Relazione Progetto Integrazione e Test di Sistemi Software - Corso ITSS A.A. 2023/2024

Studenti:

Michele Scagliola
Christian Vitucci

Gruppo:

NarrowVerse

Sommario

Home Work 1

1.0 – Introduzione

In questo progetto sono stati presi in considerazione due metodi appartenenti alla classe StringUtils, facente parte della libreria Apache Commons realizzata dall'omonima fondazione. Essa è scritta in linguaggio Java, e permette di usufruire di metodi sulle più disparate operazioni su stringhe, numeri e altri tipi di dato, che sono versioni espanse di quanto consentito dai metodi standard di Java, e a volte inediti. Come si può intuire i metodi presi in considerazione fanno parte di una classe che si occupa di operazioni su stringhe, quali troncamento, comparazione, conteggio di caratteri, o ancora metodi di utility come effettuare la verifica sul se un carattere è numerico o meno, e molti altri. In dettaglio i metodi scelti sono:

1.1 - substring

"subString" consente di estrarre dal parametro di tipo *String* passato, una nuova stringa, porzione dell'originale ottenuta specificando l'indice di inizio e fine desiderati, entrambi i parametri di tipo *int;*

```java
public static String substring(final String str, int start, int end) {
    if (str == null) {
        return null;
    }

    // handle negatives
    if (end < 0) {
        end = str.length() + end; // remember end is negative
    }
    if (start < 0) {
        start = str.length() + start; // remember start is negative
    }

    // check length next
    if (end > str.length()) {
        end = str.length();
    }

    // if start is greater than end, return ""
    if (start > end) {
        return "";
    }

    if (start < 0) {
        start = 0;
    }
    if (end < 0) {
        end = 0;
    }

    return str.substring(start, end);
}
```

1.2 - swapCase

"swapCase" consente di portare i caratteri in minuscolo di una stringa, in maiuscolo e quelli in maiuscolo in minuscolo, ottenendo quindi una formattazione invertita del parametro *String* di partenza, funzionalità mancante in Java, che permette solo di portare tutti i caratteri in blocco in minuscolo o maiuscolo.

```java
public static String swapCase(final String str) throws IllegalArgumentException {
    if (str == null) {
        throw new IllegalArgumentException();
    }

    if (Objects.equals(str, b: "")) {
        return str;
    }

    final int strLen = str.length();
    final int[] newCodePoints = new int[strLen]; // cannot be longer than the char array
    int outOffset = 0;
    for (int i = 0; i < strLen; ) {
        final int oldCodepoint = str.codePointAt(i);
        final int newCodePoint;
        if (Character.isUpperCase(oldCodepoint) || Character.isTitleCase(oldCodepoint)) {
            newCodePoint = Character.toLowerCase(oldCodepoint);
        } else if (Character.isLowerCase(oldCodepoint)) {
            newCodePoint = Character.toUpperCase(oldCodepoint);
        } else {
            newCodePoint = oldCodepoint;
        }
        newCodePoints[outOffset++] = newCodePoint;
        i += Character.charCount(newCodePoint);
    }
    return new String(newCodePoints, offset: 0, outOffset);
}
```

2.0 – Ideazione e Automazione dei test cases

2.1 substring

2.1.1 Analisi dei requisiti

Lo scopo di questa funzione è quello di generare una stringa di dimensione minore o uguale a quella passatagli in input, parametro **str** di tipo *String*, sulla base di due parametri di tipo *int* anch'essi passati alla funzione come input, cioè **start** e **end**. Questi ultimi indicano rispettivamente il range da cui partire e dove fermarsi nell'estrazione di una sottostringa di **str**, che sarà l'output generato, anch'esso di tipo *String* che veda **start** come prima lettera ed **end** come ultima.

2.1.1 Esplorazione di input, output e definizione partizioni

Analizzando il codice della funzione, sono emerse le seguenti partizioni per:

- il parametro str:
  - null
  - stringa di lunghezza zero, pertanto vuota
  - stringa di lunghezza maggiore di zero

- il parametro start:
  - positivo
  - negativo
  - maggiore di end
  - minore di end
  - uguale a end

- il parametro end:
  - positivo
  - negativo
  - maggiore di start
  - minore di start
  - uguale a start

L'output sarà un oggetto di tipo *String,* di lunghezza minore o uguale a quella del parametro di input **str** che rispetta i caratteri di controllo, laddove presenti.

2.1.3 Casi limite

Da un'analisi del codice non risultano esserci casi limite che non consentono l'esecuzione del codice in sé e il gran numero di controlli presenti nello stesso consente una copertura di casistiche decisamente superiori rispetto al *substring* standard di Java in cui invece l'acquisizione di uno start negativo genera un caso limite, in quanto valore non utilizzabile, mentre nel codice in esame si è dinanzi a 2 opzioni a seconda dell'entità della stringa e dei limiti (start e end) e cioè l'aggiunta della lunghezza di **str** allo **start**, oppure la sostituzione del valore negativo con l'**int** 0.

2.1.4 Ideazione dei test cases

Poiché la funzione *substring* punta a generare stringhe figlie di quella passatagli in input, un ruolo importante non è svolto unicamente dalla tipologia di stringa passatagli, bensì dai vincoli che rappresentano i delineatori di inizio e fine che la sottostringa dovrà rispettare. In quest'ottica vengono fuori i test cases iniziali:

- T1: input str **null**, con start e end ininfluenti
- T2: input str vuoto, con start e end ininfluenti
- T3: input str composto da uno spazio, start ed end nei limiti
- T4: start negativo di dimensioni contenute, indipendentemente da start e str
- T5: start negativo di dimensioni elevate, indipendentemente da end e str
- T6: end negativo di dimensioni contenute, indipendentemente da start e str
- T7: end negativo di dimensioni elevate, indipendentemente da end e str
- T8: start ed end uguali, str ininfluente
- T9: start ed end negativi, str ininfluente
- T10: start maggiore dellla lunghezza di str
- T11: end maggiore della lunghezza di str
- T12: start uguale alla lunghezza della stringa, indipendentemente da str ed end
- T13: stringa con caratteri di controllo
- T14: str con caratteri Unicode, end e str nei limiti

2.1.5 Automazione dei test

Per automatizzare i test, si è optato per l'uso della libreria JUnit 5.10.0, si è fatto uso della funzione *assertEquals* e assert*Null*. Il codice risultante è il seguente:

- T1: utilizzando *null* come parametro in input per str e dei valori qualsiasi per start ed end, il metodo *substring* verifica che il risultato sia nullo. Il test conferma questo comportamento.

```
@Test
void substringWithNullInput() {
    assertNull(StringUtilsSamples.substring( str: null,  start: 0,  end: 13));
}
```

