo comportamento può essere modificato impostando

ignoreLeadingAndTrailingWhitespace = false

@CsvSource(valore = { " mela , banana" },

ignoreLeadingAndTrailingWhitespace = false)

Risultato:

“mela”, “banana”

Test parametrizzati: @CsvSource

Il primo record può essere utilizzato facoltativamente per fornire intestazioni CSV impostando l'attributo useHeadersInDisplayName = true

```
@ParameterizedTest(name = "[{index}] {arguments}")
@CsvSource(useHeadersInDisplayName = true, textBlock = """
FRUIT,          RANK
apple,          1
banana,         2
'lemon, lime', 0xF1
strawberry,     700_000
""")

void testWithCsvSource(String fruit, int rank) {
    assertNotNull( actual: fruit);
    assertNotEquals( unexpected: 0,  actual: rank);
}
```

Test parametrizzati: @CsvFileSource

Possiamo fornire come argomento per il nostro metodo di test file con valori separati da virgole (CSV) dal classpath o dal file system locale.

Ogni record da un file CSV risulta in una chiamata del parametro test.

È possibile ignorare le intestazioni tramite l'attributo numLinesToSkip.

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFileR(String country, int reference) {
    assertNotNull( actual: country);
    assertNotEquals( unexpected: 0,  actual: reference);
}
```

Test parametrizzati: @CsvFileSource

Possiamo fornire come argomento per il nostro metodo di test file con valori separati da virgole (CSV) dal classpath o dal file system locale.

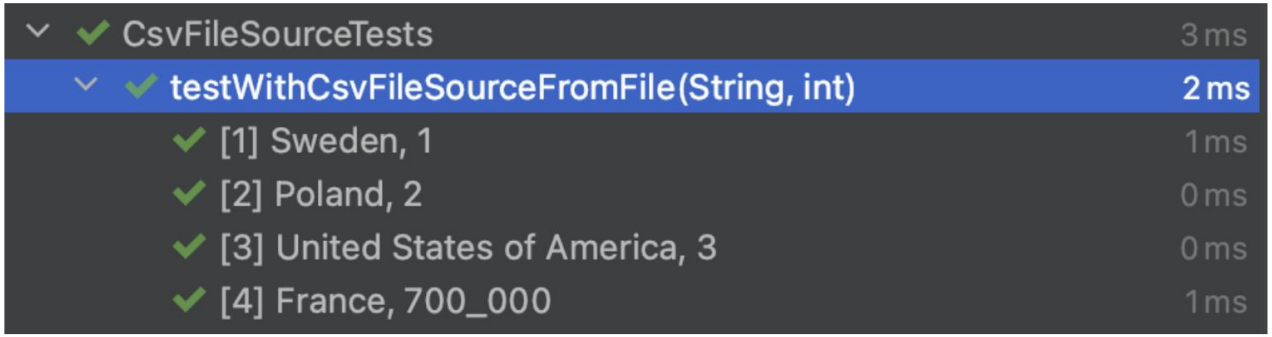
Ogni record da un file CSV risulta in una chiamata del parametro test.

È possibile ignorare le intestazioni tramite l'attributo numLinesToSkip.

