

I TESTMASTERS

Software Testing

Versione : 1.0

Data di rilascio:10/09/2024

Integrazione e Test di Sistemi Software

a.a. 2023-2024

ITPS - A

Realizzato da

Auciello Sonia 719769 s.auciello1@studenti.uniba.it

Croce Laura 717847 l.croce2@studenti.uniba.it

Dicataldo Luigi 718152 l.dicataldo2@studenti.uniba.it

Indice

1.	SPECIFICATION-BASED TESTING	5
1.1	UNDERSTANDING THE REQUIREMENT	5
1.1.1	<i>Goals.....</i>	5
1.1.2	<i>Parameters</i>	6
1.1.3	<i>Output.....</i>	6
1.2	EXPLORE WHAT THE PROGRAM DOES FOR VARIOUS INPUTS	6
1.3	EXPLORE INPUTS, OUTPUTS AND IDENTIFY PARTITIONS	7
1.3.1	<i>Individual inputs (classes of inputs)</i>	7
1.3.2	<i>Combinations of inputs</i>	8
1.3.3	<i>Classes of (expected) outputs.....</i>	9
1.4	IDENTIFY BOUNDARY CASES (AKA CORNER CASES)	9
1.5	DEVISE TEST CASES	10
1.6	AUTOMATE TEST CASES	13
1.7	ARGUMENT THE TEST SUITE WITH CREATIVITY AND EXPERIENCE	16
1.8	TEST CASES PLANNING AND EXECUTION	17
2.	STRUCTURAL TESTING AND CODE COVERAGE	19
2.1	RUN THE TEST SUITE WITH A CODE COVERAGE TOOL (TO IDENTIFY IN AN AUTOMATED WAY PARTS NOT COVERED)	19
2.2	MUTATION TESTING	20
3.	PROPERTY-BASED TESTING	25
3.1	UNDERSTANDING THE REQUIREMENTS	25
3.1.1	<i>Goals.....</i>	25
3.1.2	<i>Parameters</i>	26
3.1.3	<i>Output.....</i>	26
3.2	IDENTIFY PROPERTIES BASED ON THE REQUIREMENTS	27
3.3	TESTING THE FINDPRIMEINDEX METHOD	27
3.4	STATISTICS	31
3.4.1	<i>Occurrences of each divisor</i>	31
3.4.2	<i>Frequency of number with a specific quantity of divisors ..</i>	32
3.4.3	<i>Generation of prime and non-prime numbers</i>	34
3.4.4	<i>Probability of a prime number being generated a certain number of times</i>	35

4.	RESOURCES.....	37
4.1	REPOSITORY.....	37
4.2	GOOGLE SHEETS.....	37
4.3	TOOLS	37
5.	GLOSSARY.....	38
5.1	DEFINITIONS	38
5.1.1	<i>Capitalizzazione.....</i>	<i>38</i>

SPECIFICATION-BASED TESTING

HOMEWORK 1

1. SPECIFICATION-BASED TESTING

```
8 public String convertToCamelCase(String input, final boolean capitalizeFirstLetter, final char... delimiters) {
9     if (input == null || input.isEmpty()) {
10         return input;
11     }
12
13     input = input.toLowerCase();
14
15     final int inputLength = input.length();
16     final int[] resultCodePoints = new int[inputLength];
17     int outputOffset = 0;
18
19     final Set<Integer> delimiterSet = createDelimiterSet(delimiters);
20     boolean capitalizeNext = false;
21
22     if(capitalizeFirstLetter){
23         capitalizeNext = true;
24     }
25
26     for (int i = 0; i < inputLength; ) {
27         final int codePoint = input.codePointAt(i);
28
29         if (delimiterSet.contains(codePoint)) {
30             capitalizeNext = true;
31             if (outputOffset == 0) {
32                 capitalizeNext = false;
33             }
34             i += Character.charCount(codePoint);
35         } else if (capitalizeNext) {
36             final int titleCaseCodePoint = Character.titleCase(codePoint);
37             resultCodePoints[outputOffset++] = titleCaseCodePoint;
38             i += Character.charCount(titleCaseCodePoint);
39             capitalizeNext = false;
40         } else {
41             resultCodePoints[outputOffset++] = codePoint;
42             i += Character.charCount(codePoint);
43         }
44     }
45
46     if (outputOffset != 0){
47         return new String(resultCodePoints, 0, outputOffset);
48     }
49
50     return input;
51 }
```

1.1 Understanding the requirement

1.1.1 Goals

Il metodo *convertToCamelCase()* ha come scopo la trasformazione di una stringa generica nel formato camelCase.

Il camelCase è uno stile di scrittura in cui una frase o una serie di parole vengono unite senza spazi e ogni parola, eccetto la prima, inizia con una lettera maiuscola.



Questo formato è frequentemente utilizzato in programmazione per denominare variabili, funzioni e classi.

1.1.2 Parameters

I parametri necessari per il metodo *convertToCamelCase()* sono:

- **Input:** Rappresenta la stringa da convertire in camelCase. La stringa può essere anche *null*; in tal caso, il metodo restituisce direttamente la stringa senza modificarla
- **capitalizeFirstLetter:** È una variabile booleana che stabilisce se la prima lettera della stringa convertita debba essere maiuscola. Se è *true*, la prima lettera della stringa risultante sarà maiuscola (es. CamelCase), altrimenti sarà minuscola (es. camelCase)
- **delimiters:** È un insieme facoltativo di caratteri che indicano i punti nella stringa in cui applicare la capitalizzazione. Se questo parametro è *null* o vuoto, i delimitatori predefiniti sono considerati spazi vuoti.

1.1.3 Output

Il metodo restituisce:

- Una stringa in formato camelCase oppure *null* se la stringa in input è *null*.

1.2 Explore what the program does for various inputs

Sono stati eseguiti quattro semplici test case per verificare le funzionalità principali del metodo. In particolare, sono stati testati i seguenti scenari:

- *testSimpleCase()*: La stringa di input è semplice e composta da più parole separate da spazi.
- *testCapitalizeFirstLetter()*: La stringa di input è semplice, composta da più parole separate da spazi, e il parametro *capitalizeFirstLetter* è impostato su *true*.
- *testSimpleCaseWithoutDelimiters()*: La stringa di input non contiene più parole separate da spazi e non ci sono delimitatori specificati.
- *testSimpleCaseWithDelimiter()*: La stringa di input è composta da più parole, non contiene spazi ma viene specificato un delimitatore.

```

15  @Test
16  void testSimpleCase() {
17      input = "integrazione e test di sistemi software";
18      capitalizeFirstLetter = false;
19      assertEquals( expected: "integrazioneTestDiSistemiSoftware", caseUtils.convertToCamelCase(input, capitalizeFirstLetter));
20  }
21
22  @Test
23  void testCapitalizeFirstLetter() {
24      input = "integrazione e test di sistemi software";
25      capitalizeFirstLetter = true;
26      assertEquals( expected: "IntegrazioneTestDiSistemiSoftware", caseUtils.convertToCamelCase(input, capitalizeFirstLetter));
27  }
28
29  @Test
30  void testSimpleCaseWithoutDelimiters(){
31      input = "stringasenzaspazi";
32      capitalizeFirstLetter = false;
33      assertEquals( expected: "stringasenzaspazi", caseUtils.convertToCamelCase(input, capitalizeFirstLetter));
34  }
35
36  @Test
37  void testSimpleCaseWithDelimiter(){
38      input = "stringasenzaspazi";
39      capitalizeFirstLetter = false;
40      delimiters = new char[]{'a'};
41      assertEquals( expected: "stringSenzSpZi", caseUtils.convertToCamelCase(input, capitalizeFirstLetter, delimiters));
42  }

```

testSimpleCase()	passed	25 ms
testCapitalizeFirstLetter()	passed	1 ms
testSimpleCaseWithDelimiter()	passed	1 ms
testSimpleCaseWithoutDelimiters()	passed	1 ms

Generated by IntelliJ IDEA on 07/09/24, 18:29

1.3 Explore inputs, outputs and identify partitions

1.3.1 Individual inputs (classes of inputs)

input (stringa di input):

- *empty* (stringa vuota);
- *null* (stringa null);
- length 1 (stringa con un solo carattere);
- length > 1 (stringa con più caratteri);

capitalizeFirstLetter (capitalizzazione della prima lettera):

- *true* (la prima lettera della stringa convertita deve essere maiuscola);
- *false* (la prima lettera della stringa convertita deve rimanere minuscola);

delimiters:

- *null* (nessun delimitatore specificato, quindi i delimitatori predefiniti sono spazi vuoti);