- T2: passando in input una stringa vuota e dei parametri qualunque per start ed end, il metodo verifica questa casistica con una condizione ed evita il proseguimento ritornando la stessa stringa vuota, il test conferma questo comportamento.

```
@Test
void substringWithEmptyInput() {
    assertEquals( expected: "", StringUtilsSamples.substring( str: "",  start: 0,  end: 13));
}
```

- T3: passando in input uno spazio vuoto e dei parametri qualunque per start ed end, il metodo verifica la casistica e restituirà come sottostringa al più il carattere vuoto passatogli, il test conferma questo comportamento.

```
@Test
void substringWithSpaceInput() {
    assertEquals( expected: " ", StringUtilsSamples.substring( str: " ",  start: 0,  end: 13));
}
```

- T4: passando in input una stringa qualunque, un intero negativo per start e un intero positivo pari alla lunghezza della stringa per end, il metodo verifica la casistica e restituirà come sottostringa una stringa con i caratteri che vanno dalla posizione 11 della stringa madre alla posizione 13 in quanto start si sposterà di 3 posizioni a sinistra partendo dall'ultimo elemento della stringa, il test conferma questo comportamento.

```
@Test
void substringNegativeStart() {
    assertEquals( expected: "ld!", StringUtilsSamples.substring( str: "Hello, World!",  start: -3,  end: 13));
}
```

- T5: passando in input una stringa qualunque, un intero negativo per start molto grande e un intero positivo più grande della lunghezza della stringa per end, il metodo verifica la casistica e restituirà come sottostringa una stringa pari a quella passatagli in input in quanto entra in gioco il controllo per start e end per cui start viene sostituito con 0 mentre end viene sostituito dalla lunghezza della stringa, il test conferma questo comportamento.

```
@Test
void subStringWithBigNegativeStart() {
    assertEquals( expected: "Hello, World!", StringUtilsSamples.substring( str: "Hello, World!",  start: -32518,  end: 32));
}
```

- T6: passando in input una stringa qualunque, un intero positivo per start e un intero negativo per end, il metodo verifica la casistica e restituirà come sottostringa una stringa con i caratteri che vanno dalla posizione 0 della stringa madre alla posizione 8 in quanto end si sposterà di 5 posizioni a sinistra partendo dall'ultimo elemento della stringa, il test conferma questo comportamento.

```
@Test
void substringNegativeEnd() {
    assertEquals( expected: "Hello, W", StringUtilsSamples.substring( str: "Hello, World!",  start: 0,  end: -5));
}
```

- T7: passando in input una stringa qualunque, un intero negativo per end molto grande e un intero negativo molto grande per end ma meno di start, il metodo verifica la casistica e restituirà come sottostringa una stringa vuota in quanto entra in gioco il controllo per start e end per cui entrambi vengono sostituiti con 0, il test conferma questo comportamento.

```
@Test
void subStringWithBigNegativeEnd() {
    assertEquals( expected: "", StringUtilsSamples.substring( str: "Hello, World!", start: -1234, end: -123));
}
```

- T8: passando in input una stringa qualunque, e start ed end interi positivi uguali tra loro, il metodo verifica la casistica e restituirà come stringa, una stringa vuota, il test conferma questo comportamento.

```
@Test
void substringWithSameStartAndEnd() {
    assertEquals( expected: "", StringUtilsSamples.substring( str: "Hello, World!", start: 6, end: 6));
}
```

- T9: passando in input una stringa qualunque, e start ed end interi negativi, il metodo verifica la casistica e restituirà come stringa, una sottostringa che va dalla posizione 9 alla posizione 12, in quanto entrambi partiranno nel conteggio a partire dall'ultimo elemento della stringa e muovendosi verso sinistra, differirà da T7 in quanto stiamo prendendo in considerazione valori che sono nei limiti della lunghezza della stringa in termini assoluti. Il test conferma questo comportamento.

```
@Test
void substringWithNegativeStartAndEnd() {
    assertEquals( expected: "rld", StringUtilsSamples.substring( str: "Hello, World!", start: -4, end: -1));
}
```

- T10: passando in input una stringa qualunque, un valore intero positivo per start maggiore della lunghezza della stringa e un valore intero positivo per end pari alla lunghezza della stringa, il metodo verificherà la casistica e restituirà una stringa vuota in quanto nella generazione della sottostringa si procede prendendo caratteri da sinistra verso destra. Il test conferma questo comportamento.

```
@Test
void substringWithStartGreaterThanLength() {
    assertEquals( expected: "", StringUtilsSamples.substring( str: "Hello, World!", start: 15, end: 13));
}
```

- T11: passando in input una stringa qualunque, un valore intero positivo per start e un valore intero positivo maggiore della lunghezza della stringa per end, il metodo verificherà la casistica e restituirà una sottostringa pari a quella madre in quanto end verrà sostituito con la lunghezza massima della stringa. Il test conferma questo comportamento.

```
@Test
void substringWithEndGreaterThanLength() {
    assertEquals( expected: "Hello, World!", StringUtilsSamples.substring( str: "Hello, World!",  start: 0,  end: 19));
}
```

- T12: passando in input una stringa qualunque, un valore intero positivo per start pari alla lunghezza della stringa e un valore intero positivo per end minore di start minore di start, il metodo verificherà la casistica e restituirà una stringa vuota in quando il comportamento è simile a T10 pur restando nei limiti. Il test conferma questo comportamento.

```
@Test
void substringWithStartEqualToLength() {
    assertEquals( expected: "", StringUtilsSamples.substring( str: "Hello, World!",  start: 13,  end: 7));
}
```

- T13: passando in input una stringa che contenga dei caratteri di controllo, e start ed end nei limiti della lunghezza della stringa, il metodo verificherà la casistica e restituirà una sottostringa che rispetti il carattere di controllo in quanto lo leggerà come un unico carattere. Il test conferma questo comportamento.

```
@Test
void substringWithControlCharacter() {
    assertEquals( expected: "World!", StringUtilsSamples.substring( str: "Hello,\tWorld!",  start: 7,  end: 13));
}
```

- T14: passando in input una stringa che contenga dei caratteri Unicode (in questo caso una emoji), e come start ed end dei valori qualsiasi che siano nei limiti ed includano i caratteri in questione, il metodo verificherà la casistica e restituirà una sottostringa che contiene il carattere passatogli, riuscendolo a leggere. Il test conferma questo comportamento.

```
@Test
public void testSubstringUnicode() {
    String str = "Hello, \uD83D\uDE03 World!";
    assertEquals( expected: "\uD83D\uDE03 World!", StringUtilsSamples.substring(str,  start: 7, str.length()));
}
```

2.2 swapCase

2.2.1 Analisi dei requisiti

Lo scopo di questa funzione è quello di invertire la capitalizzazione di ogni carattere per il parametro **str** di tipo *String* ricevuto in input, quindi restituendo in output un altro oggetto *String* rappresentante il risultato.