```
@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country, int reference) {
    assertNotNull( actual: country);
    assertNotEquals( unexpected: 0,  actual: reference);
}
```

Test parametrizzati: @CsvFileSource

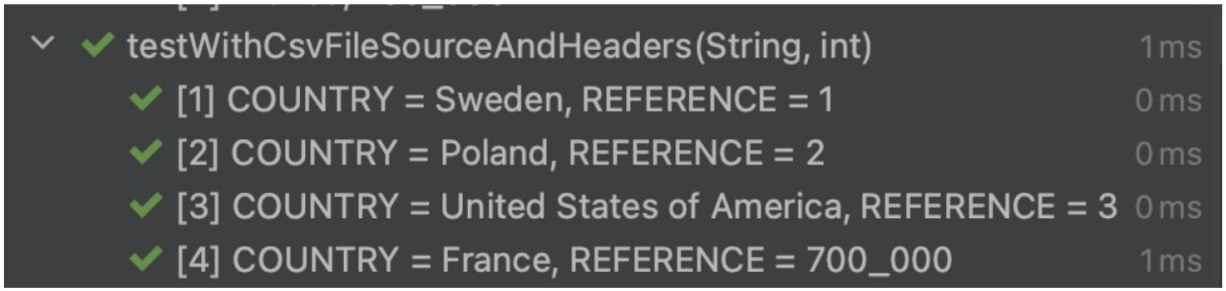
```java
@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country, int reference) {
    assertNotNull( actual: country);
    assertNotEquals( unexpected: 0,  actual: reference);
}
```



Test parametrizzati: @CsvFileSource

Se desideri che le intestazioni vengano utilizzate nei nomi visualizzati, puoi impostare l'attributo useHeadersInDisplayName su true

```
@ParameterizedTest(name = "[{index}] {arguments}")
@CsvFileSource(resources = "/two-column.csv", useHeadersInDisplayName = true)
void testWithCsvFileSourceAndHeaders(String country, int reference) {
    assertNotNull( actual: country);
    assertNotEquals( unexpected: 0,  actual: reference);
}
```



Esercizio: classificare i numeri

Verificare, tramite test parametrizzati JUnit 5, il comportamento di un **metodo** che classifica un **intero** come **"ZERO", "PARI"** o **"DISPARI".** I test devono coprire valori **positivi, negativi** e **casi limite.**

```
/**
  * Restituisce:
  * - "ZERO" se n == 0
  * - "PARI" se n è diverso da 0 ed è pari
  * - "DISPARI" se n è dispari */
```

Stringa statica pubblica classifyNumber(int n)

ÿ Usare **JUnit 5** e **Test Parametrizzati**

ÿ Coprire: ÿ

più valori per ciascuna categoria (zero, pari, dispari); ÿ numeri negativi; ÿ casi limite

ÿ Utilizzare almeno due diverse sorgenti di parametri (es. @CsvSource, @ValueSource o @MethodSource). ÿ

Dare un nome parlante ai test con @DisplayName

Esercizio: classificare i numeri

```java
package org.example;

public final class NumberUtils {

    private NumberUtils() {
        // utility class: costruttore privato
    }

    /**
     * Classifica un intero come ZERO, PARI o DISPARI.
     *
     * @param n intero in ingresso
     * @return "ZERO" se n == 0, "PARI" se n è pari (e ≠ 0), altrimenti "DISPARI"
     */
    public static String classifyNumber(int n) {
        if (n == 0) return "ZERO";
        return (n % 2 == 0) ? "PARI" : "DISPARI";
    }
}

```

Esercitazione: isMultipleof

Testare il metodo isMultipleOf(int n, int k):

- verificare che k sia un multiplo di n utilizzando i test parametrizzati con coppie (n,k).
- Inclusi valori di k negativi e zero (lanciare IllegalArgumentException se k == 0)

```java
/**
 * Restituisce true se n è un multiplo di k.
 * @param n intero
 * @param k divisore (non zero)
 * @return true se n % k == 0
 * @throws IllegalArgumentException se k == 0
 */
public static boolean isMultipleOf(int n, int k) {
    if (k == 0) {
        throw new IllegalArgumentException("k must be non-zero");
    }
    return n % k == 0;
}
```

Riferimenti

- Documentazione JUnit: https://junit.org/junit5/docs/current/user- guida/#scrittura-test-test-parametrizzati

- Metodo di interfaccia di annotazioneFonte:

  https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/MethodSource.html

- Tipo di annotazione CsvFileSource: https://junit.org/junit5/docs/5.7.2/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/CsvFileSource.html

- Interfaccia di annotazione CsvFileSource: https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/CsvFileSource.html

Azzurra Ragone

Dipartimento di Informatica - Piano VI - Stanza 616 Email: azzurra.ragone@uniba.it