- empty *char* array ({}) (array vuoto {}, quindi non ci sono delimitatori, si usano solo gli spazi);
- *char* array of length 1 (un array con un solo delimitatore);
- *char* array of length > 1 (un array con più delimitatori);

1.3.2 Combinations of inputs

Considerando i tre insiemi di input:

- *input* di 4 valori
- *capitalizeFirstLetter* di 2 valori
- *delimiters* di 4 valori

è possibile ottenere un massimo di 32 combinazioni:

T	input	<i>capitalizeFirstLetter</i>	<i>delimiters</i>
1	empty	false	<i>null</i>
2	empty	false	{ }
3	empty	true	<i>null</i>
4	empty	true	{ }
5	empty	false	1 delimiter
6	empty	true	1 delimiter
7	empty	false	> 1 delimiter
8	empty	true	> 1 delimiter
9	<i>null</i>	false	<i>null</i>
10	<i>null</i>	false	{ }
11	<i>null</i>	true	<i>null</i>
12	<i>null</i>	true	{ }
13	<i>null</i>	false	1 delimiter
14	<i>null</i>	true	1 delimiter
15	<i>null</i>	false	> 1 delimiter
16	<i>null</i>	true	> 1 delimiter
17	length 1	false	<i>null</i>





18	length 1	true	<i>null</i>
19	length 1	false	{}
20	length 1	true	{}
21	length 1	false	1 delimiter
22	length 1	true	1 delimiter
23	length 1	false	> 1 delimiter
24	length 1	true	> 1 delimiter
25	length > 1	false	<i>null</i>
26	length > 1	false	{}
27	length > 1	true	<i>null</i>
28	length > 1	true	{}
29	length > 1	false	1 delimiter
30	length > 1	true	1 delimiter
31	length > 1	false	> 1 delimiter
32	length > 1	true	> 1 delimiter

1.3.3 Classes of (expected) outputs

Le classi di output sono:

- *Null* string
- empty string
- string of length 1
- string of length > 1

1.4 Identify boundary cases (aka corner cases)

I casi limite sono situazioni particolari che testano il comportamento del metodo *convertToCamelCase ()* agli estremi delle sue condizioni di input. I boundary cases in questo caso sono:

- **Input (stringa di input)**
 - ON POINT: *Input* of length ≥ 1
 - OFF POINT: Empty or *Null* string

- ***capitalizeFirstLetter***
 - ON POINT: *capitalizeFirstLetter* is *false*
 - OFF POINT: *capitalizeFirstLetter* is *true*
- ***Delimiters***
 - ON POINT: Char array of length ≥ 1
 - OFF POINT: *null* or empty char

1.5 Devise test cases

In questa fase, l'obiettivo è selezionare i casi di test più rilevanti tra i 32 possibili identificati in [precedenza](#). La selezione dei test è basata sui seguenti criteri:

- Testare casi eccezionali come *null* ed empty, testandoli singolarmente senza combinarli
- Testare i casi che evidenziano singolarmente gli off point.

I test selezionati sono:

T	<i>input</i>	<i>capitalizeFirstLetter</i>	<i>delimiters</i>
5	empty	false	1 delimiter
13	<i>null</i>	false	1 delimiter
30	length > 1	true	1 delimiter
25	length > 1	false	<i>null</i>
26	length > 1	false	{ }

Questi test coprono:

- T5 e T13 individuano l'off point del parametro *input* (stringa vuota e null)
- T30 individua l'off point del parametro *capitalizeFirstLetter*
- T25 e T26 individuano l'off point del parametro *delimiters*

Per garantire una copertura completa, la suite di test viene ampliata con i seguenti casi aggiuntivi:

T	<i>input</i>	<i>capitalizeFirstLetter</i>	<i>delimiters</i>
21	length 1	false	1 delimiter
23	length 1	false	> 1 delimiters



20	length 1	true	}
31	length > 1	false	> 1 delimiter

- **T21:** Stringa di lunghezza 1, *capitalizeFirstLetter* = *false*, un delimitatore. Questo test esplora il comportamento del metodo con una stringa di singolo carattere e un delimitatore.
 - **T21.1:** Il delimitatore è diverso dalla stringa
 - **T21.2:** Il delimitatore è uguale alla stringa
- **T23:** Stringa di lunghezza 1, *capitalizeFirstLetter* = *false*, più delimitatore. Questo test esamina diversi scenari con una stringa di un carattere e diversi delimitatori.
 - **T23.1:** Tutti i delimitatori sono diversi dalla stringa
 - **T23.2:** I delimitatori hanno un carattere uguale alla stringa e uno o più diversi
 - **T23.3:** Tutti i delimitatori sono uguali alla stringa
- **T20:** Stringa di lunghezza 1, *capitalizeFirstLetter* = *true*, delimitatori vuoti {}. Questo test verifica il comportamento del metodo quando si rende maiuscolo l'unico carattere della stringa.
- **T31:** Stringa di lunghezza > 1, *capitalizeFirstLetter* = *false*, più delimitatori. Questo test esplora un caso di utilizzo del metodo *convertToCamelCase()*, in cui la stringa ha più caratteri e più delimitatori.

Dopo aver riorganizzato i test per una maggiore chiarezza, la test suite risulta essere composta dai seguenti test cases:

T	str	capitalizeFirstLetter	delimiters
1	empty	false	1 delimiter
2	null	false	1 delimiter
3	length > 1	true	1 delimiter
4	length > 1	false	null
5	length > 1	false	}
6	length 1	false	1 delimiter (delimitatore diverso dalla stringa)

7	length 1	false	1 delimiter (delimitatore uguale alla stringa)
8	length 1	false	> 1 delimiters (delimitatori tutti diversi dalla stringa)
9	length 1	false	> 1 delimiters (un delimitatore uguale alla stringa e uno o più diversi)
10	length 1	false	> 1 delimiters (tutti i delimitatori uguali alla stringa)
11	length 1	true	}
12	length > 1	false	> 1 delimiter

1.6 Automate test cases

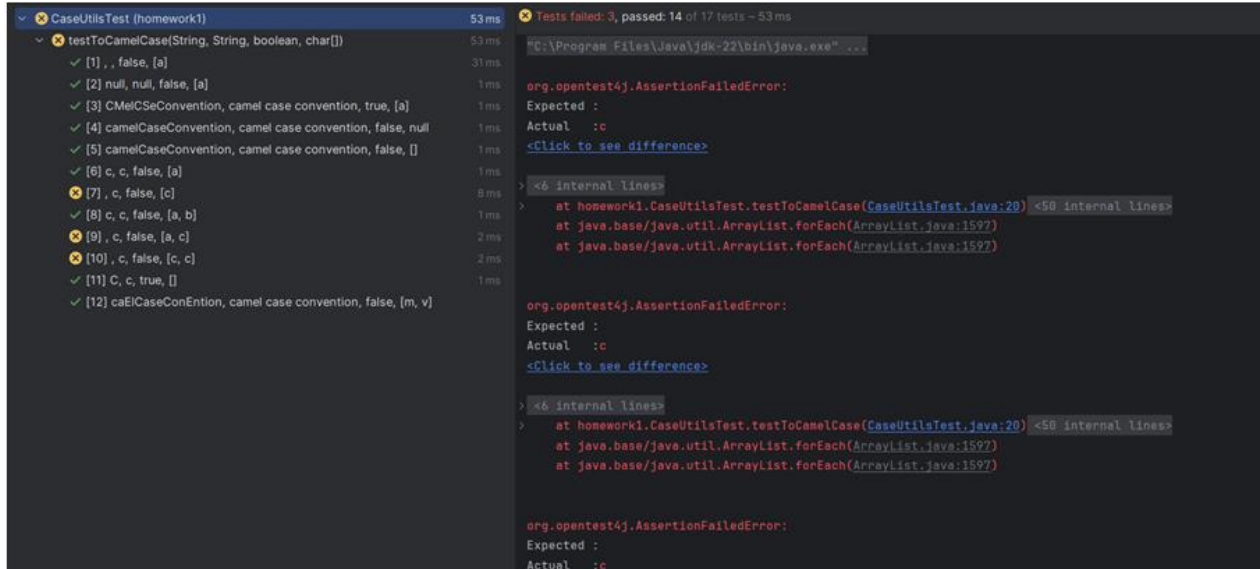
Nel corso della fase di implementazione sono stati impiegati i test parametrici, in quanto consentono di analizzare diversi scenari, variando i parametri specificati.

```

17  @ParameterizedTest
18  @MethodSource("toCamelCaseArgs")
19  void testToCamelCase(String expected, String str, final boolean capitalizeFirstLetter, final char... delimiters) {
20      assertEquals(expected, caseUtils.toCamelCase(str, capitalizeFirstLetter, delimiters));
21  }
22
23  1 usage  Vito Marco Rubino *
24  @ static Stream<Arguments> toCamelCaseArgs() {
25      return Stream.of(
26          // T1: empty string, capitalizeFirstLetter = false, 1 delimiter
27          Arguments.of( ...arguments: "", "", false, new char[]{'a'}),
28          // T2: null string, capitalizeFirstLetter = false, 1 delimiter
29          Arguments.of( ...arguments: null, null, false, new char[]{'a'}),
30          // T3: string length > 1, capitalizeFirstLetter = true, 1 delimiter
31          Arguments.of( ...arguments: "CMelCSeConvention", "camel case convention", true, new char[]{'a'}),
32          // T4: string length > 1, capitalizeFirstLetter = false, null delimiters array
33          Arguments.of( ...arguments: "camelCaseConvention", "camel case convention", false, null),
34          // T5: string length > 1, capitalizeFirstLetter = false, empty delimiters array
35          Arguments.of( ...arguments: "camelCaseConvention", "camel case convention", false, new char[]{}),
36
37          // T6: string length 1, capitalizeFirstLetter = false, 1 delimiter
38          // (Delimiter is not present in the string)
39          Arguments.of( ...arguments: "c", "c", false, new char[]{'a'}),
40          // T7: string length 1, capitalizeFirstLetter = false, 1 delimiter
41          // (Delimiter is present in the string)
42          Arguments.of( ...arguments: "", "c", false, new char[]{'c'}), // failed
43
44          // T8: string length 1, capitalizeFirstLetter = false, > 1 delimiters
45          // (Delimiters are not present in the string)
46          Arguments.of( ...arguments: "c", "c", false, new char[]{'a', 'b'}),
47          // T9: string length 1, capitalizeFirstLetter = false, > 1 delimiters
48          // (One delimiter is present in the string and the others are not)
49          Arguments.of( ...arguments: "", "c", false, new char[]{'a', 'c'}), // failed
50          // T10: string length 1, capitalizeFirstLetter = false, > 1 delimiters
51          // (All delimiters are present in the string)
52          Arguments.of( ...arguments: "", "c", false, new char[]{'c', 'c'}), // failed
53
54          // T11: string length 1, capitalizeFirstLetter = true, empty delimiters array
55          Arguments.of( ...arguments: "C", "c", true, new char[]{}),
56
57          // T12: string length > 1, capitalizeFirstLetter = false, > 1 delimiters
58          Arguments.of( ...arguments: "caElCaseConEntion", "camel case convention", false, new char[]{'m', 'v'}),
59      );
60  }