2.2.2 Esplorazione di input, output e definizione partizioni

A seguito di un'analisi del codice della funzione, sono emerse le seguenti partizioni per il parametro di input **str**:

- *null*, essendo *String* per definizione nullable
- stringa di lunghezza zero, quindi stringa vuota
- stringa di lunghezza maggiore di zero

L'output può essere esclusivamente un oggetto di tipo *String*. conversione del parametro **str** con i caratteri alfabetici con capitalizzazione invertita, valido sia per i caratteri con o senza accenti, mentre i caratteri numerici o simboli dovrebbero restare inalterati.

2.2.3 Casi limite

Inoltre, dalle analisi sono emersi alcuni casi limite, legati a valori che l'input può assumere ma che possono causare problemi in fase di esecuzione:

- input *null*, mancanza di una gestione di questa casistica
- input vuoto, il codice itera sulla lunghezza della stringa che in questo caso è zero, quindi non usuale

2.2.4. Ideazione dei test cases

Per via della natura della funzione che va a sostituire ogni carattere con il suo corrispettivo maiuscolo o minuscolo a seconda della forma passata in input, è importante verificare il comportamento per caratteri particolari, che non possono essere capitalizzati, da questa osservazione derivano i test cases iniziali:

- T3: input stringa composta dal carattere di spazio
- T4: input stringa con caratteri alfabetici in minuscolo
- T5: input stringa con caratteri alfabetici in maiuscolo
- T6: input stringa con caratteri alfabetici in minuscolo e maiuscolo
- T7: input stringa composta da caratteri alfabetici in minuscolo
- con spazio iniziale e finale
- T8: input stringa composta da caratteri alfabetici in maiuscolo con spazio iniziale e finale
- T9: input stringa composta da caratteri alfabetici in minuscolo e maiuscolo
- con spazio iniziale e finale
- T10: input stringa composta da caratteri alfabetici in minuscolo spaziati tra loro
- T11: input stringa composta da caratteri alfabetici in maiuscolo spaziati tra loro
- T12: input stringa composta da caratteri alfabetici in minuscolo e maiuscolo spaziati tra loro

- T13: input stringa composta da caratteri alfabetici con accenti, dieresi o similari
- T14: input stringa composta da caratteri numerici
- T15: input stringa composta da caratteri alfanumerici
- T16: input stringa composta da simboli
- T17: input stringa composta da caratteri alfanumerici e simboli

Test dei casi limite:
- T1: input *null*
- T2: input stringa vuota

2.2.5 Automazione dei test

Per automatizzare i test, si è optato per l'uso della libreria JUnit 5.10.0, si è fatto uso della funzione *assertEquals* e *assertThrows*. Il codice risultante è il seguente:

- T1: utilizzando *null*, come parametro di input, il metodo *swapCase* risponde con il lancio dell'eccezione *NullPointerException*, il test conferma questo comportamento.

```
@Test
void swapCaseNullInput() {
    assertThrows(IllegalArgumentException.class, () -> StringUtilsSamples.swapCase( str: null));
}
```

- T2: passando come input una stringa vuota, il metodo verifica questa casistica con una condizione, ed evita il proseguimento ritornando la stessa stringa vuota, il test conferma questo comportamento.

```
@Test
void swapCaseEmptyInput() {
    assertEquals( expected: "", StringUtilsSamples.swapCase( str: ""));
}
```

- T3: passando come input un carattere di spaziatura, il metodo dovrebbe restituire lo stesso carattere in quanto non presenta forme differenti in base alla capitalizzazione, il test conferma questo comportamento.

```
@Test
void swapCaseSpaceInput() {
    assertEquals( expected: " ", StringUtilsSamples.swapCase( str: " "));
}
```

- T4: passando come input una stringa alfabetica in minuscolo, il metodo dovrebbe restituire la stessa stringa in maiuscolo, il test conferma questo comportamento.

```
@Test
void swapCaseInputToUpperCase() {
    assertEquals( expected: "ABC", StringUtilsSamples.swapCase( str: "abc"));
}
```

- T5: passando come input una stringa alfabetica in maiuscolo, il metodo dovrebbe restituire la stessa stringa in minuscolo, il test conferma questo comportamento.

```
@Test
void swapCaseInputToLowerCase() {
    assertEquals( expected: "abc", StringUtilsSamples.swapCase( str: "ABC"));
}
```

- T6: passando come input una stringa alfabetica con caratteri in maiuscolo e minuscolo, il metodo dovrebbe restituire la stessa stringa con capitalizzazione invertita, il test conferma questo comportamento.

```
@Test
void swapCaseInputMixedCase() {
    assertEquals( expected: "aBc", StringUtilsSamples.swapCase( str: "AbC"));
}
```

- T7: passando come input una stringa alfabetica in minuscolo aperta e chiusa da spazi, il metodo dovrebbe restituire la stessa stringa in maiuscolo e lasciando inalterati gli spazi, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSpacesStartingEndingToUpperCase() {
    assertEquals( expected: " ABC ", StringUtilsSamples.swapCase( str: " abc "));
}
```

- T8: passando come input una stringa alfabetica in maiuscolo aperta e chiusa da spazi, il metodo dovrebbe restituire la stessa stringa in minuscolo e lasciando inalterati gli spazi, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSpacesStartingEndingToLowerCase() {
    assertEquals( expected: " abc ", StringUtilsSamples.swapCase( str: " ABC "));
}
```

- T9: passando come input una stringa alfabetica con caratteri in maiuscolo e minuscolo aperta e chiusa da spazi, il metodo dovrebbe restituire la stessa stringa con capitalizzazione invertita e lasciando inalterati gli spazi, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSpacesStartingEndingMixedCase() {
    assertEquals( expected: " aBc ", StringUtilsSamples.swapCase( str: " AbC "));
}
```

- T10: passando come input una stringa alfabetica in minuscolo con caratteri separati da spazi, il metodo dovrebbe restituire la stessa stringa in maiuscolo e lasciando inalterati gli spazi, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSpacesInBetweenToUpperCase() {
    assertEquals( expected: "A B C", StringUtilsSamples.swapCase( str: "a b c"));
}
```

- T11: passando come input una stringa alfabetica in maiuscolo con caratteri separati da spazi, il metodo dovrebbe restituire la stessa stringa in minuscolo e lasciando inalterati gli spazi, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSpacesInBetweenToLowerCase() {
    assertEquals( expected: "a b c", StringUtilsSamples.swapCase( str: "A B C"));
}
```

- T12: passando come input una stringa alfabetica con caratteri in minuscolo e maiuscolo separati da spazi, il metodo dovrebbe restituire la stessa stringa con capitalizzazione invertita e lasciando inalterati gli spazi, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSpacesInBetweenMixedCase() {
    assertEquals( expected: "a B c", StringUtilsSamples.swapCase( str: "A b C"));
}
```

