ITA - TDD - Week 1

Desenvolvimento de Software Guiado por Testes

Introdução:

Durante o desenvolvimento do desafio proposto, foram aplicados e revisados os conceitos do TDD, ensinados nas aulas ministradas na primeira semana. Bem como os ciclos de desenvolvimento do TDD, que são:

1. Desenvolver um pequeno teste automatizado com o caso de uso ou requisito mais simples do programa, inicialmente esse teste deverá falhar;
2. Desenvolver a classe ou objeto necessária, com o objetivo de passar no teste desenvolvido na etapa anterior;
3. Refatorar o código desenvolvido afim de manter, boas práticas de desenvolvimento, facilitando a manutenibilidade do código;

Desafio - Quebra de Strings com CamelCase:

1- Check convert simple string to list :

Este teste tem como objetivo principal de converter uma string simples para uma lista, após o desenvolvimento do teste, rodamos e o teste falhou, por conta da falta de implementação da classe CamelCase;

```
@Test
    public void checkConvertSimpleStringToList() {
        List<String> expectedList = new ArrayList<String>();
        expectedList.add("nome");
        List<String> receivedList = CamelCase.toTransform("nome");
        assertEquals(expectedList, receivedList);
    }
```

1.1- Implementing CamelCase Class and toTransform method :

Foi implementada a classe CamelCase e o método toTransform(String value) com o objetivo de passar no caso de teste;

```java
public class CamelCase {
    public static List<String> toTransform(String value) {
        List<String> list = new ArrayList<String>();
        list.add(value);
        return list;
    }
```

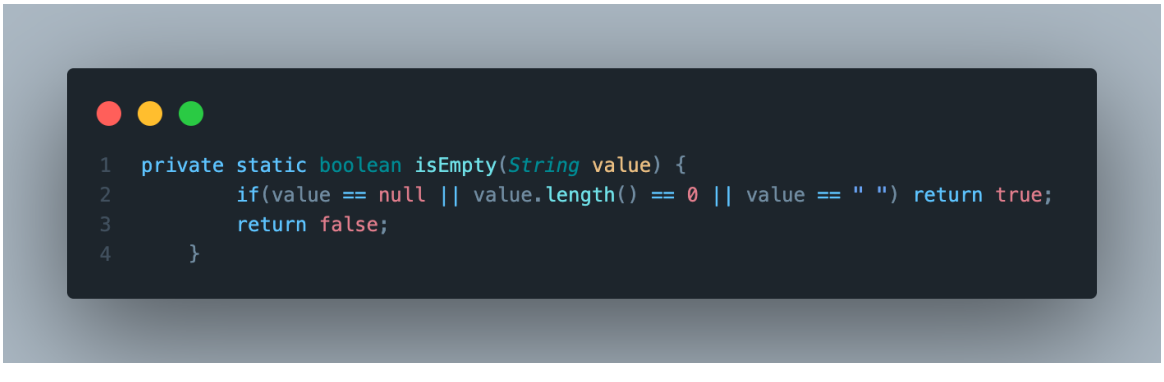
2- Check empty string:

Esse teste tem como objetivo principal garantir que ao tentar transformar uma string vazia, não deverá ser possível adiciona-lá à lista;

```java
@Test (expected = EmptyStringCamelCaseException.class)
    public void checkEmptyString() {
     List<String> expectedList = new ArrayList<String>();
     List<String> receivedList = CamelCase.toTransform(" ");
     assertEquals(expectedList, receivedList);
    }
```

2.1- Check empty string:

Após o desenvolvimento do teste foi implementado o método IsEmpty(String value) na classe CamelCase, que retorna uma flag boolean para validar se a String está vazia ou não;



```java
private static boolean isEmpty(String value) {
        if(value == null || value.length() == 0 || value == " ") return true;
        return false;
    }
```

2.2- Implementing EmptyStringCamelCaseException :

Após o desenvolvimento do método foi implementada a classe EmptyStringCamelCaseException ;

```java
package CamelCase.Exception;


public class EmptyStringCamelCaseException extends RuntimeException {
    public EmptyStringCamelCaseException(String message) {
        super(message);
    }

}
```

2.3- Refactoring create method to apply validation :

Por último refatorei o código criando um método para aplicar a validação, caso o retorno do método isEmpty for verdadeiro irá estourar a exception > EmptyStringCamelCaseException ;

```java
private static void validations(String value) {
        if(isEmpty(value)) throw new EmptyStringCamelCaseException("This string is null or empty");
    }
```