```

Di seguito viene mostrato uno screenshot dell'output dei test case:



```

CaseUtilsTest (homework1) 53ms
  testToCamelCase(String, String, boolean, char[]) 53ms
    [1] ., false, [a] 31ms
    [2] null, null, false, [a] 1ms
    [3] CMelCSeConvention, camel case convention, true, [a] 1ms
    [4] camelCaseConvention, camel case convention, false, null 1ms
    [5] camelCaseConvention, camel case convention, false, [] 1ms
    [6] c, c, false, [a] 1ms
    [7] c, c, false, [c] 8ms
    [8] c, c, false, [a, b] 1ms
    [9] c, c, false, [a, c] 2ms
    [10] c, c, false, [c, c] 2ms
    [11] C, c, true, [] 1ms
    [12] caElCaseConEntion, camel case convention, false, [m, v] 1ms

Tests failed: 3, passed: 14 of 17 tests - 53 ms
"C:\Program Files\Java\jdk-22\bin\java.exe" ...

org.opentest4j.AssertionFailedError:
Expected :
Actual   :c
<Click to see difference>

> <6 internal lines>
> at homework1.CaseUtilsTest.testToCamelCase(CaseUtilsTest.java:20) <50 internal lines>
> at java.base/java.util.ArrayList.forEach(ArrayList.java:1597)
> at java.base/java.util.ArrayList.forEach(ArrayList.java:1597)

org.opentest4j.AssertionFailedError:
Expected :
Actual   :c
<Click to see difference>

> <6 internal lines>
> at homework1.CaseUtilsTest.testToCamelCase(CaseUtilsTest.java:20) <50 internal lines>
> at java.base/java.util.ArrayList.forEach(ArrayList.java:1597)
> at java.base/java.util.ArrayList.forEach(ArrayList.java:1597)

org.opentest4j.AssertionFailedError:
Expected :
Actual   :c
  
```

Come si può osservare dallo screenshot, i test case 7, 9 e 10 non sono andati a buon fine, poiché il valore restituito non coincide con quello previsto. In particolare, il metodo dovrebbe restituire una stringa vuota quando la stringa di partenza è costituita da un solo carattere e tale carattere si trova tra i delimitatori. Questo comportamento errato è stato identificato come un bug, che abbiamo corretto modificando il codice sorgente.



```

32 public String convertToCamelCase(String input, final boolean capitalizeFirstLetter, final char... delimiters) {
33     if (input == null || input.isEmpty()) {
34         return input;
35     }
36
37     input = input.toLowerCase();
38
39     final int inputLength = input.length();
40     final int[] resultCodePoints = new int[inputLength];
41     int outputOffset = 0;
42
43     final Set<Integer> delimiterSet = createDelimiterSet(delimiters);
44     boolean capitalizeNext = false;
45
46     if(capitalizeFirstLetter){
47         capitalizeNext = true;
48     }
49
50     for (int i = 0; i < inputLength; ) {
51         final int codePoint = input.codePointAt(i);
52
53         if (delimiterSet.contains(codePoint)) {
54             capitalizeNext = true;
55             if (outputOffset == 0) {
56                 capitalizeNext = false;
57             }
58             i += Character.charCount(codePoint);
59         } else if (capitalizeNext) {
60             final int titleCaseCodePoint = Character.toTitleCase(codePoint);
61             resultCodePoints[outputOffset++] = titleCaseCodePoint;
62             i += Character.charCount(titleCaseCodePoint);
63             capitalizeNext = false;
64         } else {
65             resultCodePoints[outputOffset++] = codePoint;
66             i += Character.charCount(codePoint);
67         }
68     }
69
70     if (outputOffset != 0){
71         return new String(resultCodePoints, 0, outputOffset);
72     }
73
74     return "";
75 }

```

Pertanto, modificando il codice in modo da dichiarare '*return input*' in '*return ""*', il metodo restituirà una stringa vuota quando la stringa di input contiene gli stessi caratteri presenti nei delimitatori. Grazie a questa modifica, tutti i test case sono stati superati con successo.

✓ CaseUtilsTest (homework1)	53 ms	✓ Tests passed: 12 of 12 tests – 53 ms
✓ testToCamelCase(String, String, boolean, char[])	53 ms	"C:\Program Files\Java\jdk-22\bin\java.exe" ...
✓ [1] , , false, [a]	42 ms	Process finished with exit code 0
✓ [2] null, null, false, [a]	1 ms	
✓ [3] CMeICSeConvention, camel case convention, true, [a]	1 ms	
✓ [4] camelCaseConvention, camel case convention, false, null	1 ms	
✓ [5] camelCaseConvention, camel case convention, false, []	1 ms	
✓ [6] c, c, false, [a]	1 ms	
✓ [7] , c, false, [c]	1 ms	
✓ [8] c, c, false, [a, b]	1 ms	
✓ [9] , c, false, [a, c]	1 ms	
✓ [10] , c, false, [c, c]	1 ms	
✓ [11] C, c, true, []	1 ms	
✓ [12] caElCaseConEntion, camel case convention, false, [m, v]	1 ms	

1.7 Argument the test suite with creativity and experience

In quest'ultima fase del workflow dello specification-based testing, abbiamo arricchito la test suite con ulteriori 5 test case per analizzare scenari particolari. Nello specifico, sono stati presi in esame i seguenti casi:

- **Presenza di caratteri speciali o lettere accentate nella stringa di input.** Questo scenario è stato sviluppato in due test case:
 - Stringa di input con una lettera accentata all'inizio della seconda parola e senza delimitatori.
 - Stringa di input contenente una lettera accentata e un delimitatore che include la stessa lettera accentata.
- **Presenza di numeri nella stringa di input.** Come nel caso precedente, questo scenario è stato suddiviso in due test case:
 - Stringa di input contenente un numero senza delimitatori
 - Stringa di input contenente un numero con un delimitatore corrispondente allo stesso numero
- **Stringa di input già formattata secondo la conversione camelCase.** L'output atteso è una stringa in cui, ad eccezione della prima parola, le altre lettere maiuscole vengono trasformate in minuscole, poiché, formalmente, fanno parte di un'unica parola.