- T13: passando come input una stringa composta da un carattere con accento o dieresi, il metodo dovrebbe restituire lo stesso carattere con la capitalizzazione invertita, nel test “è” diventa effettivamente “È”, quindi il comportamento viene confermato.

```
@Test
void swapCaseInputWithAccentCharacter() {
    assertEquals( expected: "È", StringUtilsSamples.swapCase( str: "è"));
}
```

- T14: passando come input una stringa composta da un carattere numerico, il metodo dovrebbe restituire lo stesso carattere, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithNumberCharacter() {
    assertEquals( expected: "1", StringUtilsSamples.swapCase( str: "1"));
}
```

- T15: passando come input una stringa composta da caratteri alfabetici e numerici, il metodo dovrebbe restituire il carattere alfabetico con capitalizzazione invertita e inalterato quello numerico, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithNumberCharacterAndAlphabeticalCharacter() {
    assertEquals( expected: "2X", StringUtilsSamples.swapCase( str: "2x"));
}
```

- T16: passando come input una stringa composta da simboli, il metodo dovrebbe restituire il simbolo inalterato, il test conferma questo comportamento.

```
@Test
void swapCaseInputWithSymbols() {
    assertEquals( expected: "/", StringUtilsSamples.swapCase( str: "/"));
}
```

- T17: passando come input una stringa composta da caratteri alfabetici, numerici e simboli, il metodo dovrebbe restituire il carattere alfabetico con capitalizzazione inversa mentre il carattere numerico e il simbolo dovrebbero restare inalterati, il test conferma questo comportamento.

```
@Test
void swapCaseMixedInput() {
    assertEquals( expected: "2X+1", StringUtilsSamples.swapCase( str: "2x+1"));
}
```

3.0 - Discussione dei risultati e ampliamento test

3.1 substring

I test realizzati hanno avuto esito positivo, la funzione è già prevista di caso *null* per la stringa in ingresso; pertanto, restituirà un valore nullo in uscita. La funzione *substring* di StringUtils è in grado di gestire start ed end esterni al range di lunghezza della stringa in input, questo però non accade con il metodo base di java in cui bisogna gestire eccezioni quali String*IndexOutOfBoundsException*, principalmente per lo start che risulta essere sprovvisto di gestione dei valori negativi. Il metodo in esame, invece, è in grado di gestire gli indici negativi in 2 modi cioè portanto a 0 il valore dell'indice coinvolto nel caso in cui il valore in input è troppo elevato rispetto alla lunghezza della stringa, oppure andando a sommare al valore negativo la lunghezza della stringa, entro certi limiti.

3.2 swapCase

I test realizzati hanno tutti esito positivo, compresi i casi limite, in caso di input uguale a *null*, il metodo lancia un'eccezione di tipo *NullPointerException*, aspetto che potrebbe essere gestito meglio, magari con una sostituzione del parametro *str* con una stringa default. Per quanto riguarda il caso di input pari a stringa vuota, il metodo ha una gestione della casistica nella quale evita di entrare nel ciclo *for*, per la conversione ma direttamente ritorna la stessa stringa vuota, approccio logicamente corretto e che va ad evitare possibili eccezioni come *IndexOutOfBoundException*, che si verificherebbe se si provasse ad accedere al primo elemento della stringa, che essendo vuota ha dimensione pari a zero, quindi non possiede alcun elemento.

3.3. Analisi code coverage

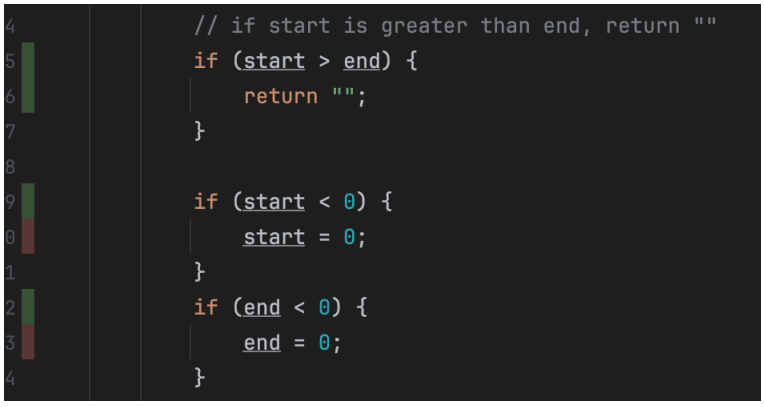
Per verificare l'efficacia dei test, si è analizzato il report relativo al code coverage, che ha evidenziato una non totale copertura, per entrambi i metodi, di seguito i dettagli.

3.3.1 substring

```java
    public static String substring(final String str, int start, int end) {
        if (str == null) {
            return null;
        }

        // handle negatives
        if (end < 0) {
            end = str.length() + end; // remember end is negative
        }
        if (start < 0) {
            start = str.length() + start; // remember start is negative
        }

        // check length next
        if (end > str.length()) {
            end = str.length();
        }

        // if start is greater than end, return ""
        if (start > end) {
            return "";
        }

        if (start < 0) {
            start = 0;
        }
        if (end < 0) {
            end = 0;
        }

        return str.substring(start, end);
    }
```

3.3.1.1 line coverage

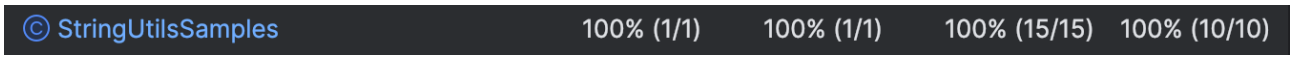
Per quanto concerne il metodo *substring*, line coverage era pari 86% con 2 linee di codice non coperte su 15.



Questa mancanza è stata risolta aggiungendo i test:

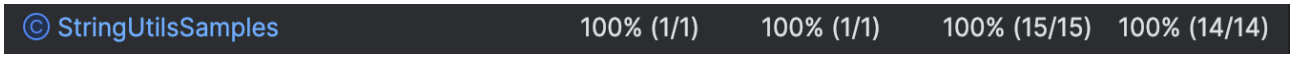
- subStringBigNegativeStart
- subStringBigNegativeEnd

Raggiungendo così il 100% della line coverage



3.3.1.2 branch coverage

Il branch coverage invece vedeva una copertura dell'85% delle linee, coprendo 12 diramazioni su 14 ma con la gestione, ma con l'aggiunta dei due test citati in precedenza si è arrivati al 100%



3.3.2 swapCase

Per quanto riguarda il metodo *swapCase*, la line coverage è pari al 100%, ma la branch coverage non è totale. Ciò è dovuto alla mancata copertura di un branch nella condizione che verifica se un carattere è in maiuscolo con il metodo *isUpperCase* di Java o in quello che viene definito titleCase, con il metodo Java *isTitleCase*. Questo metodo ritorna *true* solo in presenza di particolari caratteri come Dz, che è un particolare carattere fonetico. Per correggere questa mancanza nella copertura è stato ideato e automatizzato il seguente test:

- T18: passando come input un carattere in titleCase, il metodo dovrebbe restituire lo stesso carattere con capitalizzazione inversa, il test conferma questo comportamento.

```
@Test
void swapCaseInputInTitleCase() {
    assertEquals( expected: "ǆ", StringUtilsSamples.swapCase( str: "ǅ"));
}
```

A seguito di questa implementazione la copertura sia logica che a livello di codice per il metodo *swapCase* è totale.

```
public static String swapCase(final String str) {
    if (Objects.equals(str, "")) {
        return str;
    }

    final int strLen = str.length();
    final int[] newCodePoints = new int[strLen]; // cannot be longer than the char array
    int outOffset = 0;
    for (int i = 0; i < strLen; ) {
        final int oldCodepoint = str.codePointAt(i);
        final int newCodePoint;
        if (Character.isUpperCase(oldCodepoint) || Character.isTitleCase(oldCodepoint)) {
            newCodePoint = Character.toLowerCase(oldCodepoint);
        } else if (Character.isLowerCase(oldCodepoint)) {
            newCodePoint = Character.toUpperCase(oldCodepoint);
        } else {
            newCodePoint = oldCodepoint;
        }
        newCodePoints[outOffset++] = newCodePoint;
        i += Character.charCount(newCodePoint);
    }
    return new String(newCodePoints, 0, outOffset);
}
```

Home Work 2

1.0 – Introduzione

In merito allo sviluppo di property-based test, si è optato per utilizzare i metodi utilizzati finora, in quanto è possibile traslare tutte le partizioni rilevate, in proprietà e testarle con l'aiuto del framework jqwik nella versione 1.8.3, che genera un gran numero di input di test, e soprattutto esaustivo in grado di coprire tutte le eventualità previste.

2.0 – Definizione delle proprietà da testare

Analogamente a quanto fatto per la definizione delle partizioni, è possibile definire delle proprietà che rappresentano il comportamento da verificare del metodo, soggetto a determinati tipi input, generati questa volta in maniera casuale dal framework jqwik, ma entro un range ampio di valori ma comunque controllati in parte dai tester.

2.1 substring

Per il metodo *substring* le proprietà individuate e testate sono le seguenti:

- Input stringa *null e qualsiasi valore di start ed end*, il metodo deve restituire come risposta un elemento null.

```
@Property
void substringWithNullInput(@ForAll("nullSourceProvider") String string, @ForAll("intTotRange") int start, @ForAll("intTotRange") int end) {
    assertNull(StringUtilsSamples.substring(string, start, end));
}
```

- Input stringa vuota e qualsiasi valore di start ed end positivo, il metodo deve restituire una stringa vuota.

```
@Property
void substringWithEmptyStringInput(@ForAll("emptyOrWhitespaceSourceProvider") String string, @ForAll("intPosRange") int start, @ForAll("intPosRange") int end) {
    if (Objects.equals(string, b: "")) {
        assertEquals( expected: "", StringUtilsSamples.substring(string, start, end));
    }
}
```

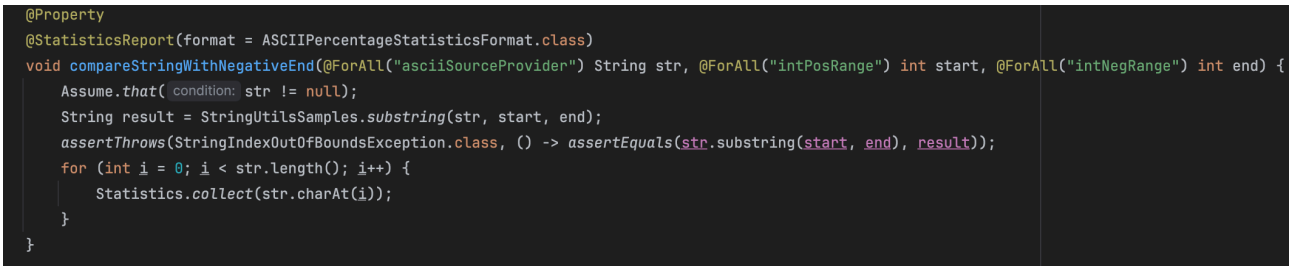
- Input stringa composta da qualsiasi carattere, assumendo che non sia nulla e start è minore di end, con start ed end interi positivi, il metodo confrontato con il *substring* di Java restituirà lo stesso risultato di StringUtils, ovvero una sottostringa di dimensione inferiore o uguale a quello passatagli in input.

```
@Property
@StatisticsReport(format = ASCIIPercentageStatisticsFormat.class)
void compareSubstringWithinLimits(@ForAll("asciiSourceProvider") String str, @ForAll("intPosRange") int start, @ForAll("intPosRange") int end) {
    Assume.that( condition: str != null);
    Assume.that( condition: start < end);
    String result = StringUtilsSamples.substring(str, start, end);
    if (start < str.length() && end <= str.length()) {
        assertEquals(str.substring(start, end), result);
    }
    for (int i = 0; i < str.length(); i++) {
        Statistics.collect(str.charAt(i));
    }
}
```

- Input stringa composta da qualsiasi carattere, assumendo che non sia nulla, con start intero negativo ed end qualsiasi intero, il metodo confrontato con il *substring* di Java dovrà restituire un *StringIndexOutOfBoundsException* in quanto *substring* di Java non è in grado autonomamente di gestire uno start negativo, a differenza del metodo di StringUtils in esame.

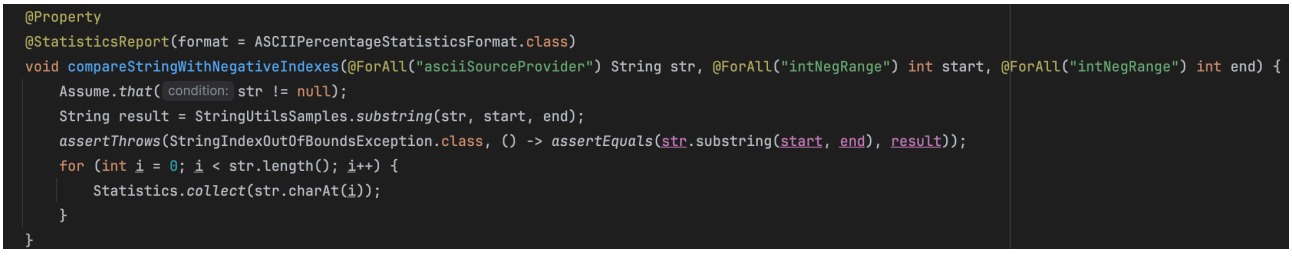
```
@Property
@StatisticsReport(format = ASCIIPercentageStatisticsFormat.class)
void compareStringWithStartLessThanZeroWithJavaFunction(@ForAll("asciiSourceProvider") String str, @ForAll("intNegRange") int start, @ForAll("intTotRange") int end) {
    Assume.that( condition: str != null);
    String result = StringUtilsSamples.substring(str, start, end);
    if (start < 0) {
        assertThrows(StringIndexOutOfBoundsException.class, () -> assertEquals(str.substring(start, end), result));
    }
    for (int i = 0; i < str.length(); i++) {
        Statistics.collect(str.charAt(i));
    }
}
```

