UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

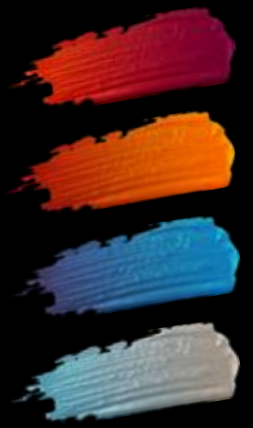
SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

JUnit: Parameterized Tests

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



What are Parameterized Tests?

Parameterized tests make it possible to run a test multiple times with different arguments.

Parameterized tests

Declared just like regular `@Test` methods but with the `@ParameterizedTest` annotation.

You must declare at least one source that will provide the arguments for each invocation and then consume the arguments in the test method.

Parameterized tests

Very simple example

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

Parameterized tests: Null and Empty Sources

Deal with **Null** and **Empty values** supplied to our parameterized tests

Check **corner cases** and verify proper behavior of our software when it is supplied ***bad*** input

- `@NullSource`: provides a **single null argument** to the annotated `@ParameterizedTest` method.
- `@EmptySource`: provides a **single empty argument** to the annotated `@ParameterizedTest` method for parameters of the following types: `java.lang.String`, `java.util.List`, `java.util.Set`, `java.util.Map`, primitive arrays (e.g., `int[]`, `char[][]`, etc.), object arrays (e.g., `String[]`, `Integer[][]`, etc.).

Parameterized tests: Null and Empty Sources

Deal with **Null** and **Empty values** supplied to our parameterized tests

Check **corner cases** and verify proper behavior of our software when it is supplied ***bad*** input

- `@NullSource`
- `@EmptySource`
- `@NullAndEmptySource`: a composed annotation that combines the functionality of `@NullSource` and `@EmptySource`.

Parameterized tests: Null and Empty Sources

Deal with **Null** and **Empty values** supplied to our parameterized tests

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

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

Parameterized tests: Null and Empty Sources

Deal with **Null** and **empty values** supplied to our parameterized tests

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

Parameterized tests: Null and Empty Sources

- `@NullSource`: provides a **single null argument** to the annotated `@ParameterizedTest` method.
- `@EmptySource`: provides a **single empty argument** to the annotated `@ParameterizedTest` method

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

How many times is the test method executed?

Parameterized tests: Null and Empty Sources

- `@NullAndEmptySource`: a composed annotation that combines the functionality of `@NullSource` and `@EmptySource`.

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

Parameterized tests: @MethodSource

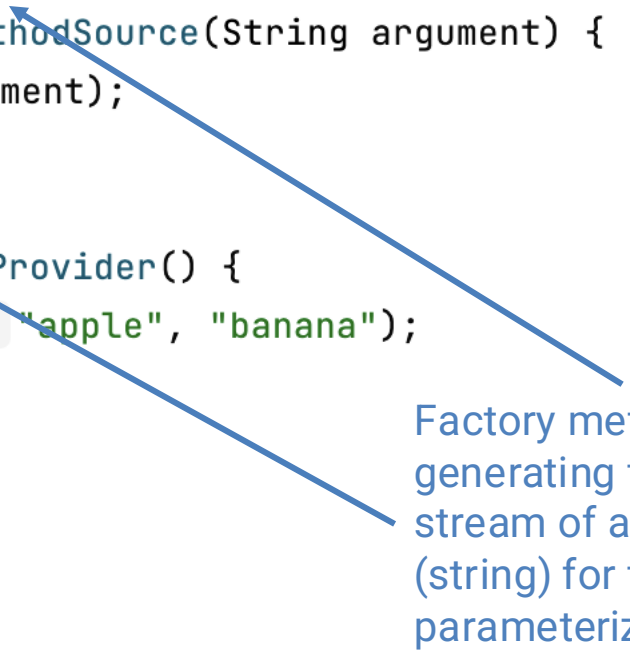
- `@MethodSource` allows you to refer to one or more factory methods
- Each **factory method** generates a **stream of arguments** (i.e., `Stream<Arguments>`)
- Each set of arguments within the stream will be the physical arguments for individual invocations of the annotated `@ParameterizedTest` method
- A "stream" is anything that JUnit can reliably convert into a Stream, such as *Stream, DoubleStream, LongStream, IntStream, Collection, Iterator, Iterable, an array of objects, or an array of primitives*.

Parameterized tests: @MethodSource

- If you only need a single parameter, you can return a Stream of instances of the parameter type

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

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



Factory method generating the stream of arguments (string) for the parameterized test

Parameterized tests: @MethodSource

- Streams for primitive types (`DoubleStream, IntStream, and LongStream`) are supported

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

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

Factory method generating a stream of Int

Parameterized tests: @MethodSource

If a parameterized test method has multiple parameters, you need to return a collection, stream, or array of Arguments instances or object arrays

```
@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…) is a Factory method

Parameterized tests: @CsvSource

To express argument lists as comma-separated values (csv)

Each string provided via the value attribute in `@CsvSource` represents a CSV record and results in one invocation of the parameterized test.