T	<i>str</i>	<i>capitalizeFirstLetter</i>	<i>delimiters</i>
13	camelCase formatted	false	<i>null</i>
14	contains special characters	false	<i>null</i>
15	contains special characters	false	1 delimiter (special character contained in the input string)
16	contains numbers	false	<i>null</i>
17	contains number	false	1 delimiter (number contained in the input string)


```

16 class CaseUtilsTest {
17     CaseUtils caseUtils = new CaseUtils(); 1 usage
18
19     @ParameterizedTest
20     @MethodSource("toCamelCaseArgs")
21     void testToCamelCase(String expected, String str, final boolean capitalizeFirstLetter, final char... delimiters) {
22         assertEquals(expected, caseUtils.convertToCamelCase(str, capitalizeFirstLetter, delimiters));
23     }
24
25     @
26     static Stream<Arguments> toCamelCaseArgs() { 1 usage
27         return Stream.of(
28             // CREATIVITY TEST CASES
29             // Input string is already camel case formatted,
30             Arguments.of(...arguments: "camelcaseconvention", "camelCaseConvention", false, null),
31             // Input string contains special characters
32             Arguments.of(...arguments: "uneÉcole", "une école", false, null),
33             Arguments.of(...arguments: "uneCole", "une école", false, new char[]{'é'}),
34             // Input string contains numbers
35             Arguments.of(...arguments: "junit5TestingFramework", "JUnit5 Testing Framework", false, null),
36             Arguments.of(...arguments: "junitTestingFramework", "JUnit5 Testing Framework", false, new char[]{'5'})
37         );
38     }
39 }

```

L'esecuzione dei 5 test case aggiuntivi è stata completata con successo.

Di seguito è riportato uno screenshot che mostra i risultati dell'intera test suite

✓ CaseUtilsTest (homework1)	42 ms	✓ Tests passed: 17 of 17 tests – 42 ms
✓ testToCamelCase(String, String, boolean, char[])	42 ms	"C:\Program Files\Java\jdk-22\bin\java.exe" ...
✓ [1] , , false, [a]	26 ms	Process finished with exit code 0
✓ [2] null, null, false, [a]	1 ms	
✓ [3] CMelCSseConvention, camel case convention, true, [a]	1 ms	
✓ [4] camelCaseConvention, camel case convention, false, null	1 ms	
✓ [5] camelCaseConvention, camel case convention, false, []	1 ms	
✓ [6] c, c, false, [a]	1 ms	
✓ [7] , c, false, [c]	1 ms	
✓ [8] c, c, false, [a, b]	1 ms	
✓ [9] , c, false, [a, c]	1 ms	
✓ [10] , c, false, [c, c]	1 ms	
✓ [11] C, c, true, []	1 ms	
✓ [12] caElCaseConEntion, camel case convention, false, [m, v]	1 ms	
✓ [13] camelcaseconvention, camelCaseConvention, false, null	1 ms	
✓ [14] uneÉcole, une école, false, null	1 ms	
✓ [15] uneCole, une école, false, [é]	1 ms	
✓ [16] junit5TestingFramework, JUnit5 Testing Framework, false, null	1 ms	
✓ [17] junitTestingFramework, JUnit5 Testing Framework, false, [5]	1 ms	

1.8 Test cases planning and execution

I test case relativi a questo homework sono stati organizzati all'interno di un foglio di pianificazione ed esecuzione, che ha permesso di monitorare l'intero processo di testing e di tracciare i risultati ottenuti. [Test Cases](#)



STRUCTURAL TESTING AND CODE COVERAGE




HOMEWORK 1

2. STRUCTURAL TESTING AND CODE COVERAGE

2.1 Run the test suite with a code coverage tool (to identify in an automated way parts not covered)






homework > homework1

homework1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main		0%	n/a		2	2	9	9	2	2	1	1
CaseUtils		100%		100%	0	14	0	38	0	3	0	1
Total	28 of 164	82%	0 of 22	100%	2	16	9	47	2	5	1	2

homework > homework1 > CaseUtils

CaseUtils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
convertToCamelCase(String, boolean, char[])		100%		100%	0	9	0	29	0	1
createDelimiterSet(char[])		100%		100%	0	4	0	7	0	1
CaseUtils()		100%	n/a		0	1	0	2	0	1
Total	0 of 136	100%	0 of 22	100%	0	14	0	38	0	3

CaseUtils.java

```

1. package homework1;
2.
3. import java.util.HashSet;
4. import java.util.Set;
5.
6. public class CaseUtils {
7.
8.     /**
9.      * Converts delimiter-separated words into camelCase, with each word starting
10.     * with an uppercase character and the rest in lowercase.
11.     *
12.     * Delimiters separate the words, and the first character after a delimiter is capitalized.
13.     * The first letter may or may not be capitalized based on the capitalizeFirstLetter flag.
14.     *
15.     * Returns {@code null} if the input is {@code null} and an empty string ("" ) if only delimiters are present.
16.     * Capitalization uses Unicode title case, equivalent to standard upper case.
17.     *
18.     * @param input the string to convert, may be {@code null}
19.     * @param capitalizeFirstLetter if true, the first letter is capitalized
20.     * @param delimiters characters separating the words, {@code null} or empty array means whitespace
21.     * @return camelCase string, or {@code null} if the input is null
22.     */
23.     public String convertToCamelCase(String input, final boolean capitalizeFirstLetter, final char... delimiters) {
24.         if (input == null || input.isEmpty()) {
25.             return input;
26.         }
27.
28.         input = input.toLowerCase();
29.
30.         final int inputLength = input.length();
31.         final int[] resultCodePoints = new int[inputLength];
32.         int outputOffset = 0;
33.
34.         final Set<Integer> delimiterSet = createDelimiterSet(delimiters);
35.         boolean capitalizeNext = false;
36.
37.         if (capitalizeFirstLetter) {
38.             capitalizeNext = true;
39.         }
40.
41.         for (int i = 0; i < inputLength; ) {
42.             final int codePoint = input.codePointAt(i);
43.
44.             if (delimiterSet.contains(codePoint)) {
45.                 capitalizeNext = true;
46.                 if (outputOffset == 0) {
47.                     capitalizeNext = false;
48.                 }

```



```

49.         i += Character.charCount(codePoint);
50.     } else if (capitalizeNext) {
51.         final int titleCaseCodePoint = Character.toTitleCase(codePoint);
52.         resultCodePoints[outputOffset++] = titleCaseCodePoint;
53.         i += Character.charCount(titleCaseCodePoint);
54.         capitalizeNext = false;
55.     } else {
56.         resultCodePoints[outputOffset++] = codePoint;
57.         i += Character.charCount(codePoint);
58.     }
59. }
60.
61.     if (outputOffset != 0) {
62.         return new String(resultCodePoints, 0, outputOffset);
63.     }
64.
65.     return "";
66. }
67.
68. /**
69.  * Converts an array of delimiters to a hash set of code points. Code point of space(32) is added
70.  * as the default value. The generated hash set provides O(1) lookup time.
71.  *
72.  * @param delimiters set of characters to determine capitalization, null means whitespace
73.  * @return Set
74.  */
75.     private static Set<Integer> createDelimiterSet(final char[] delimiters) {
76.
77.         Set<Integer> delimiterHashSet = new HashSet<Integer>();
78.         delimiterHashSet.add(Character.codePointAt(new char[]{' '}, 0));
79.         if (delimiters == null || delimiters.length == 0) {
80.             return delimiterHashSet;
81.         }
82.
83.         for (int index = 0; index < delimiters.length; index++) {
84.             delimiterHashSet.add(Character.codePointAt(delimiters, index));
85.         }
86.
87.         return delimiterHashSet;
88.     }
89.
90. /**
91.  * {@code CaseUtils} instances should not be constructed directly;
92.  * the class should be used statically.
93.  *
94.  * The constructor is public to support tools requiring a JavaBean instance.
95.  */
96.     public CaseUtils() {
97.
98.     }
99. }

```

2.2 Mutation testing

Il Mutation Testing è una tecnica di testing che prevede l'introduzione intenzionale di errori nel codice da verificare, con lo scopo di valutare se la suite di test è sufficientemente efficace nell'individuareli. Questa strategia si basa sul principio del *Coupling Effect*, il quale afferma che un errore complesso è solitamente il risultato di una combinazione di errori più semplici. Di conseguenza, se la suite di test è in grado di rilevare questi piccoli errori, sarà capace anche di individuare quelli più complessi.