- Input stringa composta da qualsiasi carattere, assumendo che non sia nulla, con start intero positivo ed end qualsiasi intero negativo, il metodo confrontato con il *substring* di Java dovrà restituire un *StringIndexOutOfBoundsException* in quanto *substring* di Java non è in grado autonomamente di gestire end negativo, a differenza del metodo di StringUtils in esame.



```
@Property
@StatisticsReport(format = ASCIIPercentageStatisticsFormat.class)
void compareStringWithNegativeEnd(@ForAll("asciiSourceProvider") String str, @ForAll("intPosRange") int start, @ForAll("intNegRange") int end) {
    Assume.that( condition: str != null);
    String result = StringUtilsSamples.substring(str, start, end);
    assertThrows(StringIndexOutOfBoundsException.class, () -> assertEquals(str.substring(start, end), result));
    for (int i = 0; i < str.length(); i++) {
        Statistics.collect(str.charAt(i));
    }
}
```

- Input stringa composta da qualsiasi carattere, assumendo che non sia nulla, con start ed end interi negativi, il metodo confrontato con il *substring* di Java dovrà restituire un *StringIndexOutOfBoundsException* in quanto *substring* di Java non è in grado autonomamente di gestire start ed end negativi a differenza del metodo di StringUtils in esame.



```
@Property
@StatisticsReport(format = ASCIIPercentageStatisticsFormat.class)
void compareStringWithNegativeIndexes(@ForAll("asciiSourceProvider") String str, @ForAll("intNegRange") int start, @ForAll("intNegRange") int end) {
    Assume.that( condition: str != null);
    String result = StringUtilsSamples.substring(str, start, end);
    assertThrows(StringIndexOutOfBoundsException.class, () -> assertEquals(str.substring(start, end), result));
    for (int i = 0; i < str.length(); i++) {
        Statistics.collect(str.charAt(i));
    }
}
```

- Input stringa composta da qualsiasi carattere, assumendo che non sia nulla e che start maggiore o uguale a end, di cui start qualsiasi intero ed end intero positivo, il metodo restituirà una sottostringa vuota, in quanto la generazione delle sottostringhe prendono valori partendo da start e concludendo in end da sinistra verso destra.

```
@Property
@StatisticsReport(format = ASCIIPercentageStatisticsFormat.class)
void substringWithStartGreaterOrEqualThanEnd(@ForAll("asciiSourceProvider") String str, @ForAll("intTotRange") int start, @ForAll("intPosRange") int end) {
    Assume.that( condition: str != null);
    Assume.that( condition: start >= end);
    String result = StringUtilsSamples.substring(str, start, end);
    assertEquals( expected: "", result);
    for (int i = 0; i < str.length(); i++) {
        Statistics.collect(str.charAt(i));
    }
}
```

- Input stringa composta da qualsiasi carattere, assumendo che non sia nulla e che start maggiore o uguale a end, di cui start qualsiasi intero ed end intero positivo, il metodo restituirà una sottostringa vuota, in quanto la generazione delle sottostringhe prendono valori partendo da start e concludendo in end da sinistra verso destra, quindi come il caso precedente oltre al fatto che se entrambi superano i limiti della dimensione della stringa verranno sostituisti con la lunghezza della stringa stessa. Pertanto entrebbe in gioco il caso di start ed end uguali.

```
@Property
@StatisticsReport(format = ASCIIPercentageStatisticsFormat.class)
void substringWithStartGreaterOrEqualThanStringLength(@ForAll("asciiSourceProvider") String str, @ForAll("intPosRange") int start, @ForAll("intTotRange") int end) {
    Assume.that( condition: str != null);
    String result = StringUtilsSamples.substring(str, start, end);
    if (start > str.length()) {
        assertEquals( expected: "", result);
    }
    for (int i = 0; i < str.length(); i++) {
        Statistics.collect(str.charAt(i));
    }
}
```

2.2 swapCase

Per il metodo *swapCase* le proprietà individuate e testate sono le seguenti:

- Input *null*, il metodo deve lanciare come risposta l'eccezione *NullPointerException*.

```
@Property
void swapCaseWithNullInput(@ForAll("nullSourceProvider") String s) {
    assertThrows(NullPointerException.class, () -> StringUtilsSamples.swapCase(s));
}
```

- Input stringa vuota o carattere di spazio, il metodo deve restituire il medesimo carattere in output.

```
@Property
void swapCaseWithEmptyOrWhitespaceStringInput(@ForAll("emptyOrWhitespaceSourceProvider") String s) {
    if (Objects.equals(s, b: "")) {
        assertEquals( expected: "", StringUtilsSamples.swapCase(s));
    }
    if (Objects.equals(s, b: " ")) {
        assertEquals( expected: " ", StringUtilsSamples.swapCase(s));
    }
}
```

- Input stringa composta da soli caratteri alfabetici in minuscolo, il metodo deve restituire la medesima stringa ma in maiuscolo.

```
@Property
@StatisticsReport(format = AlphabeticPercentageStatisticsFormat.class)
void swapCaseWithAlphabeticLowerCaseStringInput(@ForAll("alphabeticSourceProvider") String s) {
    assertEquals(s.toUpperCase(), StringUtilsSamples.swapCase(s.toLowerCase()));
    for (int i = 0; i < s.length(); i++) {
        Statistics.collect(s.charAt(i));
    }
}
```

- Input stringa composta da soli caratteri alfabetici in maiuscolo, il metodo deve restituire la medesima stringa ma in minuscolo.

```
@Property
@StatisticsReport(format = AlphabeticPercentageStatisticsFormat.class)
void swapCaseWithAlphabeticUpperCaseStringInput(@ForAll("alphabeticSourceProvider") String s) {
    assertEquals(s.toLowerCase(), StringUtilsSamples.swapCase(s.toUpperCase()));
    for (int i = 0; i < s.length(); i++) {
        Statistics.collect(s.charAt(i));
    }
}
```

- Input stringa composta da soli caratteri numerici, il metodo deve restituire la medesima stringa senza alterazioni.

```
@Property
@StatisticsReport(format = NumericPercentageStatisticsFormat.class)
void swapCaseWithNumericStringInput(@ForAll("numericSourceProvider") String s) {
    assertEquals(s, StringUtilsSamples.swapCase(s));
    for (int i = 0; i < s.length(); i++) {
        Statistics.collect(s.charAt(i));
    }
}
```