```
@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);
}
```

Single quote (') as its quote character

Parameterized tests: @CsvSource

To express argument lists as comma-separated values (csv)

Leading and trailing whitespace in a CSV column is trimmed by default.

This behavior can be changed by setting the

`ignoreLeadingAndTrailingWhitespace = false`

```
@CsvSource(value = { " apple , banana" },
ignoreLeadingAndTrailingWhitespace = false)
```

Result:

```
“ apple ”," banana”
```

Parameterized tests: @CsvSource

The first record may optionally be used to supply CSV headers by setting the `useHeadersInDisplayName attribute = 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);
}
```

Parameterized tests: @CsvFileSource

We can provide as argument for our test method comma-separated value (CSV) files from the classpath or the local file system.

Each record from a CSV file results in one invocation of the parameterized test.

It is possible to ignore the headers via the `numLinesToSkip` attribute.

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

Parameterized tests: @CsvFileSource

We can provide as argument for our test method comma-separated value (CSV) files from the classpath or the local file system.

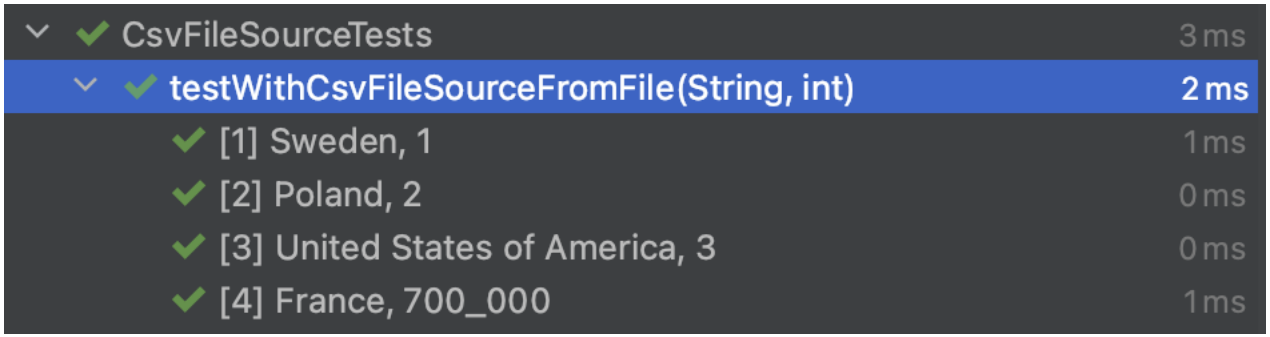
Each record from a CSV file results in one invocation of the parameterized test.

It is possible to ignore the headers via the `numLinesToSkip` attribute.

```
@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);
}
```

Parameterized tests: @CsvFileSource

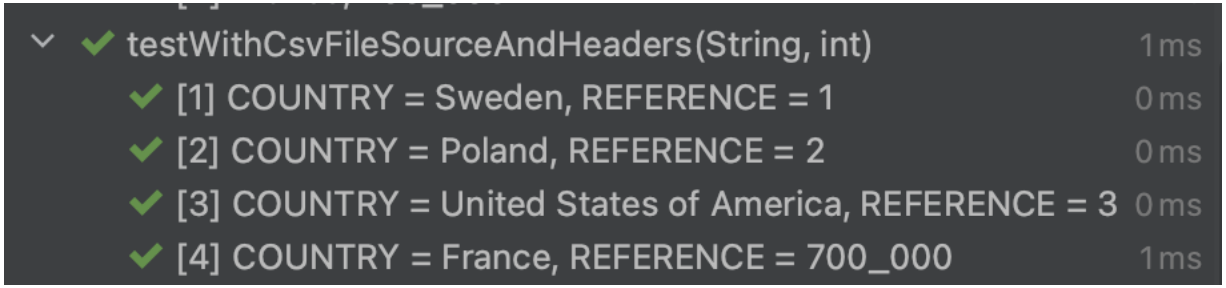
```java
@ParameterizedTest
@CsvFileSource(files = "src/test/resources/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSourceFromFile(String country, int reference) {
    assertNotNull( actual: country);
    assertNotEquals( unexpected: 0,  actual: reference);
}
```



Parameterized tests: @CsvFileSource

If you would like for the headers to be used in the display names, you can set the `useHeadersInDisplayName` attribute to 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);
}
```



Excercise: Classify numbers

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
 */
public static String classifyNumber(int n)
```

- Usare **JUnit 5** e **Parameterized Tests**
- Coprire:
  - più valori per ciascuna categoria (zero, pari, dispari);
  - numeri negativi;
  - casi limite
- Usare almeno due diverse sorgenti di parametri (es. @CsvSource, @ValueSource o @MethodSource).
- Dare un nome parlante ai test con @DisplayName

Excercise: Classify numbers

```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 usando i test parametrizzati con coppie (n,k).
- Includi 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;
}
```

References

- JUnit doc: https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests
- Annotation Interface MethodSource: https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/MethodSource.html
- Annotation Type CsvFileSource: https://junit.org/junit5/docs/5.7.2/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/CsvFileSource.html
- Annotation Interface CsvFileSource: https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/CsvFileSource.html

Azzurra Ragone

Department of Computer Science- Floor VI – Room 616
Email: azzurra.ragone@uniba.it