Per eseguire il Mutation Testing è stato impiegato [PIT Mutation Testing](#), che ha generato mutanti secondo 22 diverse categorie:

- **NEGATE_CONDITIONALS** : Questo mutatore altera tutte le condizioni logiche trovate nel codice, ad esempio sostituendo l'operatore "==" con "!=".
- **EMPTY_RETURNS**: Sostituisce il valore di ritorno di una funzione con un valore vuoto, come ad esempio una stringa vuota.
- **CONDITIONALS_BOUNDARY**: Modifica un operatore relazionale (come "<") con il suo contrario (come ">").
- **INCREMENTS**: Trasforma le operazioni di incremento di una variabile in decrementi e viceversa.

Pit Test Coverage Report

Package Summary

homework1

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
1	100% <div>38/38</div>	100% <div>22/22</div>	100% <div>22/22</div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
CaseUtils.java	100% <div>38/38</div>	100% <div>22/22</div>	100% <div>22/22</div>

Report generated by [PIT](#) 1.15.8

Come evidenziato dal report prodotto dallo strumento, la suite di test sviluppata è riuscita a individuare tutte le mutazioni introdotte nel codice da verificare. Di conseguenza, sebbene non sia possibile avere una test suite completamente esaustiva, si può affermare che quella realizzata possieda un elevato grado di robustezza.

Di seguito viene riportato il dettaglio delle mutazioni effettuate, con l'indicazione della riga in cui è stato inserito un errore e la tipologia di mutazione applicata.





CaseUtils.java

```

1  package homework1;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  public class CaseUtils {
7
8      /**
9       * Converts delimiter-separated words into camelCase, with each word starting
10      * with an uppercase character and the rest in lowercase.
11      *
12      * Delimiters separate the words, and the first character after a delimiter is capitalized.
13      * The first letter may or may not be capitalized based on the capitalizeFirstLetter flag.
14      *
15      * Returns {@code null} if the input is {@code null} and an empty string ("") if only delimiters are present.
16      * Capitalization uses Unicode title case, equivalent to standard upper case.
17      *
18      * @param input the string to convert, may be {@code null}
19      * @param capitalizeFirstLetter if true, the first letter is capitalized
20      * @param delimiters characters separating the words, {@code null} or empty array means whitespace
21      * @return camelCase string, or {@code null} if the input is null
22      */
23     public String convertToCamelCase(String input, final boolean capitalizeFirstLetter, final char... delimiters) {
24         if (input == null || input.isEmpty()) {
25             return input;
26         }
27
28         input = input.toLowerCase();
29
30         final int inputLength = input.length();
31         final int[] resultCodePoints = new int[inputLength];
32         int outputOffset = 0;
33
34         final Set<Integer> delimiterSet = createDelimiterSet(delimiters);
35         boolean capitalizeNext = false;
36
37         if (capitalizeFirstLetter) {
38             capitalizeNext = true;
39         }
40
41         for (int i = 0; i < inputLength; ) {
42             final int codePoint = input.codePointAt(i);
43
44             if (delimiterSet.contains(codePoint)) {
45                 capitalizeNext = true;
46                 if (outputOffset == 0) {
47                     capitalizeNext = false;
48                 }
49                 i += Character.charCount(codePoint);
50             } else if (capitalizeNext) {
51                 final int titleCaseCodePoint = Character.toTitleCase(codePoint);
52                 resultCodePoints[outputOffset++] = titleCaseCodePoint;
53                 i += Character.charCount(titleCaseCodePoint);
54                 capitalizeNext = false;
55             } else {
56                 resultCodePoints[outputOffset++] = codePoint;
57                 i += Character.charCount(codePoint);
58             }
59         }
60
61         if (outputOffset != 0) {
62             return new String(resultCodePoints, 0, outputOffset);
63         }
64
65         return "";
66     }
67
68     /**
69      * Converts an array of delimiters to a hash set of code points. Code point of space(32) is added
70      * as the default value. The generated hash set provides O(1) lookup time.
71      *
72      * @param delimiters set of characters to determine capitalization, null means whitespace
73      * @return Set
74      */
75     private static Set<Integer> createDelimiterSet(final char[] delimiters) {
76
77         Set<Integer> delimiterHashSet = new HashSet<Integer>();
78         delimiterHashSet.add(Character.codePointAt(new char[]{' '}, 0));
79         if (delimiters == null || delimiters.length == 0) {
80             return delimiterHashSet;
81         }

```

```

81     }
82
83     for (int index = 0; index < delimiters.length; index++) {
84         delimiterHashSet.add(Character.codePointAt(delimiters, index));
85     }
86
87     return delimiterHashSet;
88 }
89
90 /**
91  * {@code CaseUtils} instances should not be constructed directly;
92  * the class should be used statically.
93  *
94  * The constructor is public to support tools requiring a JavaBean instance.
95  */
96 public CaseUtils() {
97
98 }
99 }

```

Mutations

```

24 1. negated conditional → KILLED
25 2. negated conditional → KILLED
26 1. replaced return value with "" for homework1/CaseUtils::convertToCamelCase → KILLED
27 1. negated conditional → KILLED
41 1. changed conditional boundary → KILLED
42 2. negated conditional → KILLED
44 1. negated conditional → KILLED
46 1. negated conditional → KILLED
49 1. Replaced integer addition with subtraction → KILLED
50 1. negated conditional → KILLED
52 1. Changed increment from 1 to -1 → KILLED
53 1. Replaced integer addition with subtraction → KILLED
56 1. Changed increment from 1 to -1 → KILLED
57 1. Replaced integer addition with subtraction → KILLED
61 1. negated conditional → KILLED
62 1. replaced return value with "" for homework1/CaseUtils::convertToCamelCase → KILLED
79 1. negated conditional → KILLED
80 2. negated conditional → KILLED
81 1. replaced return value with Collections.emptySet for homework1/CaseUtils::createDelimiterSet → KILLED
83 1. changed conditional boundary → KILLED
84 2. negated conditional → KILLED
87 1. replaced return value with Collections.emptySet for homework1/CaseUtils::createDelimiterSet → KILLED

```

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

```

• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#5] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#16] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#14] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#4] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#11] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#17] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#15] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#3] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#12] (0 ms)
• homework1.ExplorationStepTest [engine:junit-jupiter] [class:homework1.ExplorationStepTest] [method:testCapitalizeFirstLetter()] (1 ms)
• homework1.ExplorationStepTest [engine:junit-jupiter] [class:homework1.ExplorationStepTest] [method:testSimpleCase()] (2 ms)
• homework1.ExplorationStepTest [engine:junit-jupiter] [class:homework1.ExplorationStepTest] [method:testSimpleCaseWithDelimiter()] (2 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#13] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#6] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#8] (0 ms)
• homework1.ExplorationStepTest [engine:junit-jupiter] [class:homework1.ExplorationStepTest] [method:testSimpleCaseWithoutDelimiters()] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#7] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#10] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#9] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#2] (0 ms)
• homework1.CaseUtilsTest [engine:junit-jupiter] [class:homework1.CaseUtilsTest] [test-template:testToCamelCase(java.lang.String, java.lang.String, boolean, %$BC)] [test-template-invocation:#1] (31 ms)

```

Report generated by PIT 1.15.8

PROPERTY-BASED TESTING

HOMEWORK 2

3. PROPERTY-BASED TESTING

```
11 public int findPrimeIndex(List<Integer> numbers, boolean findFirst) {
12     if (numbers == null || numbers.isEmpty()) {
13         throw new IllegalArgumentException("Invalid input"); // Invalid input
14     }
15
16     // Check if the list contains only non-positive numbers
17     if (numbers.stream().allMatch(num -> num <= 0)) {
18         throw new IllegalArgumentException("The list does not contain any positive numbers");
19     }
20
21     int index = -1;
22
23     if (findFirst) {
24         for (int i = 0; i < numbers.size(); i++) {
25             int number = numbers.get(i);
26             boolean isPrime = number > 1;
27             for (int j = 2; isPrime && j <= Math.sqrt(number); j++) {
28                 if (number % j == 0) {
29                     isPrime = false;
30                 }
31             }
32             if (isPrime) {
33                 return i; // return the index immediately when we find the first prime number
34             }
35         }
36     } else {
37         for (int i = numbers.size() - 1; i >= 0; i--) {
38             int number = numbers.get(i);
39             boolean isPrime = number > 1;
40             for (int j = 2; isPrime && j <= Math.sqrt(number); j++) {
41                 if (number % j == 0) {
42                     isPrime = false;
43                 }
44             }
45             if (isPrime) {
46                 return i; // return the index immediately when we find the last prime number
47             }
48         }
49     }
50
51     return index;
52 }
```

3.1 Understanding the requirements

3.1.1 Goals

Il metodo *findPrimeIndex()* ha l'obiettivo di restituire l'indice della prima o dell'ultima occorrenza di un numero primo all'interno di una lista di numeri interi, in base al valore del parametro *findFirst*. Se *findFirst* è impostato su



true, verrà restituito l'indice del primo *numero primo* trovato. Altrimenti, verrà restituito l'indice dell'ultimo *numero primo* presente nella lista.