- Input stringa composta da caratteri numerici e alfabetici in minuscolo, il metodo deve restituire la medesima stringa con i caratteri numerici privi di alterazioni e quelli alfabetici in maiuscolo.

```
@Property
@StatisticsReport(format = AlphaNumericPercentageStatisticsFormat.class)
void swapCaseWithAlphaNumericLowerCaseStringInput(@ForAll("alphaNumericSourceProvider") String s) {
    assertEquals(s.toUpperCase(), StringUtilsSamples.swapCase(s.toLowerCase()));
    for (int i = 0; i < s.length(); i++) {
        Statistics.collect(s.charAt(i));
    }
}
```

- Input stringa composta da caratteri numerici e alfabetici in maiuscolo, il metodo deve restituire la medesima stringa con i caratteri numerici privi di alterazioni e quelli alfabetici in minuscolo.

```
@Property
@StatisticsReport(format = AlphaNumericPercentageStatisticsFormat.class)
void swapCaseWithAlphaNumericUpperCaseStringInput(@ForAll("alphaNumericSourceProvider") String s) {
    assertEquals(s.toLowerCase(), StringUtilsSamples.swapCase(s.toUpperCase()));
    for (int i = 0; i < s.length(); i++) {
        Statistics.collect(s.charAt(i));
    }
}
```

2.3 Provider Utilizzati

Per l'esecuzione dei test property si è optato per l'utilizzo dei seguenti provider:

- nullSourceProvider è un provider che fornisce solo elementi nulli con probabilità al 100%.

```
@Provide
private Arbitrary<String> nullSourceProvider() {
    return Arbitraries.oneOf(
            Arbitraries.strings().injectNull( nullProbability: 1)
    );
}
```

- emptyOrWhitespaceSourceProvider è un provider che fornisce una stringa vuota o un carattere di spazio.

```
@Provide
private Arbitrary<String> emptyOrWhitespaceSourceProvider() {
    return Arbitraries.strings().whitespace();
}
```

- alphabeticSourceProvider è un provider che fornisce una stringa dei soli 52 caratteri alfabetici.

```
@Provide
private Arbitrary<String> alphabeticSourceProvider() {
    return Arbitraries.strings().alpha();
}
```

- alphaNumericSourceProvider è un provider che fornisce una stringa dei 52 caratteri alfabetici o dei 10 caratteri numerici.

```
@Provide
private Arbitrary<String> alphaNumericSourceProvider() {
    return Arbitraries.strings().alpha().numeric();
}
```

- numericSourceProvider è un provider che fornisce una stringa dei soli 10 caratteri numerici.

```
@Provide
private Arbitrary<String> numericSourceProvider() {
    return Arbitraries.strings().numeric();
}
```

- intPosRange è un provider che fornisce un valore intero maggiore o uguale a 0.

```
@Provide
private IntegerArbitrary intPosRange() {
    return Arbitraries.integers().greaterOrEqual( i: 0);
}
```

- intNegRange è un provider che fornisce un valore intero minore o uguale a -1.

```
@Provide
private IntegerArbitrary intNegRange() {
    return Arbitraries.integers().lessOrEqual( i: -1);
}
```

- intTotRange è un provider che fornisce valori interi positivi o negativi.

```
@Provide
private IntegerArbitrary intTotRange() {
    return Arbitraries.integers();
}
```

- asciiSourceProvider è un provider che fornisce una stringa composta dai 256 caratteri ascii ad eccezione di quelli inclusi nel metodo “excludeChars”.

```
@Provide
private Arbitrary<String> asciiSourceProvider() {
    return Arbitraries.strings().withCharRange( c: '\u0000',  c1: 'ÿ').excludeChars('ß', 'µ').ascii();
}
```

- titleCaseProvider è un provider che fornisce una stringa composta dai caratteri speciali inclusi nel metodo "withChars" che sono definiti "titleCase".

```
@Provide
private Arbitrary<String> titleCaseProvider() {
    return Arbitraries.strings().withChars( charSequence: "DzLjDžNj");
}
```

3.0 – Statistiche

Per ogni proprietà testata, sono state generate delle statistiche realizzate ad hoc, effettuando un'implementazione dell'interfaccia *StatisticsReportFormat* di jqwik, e il successivo override del metodo *formatReport*. Ciò che si è voluto esaminare è la quantità e la percentuale che ogni carattere presenta all'interno delle 1000 stringhe che jqwik genera per eseguire i test. Oltre ciò, si è voluto verificare la completa copertura rispetto alla totalità delle tipologie di carattere relativa ad ogni test, ovvero rispetto ai caratteri alfabetici, numerici, alfanumerici e ascii.
Per ogni tipologia di stringa usata in input, è stata generata una relativa statistica, di seguito i dettagli:

- Con questa statistica si vogliono misurare le performance della generazione di caratteri alfabetici; a tal fine, essi vengono rilevati all'interno di ogni test attraverso un'iterazione su ogni stringa e ogni carattere viene passato alla funzione *collect*, della classe *Statistics* di jqwik. La statistica mostra la lista di ogni carattere generato accompagnato dal numero di occorrenze e la relativa percentuale. Al termine della lista vi è un calcolo della percentuale dei caratteri alfabetici rispetto ai 52 totali (26 minuscoli e 26 maiuscoli) dell'alfabeto latino tradizionale, escludendo caratteri particolari come lettere con accenti e dieresi. Inoltre, viene riportata la percentuale dei caratteri generati rispetto ai 256 caratteri ascii.

```
class NumericPercentageStatisticsFormat implements StatisticsReportFormat {
    @Override
    public List<String> formatReport(List<StatisticsEntry> entries) {
        List<String> unique = entries.stream().distinct().map(e -> e.name()).toList();
        double size = unique.size();
        return
                Stream.concat(
                        entries
                                .stream()
                                .map(
                                        e ->
                                                String.format("%s: %d (%s)", e.name(), e.count(), e.percentage())
                                ), Stream.of(
                                t: "Percentuale di caratteri numerici generati: " + ((size / 10) * 100) + "%\n" +
                                        "Percentuale di caratteri ASCII generati: " + ((size / 256) * 100) + "%"
                        )).toList();
    }
}
```

- Con questa statistica si vogliono misurare le performance della generazione di caratteri numerici. La statistica mostra la lista di ogni carattere generato accompagnato dal numero di occorrenze e la relativa percentuale. Al termine della lista vi è un calcolo della percentuale dei caratteri alfabetici rispetto alle 10 cifre totali. Inoltre, viene riportata la percentuale dei caratteri generati rispetto ai 256 caratteri ascii.

```
class AlphaNumericPercentageStatisticsFormat implements StatisticsReportFormat {
    @Override
    public List<String> formatReport(List<StatisticsEntry> entries) {
        List<String> unique = entries.stream().distinct().map(e -> e.name()).toList();
        double size = unique.size();
        return
                Stream.concat(
                        entries
                                .stream()
                                .map(
                                        e ->
                                                String.format("%s: %d (%s)", e.name(), e.count(), e.percentage())
                                ), Stream.of(
                                 t: "Percentuale di caratteri alfanumerici generati: " + ((size / 62) * 100) + "%\n" +
                                        "Percentuale di caratteri ASCII generati: " + ((size / 256) * 100) + "%"
                        )).toList();
    }
}
```