Com isso foi necessário refatorar o método toTransform para aplicar a validação:

```java
public class CamelCase {
    public static List<String> toTransform(String value) {
        List<String> list = new ArrayList<String>();
        validations(value);
        list.add(value);
        return list;
    }
```

3- Check Init With Number:

No desenvolvimento deste teste é foi levado em conta o requisito de que caso o input inicie com um número retorne uma Exception do tipo InitWithNumberCamelCaseException, no primeiro ciclo foi desenvolvido o teste que a príncipio não passou;

```java
@Test(expected = InitWithNumberCamelCaseException.class)
    public void checkInitWithNumber() {
```

```
3          List<String> list = CamelCase.toTransform("10Primeiros");
4      }
```

3.1- Implementing initWithNumber method:

Nessa fase foi implementado o método initWithNumber que retorna uma flag informando se a string inicia com números.

```
1  private static boolean initWithNumber(String value) {
2          if(value.substring(0).matches("[0-9].*")) return true;
3          return false;
4      }
```

Posteriormente foi desenvolvida a Exception InitWithNumberCamelCaseException:

```
1  package CamelCase.Exception;
2
3  public class InitWithNumberCamelCaseException extends RuntimeException {
4      public InitWithNumberCamelCaseException(String message) {
5          super(message);
6      }
7
8  }
9
```

3.2- Refactoring validations method implementation:

Nessa fase foi necessário fazer a refatoração do método validations() afim de manter a organização do código:

```
1  private static void validations(String value) {
2          if(isEmpty(value)) throw new EmptyStringCamelCaseException("This string is null or empty");
3          if(initWithNumber(value)) throw new InitWithNumberCamelCaseException("String cannot start with numbers");
4      }
```

4- Check Init With Especial Char:

No desenvolvimento deste teste é foi levado como requisito, que não é permitido input contendo caracteres especiais bem como: @!#$%ˆ&~, entre outros com isso foi desenvolvido o teste esperando uma Exception como retorno:*

```
@Test(expected = HasEspecialCharCamelCaseException.class)
    public void checkInitWithEspecialCharacteres() {
        List<String> list = CamelCase.toTransform("#nome");
    }
```

4.1- Implementing HasEspecialCharCamelCaseException:

Nesta fase foi implementada a Exception HasEspecialCharCamelCaseException afim de atender o caso de teste:

```
package CamelCase.Exception;

public class HasEspecialCharCamelCaseException extends RuntimeException {

    public HasEspecialCharCamelCaseException(String message) {
        super(message);
    }
}
```

4.2- Implementing has Especial Characteres method :

Nesta fase foi implementada o método HasEspecialCharacteres(), afim de retornar uma flag caso o input tenha algum caractere especial:

```

private static boolean hasEspecialCharacteres(String value) {
    if(value.substring(0).matches("[çÇ$&+,:;=?@#|'<>.^*()%!-].*")) return true;
    return false;
}
```

4.3- Refactoring validations method implementation:

Nesta fase foi implementada a Exception HasEspecialCharCamelCaseException afim de atender o caso de teste:

```
private static void validations(String value) {
        if(isEmpty(value)) throw new EmptyStringCamelCaseException("This string is null or empty");
        if(initWithNumber(value)) throw new InitWithNumberCamelCaseException("String cannot start with numbers");
        if(hasEspecialCharacteres(value)) throw new HasEspecialCharCamelCaseException("String cannot start with especial characteres");
    }
```

5- Check Convert Compound String:

Nesta fase foi implementado o test CheckCompoundString afim de validar que uma string composta será quebrada partindo partindo da próxima ocorrência de UpperCase:

```
@Test
public void checkConvertCompoundString() {
    List<String> expectedList = new ArrayList<String>();
    expectedList.add("nome");
    expectedList.add("Composto");
    List<String> receivedList = CamelCase.toTransform("nomeComposto");
    assertEquals(expectedList, receivedList);
}
```

5.1- Implementing to transform method to split compound String:

Nesta fase foi implementada uma quebra de caracteres afim a cada ocorrência de camelCase, dessa forma atendendo todos os requisitos do desafio e concluindo o ciclo do TDD:

```
public static List<String> toTransform(String value) {
      List<String> list = new ArrayList<String>();
      validations(value);
      List<String> result = Arrays.asList(value.split("(?<!(^|[A-Z]))(?=[A-Z])|(?<!^)(?=[A-Z][a-z]|([0-9][0-9]))"));
      return result;
  }
```