Ad esempio, se:

- *numbers* = [4, 2, 6, 8, 3]
- *findFirst* = *true*

L'output atteso sarà 1, ovvero l'indice del primo numero primo che appare nella lista di interi (in questo caso 2).

Diversamente:

- *numbers* = [4, 2, 6, 8, 3]
- *findFirst* = *false*

L'output atteso sarà 4, che rappresenta l'indice del numero primo che compare per ultimo nella lista di interi (in questo caso 3).

3.1.2 Parameters

I parametri richiesti dal metodo *findPrimeIndex()* sono i seguenti:

- *numbers*: è la lista di numeri interi nella quale si vuole individuare la prima o l'ultima occorrenza di numero primo.
- *findFirst*: è una variabile booleana che determina se restituire l'indice della prima occorrenza di un numero primo (*true*), o l'indice dell'ultima occorrenza di un numero primo (*false*) all'interno della lista

3.1.3 Output

Il metodo restituisce:

- *int*: l'indice, se trovato, della prima o dell'ultima occorrenza di un numero primo all'interno della lista di numeri interi. Altrimenti, se non viene trovata alcuna occorrenza, il metodo restituisce -1.
- *IllegalArgumentException*: viene lanciata l'eccezione se i parametri passati non sono validi, ovvero nei seguenti casi:
 - *null numbers* list
 - *empty numbers* list
 - La lista contiene solo numeri interi negative, che non possono essere primi.

3.2 Identify properties based on the requirements

Per il metodo *findPrimeIndex()*, possiamo identificare 3 possibili scenari di output:

1. **Fail:** questo scenario si verifica quando nella lista di numeri interi non è presente alcun numero primo. L'output atteso in questi casi è -1.
 - a. *numbers* = [4, 6, 8, 10, 12]
 - b. *findFirst* = *true*
 - c. Output atteso: -1
2. **Passed:** questo scenario si verifica quando nella lista c'è almeno un numero primo. L'output atteso sarà un intero maggiore o uguale a 0, che corrisponde all'indice della prima o dell'ultima occorrenza del numero primo, a seconda del valore di *findFirst*.
 - a. *numbers* = [4, 2, 6, 8, 3]
 - b. *findFirst* = *true*
 - c. Output atteso: 1, l'indice del primo numero primo trovato (2).
3. **Invalid:** Questo scenario si verifica quando almeno uno degli input è non valido. In questi casi, viene lanciata un'eccezione (*IllegalArgumentException*).
Gli input non validi includono:
 - a. *Null numbers* list
 - b. *Empty numbers* list
 - c. Lista che contiene solo numeri negativi

3.3 Testing the findPrimeIndex method

Sulla base degli scenari definiti in [precedenza](#), i test case sono i seguenti:

- **Fail.:** Per questo scenario abbiamo scelto di utilizzare i seguenti parametri di input:
 - ***numbers*:** una lista di numeri interi di dimensione variabile tra 1 e 100, con valori generati nell'intervallo tra 4604 e 4620.
 - ***findFirst*:** una variabile booleana che indica se cercare la prima o l'ultima occorrenza di un numero primo nella lista di interi.

Per la lista di numeri interi, abbiamo optato per una dimensione variabile tra 1 e 100, per garantire che le liste siano sufficientemente ampie e non vuote, rispettando così la preconditione del metodo. I valori saranno compresi tra

4604 e 4620, che include solo numeri non primi. Pertanto, l'output previsto sarà -1.

```
29  @Property(generation = GenerationMode.RANDOMIZED) no usages
30  @Report(Reporting.GENERATED)
31  @StatisticsReport(format = Histogram.class)
32  void testFail(@ForAll @Size(min = 1, max = 100) List<@IntRange(min = 4604, max = 4620) Integer> numbers,
33               @ForAll boolean findFirst) {
34
35      int expected, actual;
36
37      expected = -1;
38      actual = primeIndex.findPrimeIndex(numbers, findFirst);
39
40      assertEquals(expected, actual);
41  }
```

- **Passed.:** Per questa partizione abbiamo deciso di usare i seguenti parametri:
 - ***numbers***: una lista di numeri interi con dimensione variabile tra 20 e 50, con valori generati tra 4604 e 4620.
 - ***findFirst***: una variabile booleana che indica se cercare la prima o l'ultima occorrenza di un numero primo nella lista di interi.
 - ***indexesToAddPrimes***: una lista di 3 numeri interi distinti che rappresentano le posizioni in cui inserire 3 numeri primi nella lista *numbers*.
 - ***primesToAdd***: una lista di 3 numeri primi generati tramite il metodo arbitrario *primes()*

```

91     @Provide no usages
92     Arbitrary<List<Integer>> primes() {
93         return Arbitraries.randomValue(this::generatePrime).list();
94     }
95
96
97     @ private Integer generatePrime(Random random) { 1 usage
98         int candidate;
99         do {
100             candidate = random.nextInt( bound: 10000) + 2;
101             Statistics.collect(isPrime(candidate) ? "prime" : "nonPrime");
102         } while (!isPrime(candidate));
103         Statistics.label("Prime number").collect(candidate);
104         return candidate;
105     }
106
107     private boolean isPrime(int number) { 2 usages
108         if (number <= 1) {
109             return false;
110         }
111         for (int j = 2; j <= Math.sqrt(number); j++) {
112             if (number % j == 0) {
113                 return false;
114             }
115         }
116         return true;
117     }
118 }

```

La lista di numeri interi sarà creata con una dimensione variabile tra 20 e 50, garantendo che le liste non siano mai vuote e rispettino la preconditione del metodo. I valori generati saranno compresi tra 4604 e 4620, che contiene solo numeri non primi. Poiché il test mira a trovare almeno un numero primo nella lista *numbers*, sono stati generati 3 numeri primi per essere inseriti nelle posizioni specificate dalla lista di indici *indexesToAddPrimes*. Gli indici sono stati ordinati in modo crescente in modo che:

- Se *findFirst* è impostato come *true*, cercando il primo numero primo della lista, l'output atteso sarà il primo indice generato in *indexesToAddPrimes*
- Se *findFirst* è impostato come *false*, cercando l'ultimo numero primo della lista, l'output atteso sarà l'ultimo indice generato in *indexesToAddPrimes*



```

43  @Property(generation = GenerationMode.RANDOMIZED) no usages
44  @Report(Reporting.GENERATED)
45  @StatisticsReport(format = Histogram.class)
46  void testPass(@ForAll @Size(min = 20, max = 50) List<@IntRange(min = 4604, max = 4620) Integer> numbers,
47               @ForAll boolean findFirst,
48               @ForAll @Size(value = 3) @UniqueElements List<@IntRange(max = 19) Integer> indexesToAddPrimes,
49               @ForAll("primes") @Size(value = 3) List<Integer> primeToAdd) {
50
51      int expected, actual;
52
53      Collections.sort(indexesToAddPrimes); // Sort the list of indexes to add primes in ascending order
54
55      int i, j;
56      for (i = 0; i < indexesToAddPrimes.size(); i++) {
57          numbers.add(indexesToAddPrimes.get(i), primeToAdd.get(i)); // Add primes to the list at the specified indexes
58      }
59
60      if (findFirst) {
61          // If we want to find the first prime number, the expected index is the first index in the list
62          expected = indexesToAddPrimes.get(0);
63      } else {
64          // If we want to find the last prime number, the expected index is the last index in the list
65          expected = indexesToAddPrimes.get(indexesToAddPrimes.size() - 1);
66      }
67
68      actual = primeIndex.findPrimeIndex(numbers, findFirst);
69
70      assertEquals(expected, actual);
71  }

```

Invalid : Per questa partizione abbiamo scelto i seguenti parametri:

- **numbers**: una lista di numeri interi di dimensione variabile tra 0 e 100, con valori compresi tra -200 e -100.
- **findFirst**: una variabile booleana che indica se cercare la prima o l'ultima occorrenza di un numero primo nella lista di interi.

Questo test ha l'obiettivo di verificare che il metodo da testare generi un'eccezione *IllegalArgumentException* quando i parametri forniti non sono validi. Il limite inferiore della dimensione della lista di interi include 0 per testare anche il caso di una lista vuota. Per le liste non vuote, i valori saranno compresi tra -200 e -100, per violare la preconditione del metodo, ovvero che tutti i numeri nella lista di interi siano positivi.