- Con questa statistica si vogliono misurare le performance della generazione di caratteri numerici. La statistica mostra la lista di ogni carattere generato accompagnato dal numero di occorrenze e la relativa percentuale. Al termine della lista vi è un calcolo della percentuale dei caratteri alfabetici rispetto ai 62 caratteri alfanumerici totali, ovvero 52 alfabetici e 10 numerici. Inoltre, viene riportata la percentuale dei caratteri generati rispetto ai 256 caratteri ascii.

```
class ASCIIPercentageStatisticsFormat implements StatisticsReportFormat {
    @Override
    public List<String> formatReport(List<StatisticsEntry> entries) {
        List<String> unique = entries.stream().distinct().map(e -> e.name()).toList();
        double size = unique.size();
        return
                Stream.concat(
                        entries
                                .stream()
                                .map(
                                        e ->
                                                String.format("%s: %d (%s)", e.name(), e.count(), e.percentage())
                                ), Stream.of( t: "Percentuale di caratteri ASCII generati: " + ((size / 256) * 100) + "%")).toList();
    }
}
```

- Con questa statistica si vogliono misurare le performance della generazione di caratteri ascii, quindi caratteri alfabetici, compresi quelli con accenti e dieresi, numerici e simboli. La statistica mostra la lista di ogni carattere generato accompagnato dal numero di occorrenze e la relativa percentuale. Al termine della lista viene riportata la percentuale dei caratteri generati rispetto ai 256 caratteri ascii.

Un esempio di quanto ottenuto segue:

```
statistics =
  e: 544 (1.8486424032351243)
  q: 525 (1.784075848710368)
  7: 521 (1.770482889863051)
  P: 508 (1.7263057736092704)
  W: 508 (1.7263057736092704)
  T: 506 (1.7195092941856118)
  8: 506 (1.7195092941856118)
  1: 503 (1.709314575050124)
  A: 500 (1.6991198559146363)
  C: 499 (1.695721616202807)
  H: 498 (1.6923233764909777)
  B: 497 (1.6889251367791485)
  Y: 497 (1.6889251367791485)
  k: 496 (1.6855268970673192)
  0: 494 (1.6787304176436606)
  j: 494 (1.6787304176436606)
  U: 493 (1.6753321779318313)
  9: 492 (1.671933938220002)
  5: 489 (1.6617392190845142)
  g: 488 (1.6583409793726849)
  b: 488 (1.6583409793726849)
  c: 486 (1.6515444999490263)
  S: 485 (1.6481462602371972)
  6: 485 (1.6481462602371972)
  h: 484 (1.644748020525368)
  L: 484 (1.644748020525368)
  t: 483 (1.6413497808135387)
  X: 482 (1.6379515411017094)
  l: 481 (1.63455330138988)
  F: 481 (1.63455330138988)
  r: 480 (1.6311550616780508)
  O: 480 (1.6311550616780508)
  x: 478 (1.6243585822543922)
  2: 478 (1.6243585822543922)
  z: 474 (1.610765623407075)
  w: 472 (1.6039691439834165)
  4: 471 (1.6005709042715872)
  D: 471 (1.6005709042715872)
  N: 470 (1.5971726645597581)
  3: 469 (1.5937744248479289)
  m: 469 (1.5937744248479289)
  a: 462 (1.5699867468651239)
  y: 458 (1.5563937880178067)
  Z: 458 (1.5563937880178067)
  s: 456 (1.5495973085941481)
  n: 454 (1.5428008291704898)
  o: 454 (1.5428008291704898)
  Q: 454 (1.5428008291704898)
  G: 453 (1.5394025894586605)
  v: 453 (1.5394025894586605)
  V: 452 (1.5360043497468312)
  M: 451 (1.532606110035002)
  d: 447 (1.5190131511876848)
  E: 443 (1.5054201923403676)
  R: 443 (1.5054201923403676)
  I: 437 (1.4850307540693921)
  f: 435 (1.4782342746457335)
  p: 434 (1.4748360349339042)
  K: 429 (1.4578448363747578)
  J: 428 (1.4544465966629285)
  u: 413 (1.4034730009854894)
  i: 404 (1.372888843579026)
  Percentuale di caratteri alfanumerici generati:
100.0%
Percentuale di caratteri ASCII generati: 24.21875%
```

I test property-based, eseguiti danno una percentuale del 100% relativa ai sottoinsiemi di caratteri corrispondenti agli alfabetici, numerici e alfanumerici. Per quanto riguarda il totale dei caratteri ascii, misurato con la statistica *ACIIPercentageStatisticsFormat*, si raggiunge un 99.21% in quanto due caratteri, rispettivamente ß e µ, non vengono correttamente convertiti dal metodo *swapCase*, e quindi farebbero fallire il test.

4.0 – Considerazioni

Prendendo in esame i metodi *substring* e *swapCase* di *StringUtils*, si è arrivati alla conclusione che i test effettuati funzionano in maniera ottimale per la maggior parte dei casi. Fermo restando che sono emerse alcune mancanze, quali l'errata conversione di alcuni caratteri come ß e µ, che rientrano nei 256 caratteri ascii, come già dichiarato precedentemente. Per completezza documentativa sono stati provati i test precedenti con un provider risultante dall'utilizzo di *Arbitraries.strings().all()*, che genera stringhe contenenti tutti i caratteri Unicode, causando ulteriori casi di errore nella conversione. Pertanto, il metodo *swapCase* andrebbe rivisto per gestire questi caratteri divergenti, oltre che una gestione migliore del caso di input *null,* evitando il lancio dell'eccezione *NullPointerException*.

Per quanto concerne il metodo *substring*, l'elevato numero di controlli presenti, consentono una gestione totale dei casi rispetto al metodo omonimo di Java; pertanto, non sono emerse modifiche da apporvi.

Infine, mostriamo interesse nel voler segnalare la presenza di un comportamento anomalo verificatosi nell'esecuzione dei test. Difatti, i test sono stati sviluppati ed eseguiti interamente su macchine **macOS**, portando tutti a esito positivo. Diversamente, gli stessi testati su macchine **Windows**, non sono andati tutti a buon fine, più precisamente alcuni test del metodo *swapCase*, in particolare quelli che fanno uso di caratteri alfabetici con accenti e dieresi o *titleCase*.