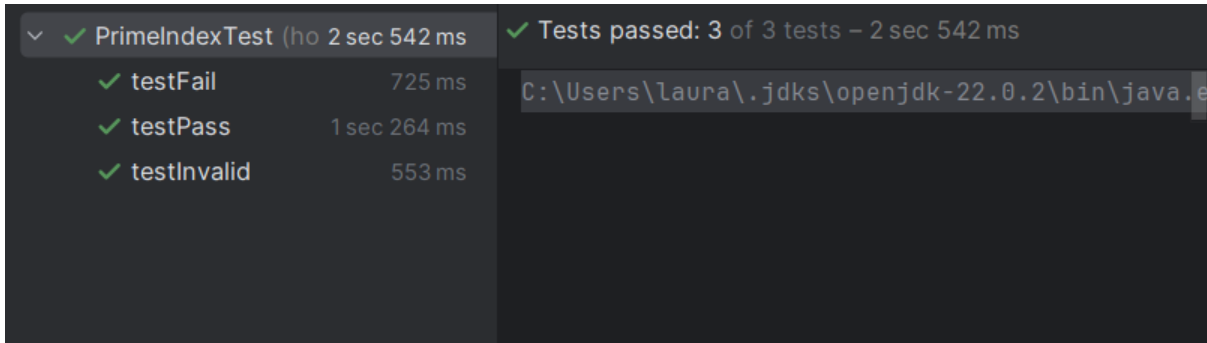
```

20  @Property(generation = GenerationMode.RANDOMIZED)
21  @Report(Reporting.GENERATED)
22  @StatisticsReport(format = Histogram.class)
23  void testInvalid(@ForAll @Size(max = 100) List<@IntRange(min = -200, max = -100) Integer> numbers,
24                 @ForAll boolean findFirst) {
25
26      assertThrows(IllegalArgumentException.class, () -> primeIndex.findPrimeIndex(numbers, findFirst));
27  }

```



I test per le tre partizioni sono andati tutti a buon fine



```

✓ PrimeIndexTest (ho 2 sec 542 ms)
  ✓ testFail      725 ms
  ✓ testPass     1 sec 264 ms
  ✓ testInvalid   553 ms
✓ Tests passed: 3 of 3 tests – 2 sec 542 ms
C:\Users\laura\.jdk\openjdk-22.0.2\bin\java.exe
  
```

3.4 Statistics

3.4.1 Occurrences of each divisor




```

91      @Provide no usages
92      Arbitrary<List<Integer>> primes() {
93          return Arbitraries.randomValue(this::generatePrime).list();
94      }
95
96
97  @      private Integer generatePrime(Random random) { 1 usage
98          int candidate;
99          do {
100              candidate = random.nextInt( bound: 10000) + 2;
101              Statistics.collect(isPrime(candidate) ? "prime" : "nonPrime");
102          } while (!isPrime(candidate));
103          Statistics.label("Prime number").collect(candidate);
104          return candidate;
105      }
  
```

La prima statistica esamina i divisori di ogni numero in un campione di test, escludendo 1 e il numero stesso, poiché il divisore 1 è una proprietà implicita di ogni intero positivo.

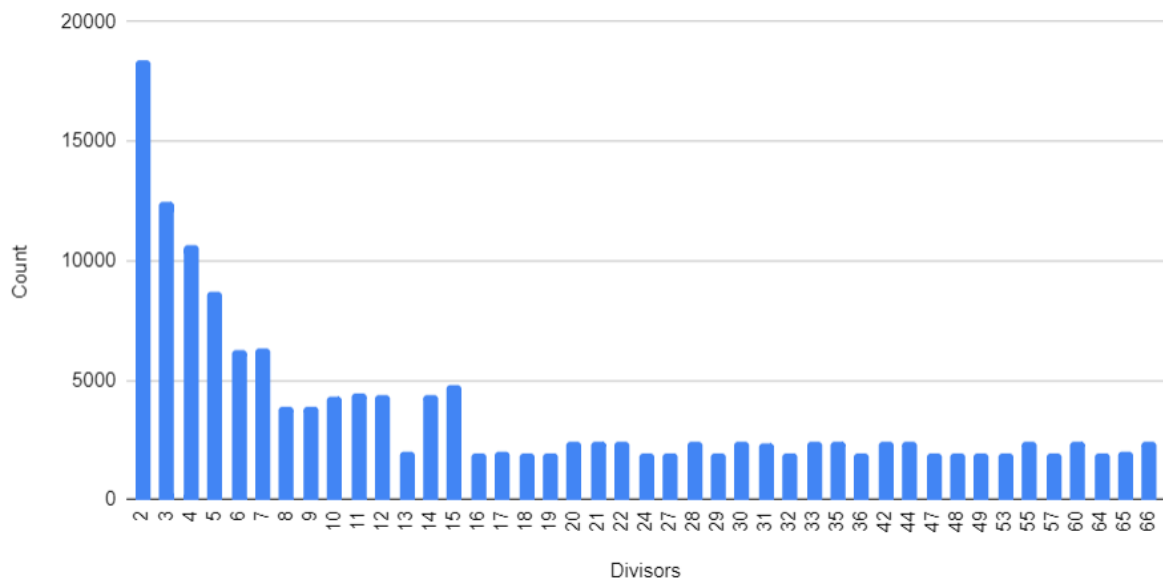
L'obiettivo di questa analisi è mostrare quanti numeri del campione possono essere divisi da un determinato divisore.

Ad esempio, se viene generato il numero 4605, i suoi divisori saranno {3,5,15,307,921,1535}. Di conseguenza, le occorrenze di ciascun divisore verranno incrementate. I dati raccontati per questa statistica sono riportati nel primo foglio del file.  [Statistic](#)



Per facilitare l'interpretazione, i dati sono stati rappresentati tramite un istogramma. In questo istogramma, sull'asse delle x sono riportati i divisori trovati per ciascun numero e sull'asse delle y si conta il numero delle occorrenze del divisore

Statistics data - Occurrences of each divisor



Dall'analisi del grafico, si nota una chiara asimmetria verso sinistra, che indica una prevalenza di divisori di dimensioni ridotte. Questo risultato è atteso, poiché è statisticamente più probabile che i numeri più piccoli abbiano più divisori. Inoltre, si osserva che il divisore più frequente è il 2, il che è in linea con le aspettative, poiché ogni numero pari è divisibile per 2. Infine, il valore massimo registrato per i divisori è 66, che corrisponde alla radice quadrata dei numeri analizzati. Questo è coerente con le aspettative, poiché la radice quadrata rappresenta il limite massimo teorico per un divisore di un dato numero. Ad esempio, generando valori compresi tra 4604 e 4620, la radice quadrata di questi numeri si avvicina a 66.

3.4.2 Frequency of number with a specific quantity of divisors

La seconda statistica analizza quanti elementi nel campione test possiedono un numero specifico di divisore, includendo sia 1 che il numero stesso.

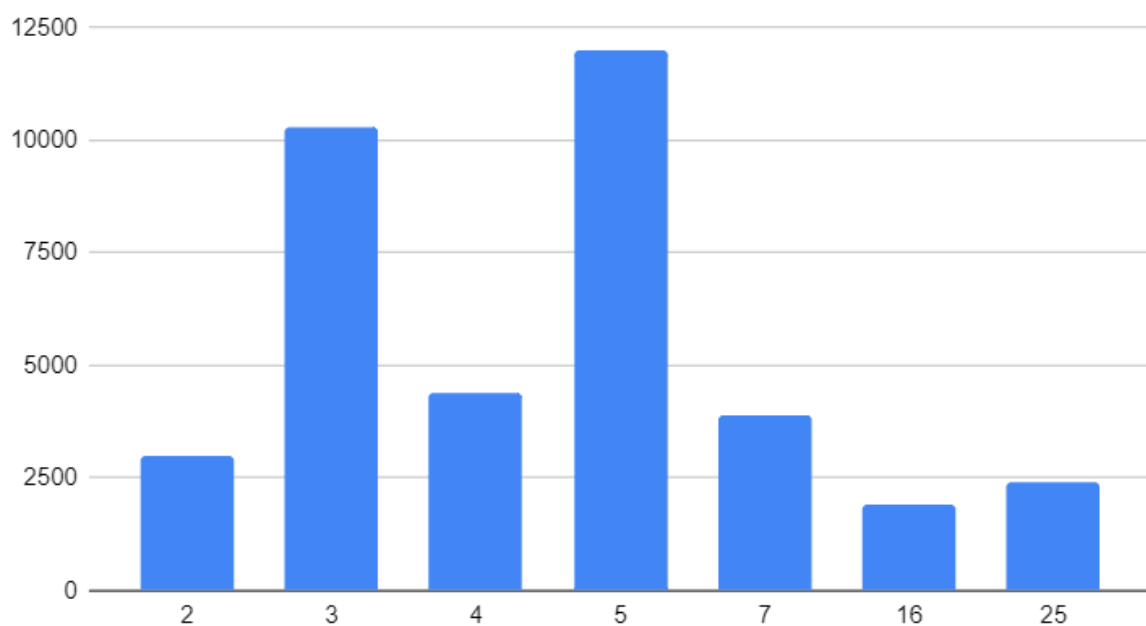
Ad esempio, per il numero 64, i divisori sono $\{1, 2, 4, 8, 16, 32, 64\}$, per un totale di 7 divisori. Di conseguenza, viene incrementato il conteggio dei numeri che hanno esattamente 7 divisori.

I dati raccolti sono i seguenti e possono essere consultati anche se secondo foglio del file. [Statistic](#)

Number of divisors	count
2	3000
3	10271
4	4395
5	12010
7	3882
16	1922
25	2420

Per facilitare l'interpretazione, i dati sono stati rappresentati tramite un istogramma. Sull'asse delle x è indicato il numero di divisori, mentre sull'asse delle y è riportato il numero degli elementi nel campione di test che hanno una specifica quantità di divisori.

Quantity of divisors



A differenza della prima statistica, questa volta il grafico mostra una distribuzione più equilibrata. Si osservano chiaramente 3000 numeri con

solo 2 divisori, il che riflette la presenza dei 3 numeri primi inclusi nei 1000 test effettuati con il metodo *testPass()*, il cui scopo è identificare almeno un numero primo all'interno di una lista di interi.

Inoltre, il grafico evidenzia due picchi distinti. Il primo picco rivela un andamento interessante: la maggior parte dei numeri testati ha esattamente 5 divisori. Il secondo picco indica che un altro gruppo di circa 10000 numeri presenta esattamente 3 divisori.

3.4.3 Generation of prime and non-prime numbers

Attualmente non esiste un criterio noto per determinare l'intervallo tra i numeri primi. A eccezione della coppia formata da 2 e 3, l'intervallo deve essere un numero pari maggiore o uguale a 2, poiché tra due numeri consecutivi almeno uno è pari e quindi non è primo. Di conseguenza, non è possibile creare un algoritmo che generi esclusivamente numeri primi. Le opzioni disponibili sono quindi:

1. Fornire una lista di numeri primi già selezionati
2. Generare casualmente dei numeri all'interno di un intervallo e restituire solo quello primi

Per i test effettuati, è stata scelta la seconda opzione, implementata nel metodo *@Provide*.

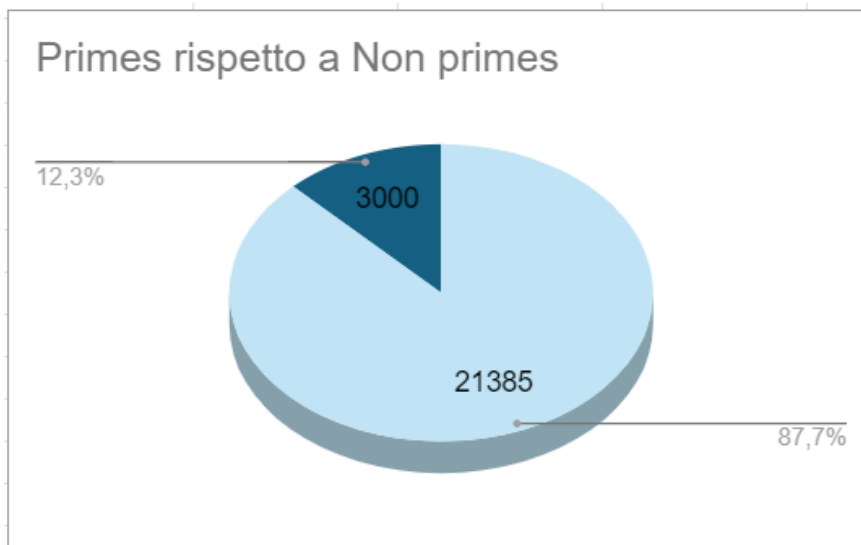
```
91      @Provide no usages
92      Arbitrary<List<Integer>> primes() {
93          return Arbitraries.randomValue(this::generatePrime).list();
94      }
95
96
97  @
98      private Integer generatePrime(Random random) { 1 usage
99          int candidate;
100          do {
101              candidate = random.nextInt(bound: 10000) + 2;
102              Statistics.collect(isPrime(candidate) ? "prime" : "nonPrime");
103          } while (!isPrime(candidate));
104          Statistics.label("Prime number").collect(candidate);
105          return candidate;
106      }
```

La terza statistica ha analizzato la generazione di 3000 numeri primi, valutando la proporzione di numeri non primi generati durante il processo.

I dati rilevati sono i seguenti e si possono trovare anche all'interno del terzo foglio del file [Statistic](#)

Non primes	Primes
21385	3000

I risultati mostrano che, al fine di ottenere 3000 numeri primi, sono stati generati oltre 21000 numeri non primi.



Dunque, si può affermare che il metodo utilizzato nella generazione dei numeri primi mostra un tasso di successo di circa il 12%.

3.4.4 Probability of a prime number being generated a certain number of times

L'ultima statistica esamina la probabilità che un numero primo venga estratto un certo numero di volte.

```

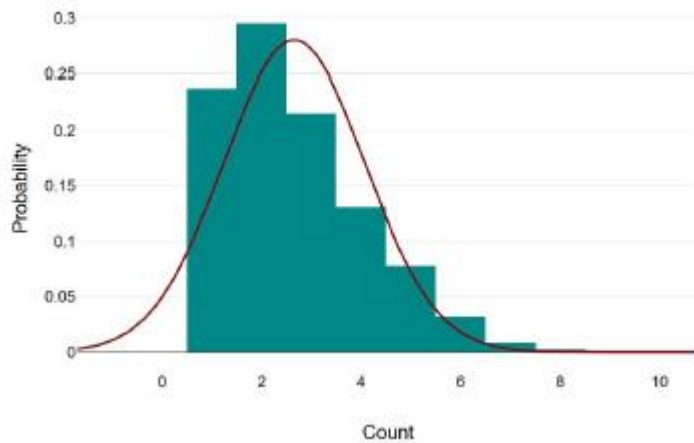
91      @Provide no usages
92      Arbitrary<List<Integer>> primes() {
93          return Arbitraries.randomValue(this::generatePrime).list();
94      }
95
96
97  @
98      private Integer generatePrime(Random random) { 1 usage
99          int candidate;
100         do {
101             candidate = random.nextInt(bound: 10000) + 2;
102             Statistics.collect(isPrime(candidate) ? "prime" : "nonPrime");
103         } while (!isPrime(candidate));
104         Statistics.label("Prime number").collect(candidate);
105         return candidate;
106     }
  
```

Per elaborare questa statistica, il primo passo è stato contare le occorrenze di ciascun numero primo generato.

I dati grezzi per questa analisi sono raccolti nell'ultimo foglio del file.

Successivamente, sono stati calcolati la media e la deviazione standard del campione per ottenere i parametri necessari a visualizzare la funzione di distribuzione. [Statistic](#)

Average	Standard deviation
2,68	1,43



4. RESOURCES

4.1 Repository

Le classi testate e i relativi metodi di test sono disponibili all'interno della seguente repository GitHub:

 <https://github.com/LauraCroce/TestMasters>

4.2 Google Sheets

- Test cases planning and execution.

 [Test Cases](#)

- Dati generati da jqwik e successivamente elaborati con le statistiche

 [Statistics](#)

4.3 Tools

Per l'esecuzione della Code Coverage è stato utilizzato JaCoCo

 <https://www.eclemma.org/jacoco/>

Per l'esecuzione della Code Coverage è stato utilizzato PIT Mutation Testing

 <https://pitest.org/>

Per l'esecuzione del Property-based Testing è stato utilizzato jqwik

 <https://jqwik.net/>

Per la visualizzazione dei dati e l'elaborazione dei grafici per il Property-based Testing è stato utilizzato Google Sheets

 <https://workspace.google.com/intl/it/products/sheets/>

5. GLOSSARY

5.1 Definitions

5.1.1 Capitalizzazione

Per **CAPITALIZZAZIONE** si intende trasformare la prima lettera di una stringa in una lettera maiuscola, mentre le lettere restanti sono in minuscolo