1 - Criação da classe de teste com um teste de uma única palavra contendo apenas caracteres minúsculos:

```java
import static org.junit.Assert.*;
import org.junit.Test;

import java.util.Arrays;

public class ConversorTeste {
    @Test
    public void umaPalavraTodaMinuscula() {
        Conversor conversor = new Conversor();
        assertEquals(
                Arrays.asList("nome"),
                conversor.converterCamelCase("nome")
        );
    }
}
```

Nesse caso, foi necessário criar a classe Conversor:

```java
public class Conversor {

}
```

Após rodar os testes e falhar, foi preciso criar o método “converterCamelCase”:

```java
import java.util.ArrayList;
import java.util.List;

public class Conversor {

    public List<String> converterCamelCase(String original) {
        return new ArrayList<String>();
    }
}
```

Após rodar os testes e falhar novamente, foi realizado uma implentação simples que passasse no teste:

```java
import java.util.Arrays;
import java.util.List;

public class Conversor {

    public List<String> converterCamelCase(String original) {
        return Arrays.asList("nome");
    }
}
```

O teste passou.

2 - Teste com apenas uma palavra que começa com maiúscula:

```java
@Test
public void umaPalavraComecaComMaiuscula() {
    Conversor conversor = new Conversor();
    assertEquals(
            Arrays.asList("nome"),
            conversor.converterCamelCase("Nome")
    );
}
```

Após executar os testes, ambos passaram. Hora de refatorar o teste para extrair a instanciação do objeto testado para um método que será executado antes de cada teste:

```java
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

import java.util.Arrays;

public class ConversorTeste {
    private Conversor conversor;

    @Before
    public void setup() {
        conversor = new Conversor();
    }

    @Test
    public void umaPalavraTodaMinuscula() {
        assertEquals(
                Arrays.asList("nome"),
                conversor.converterCamelCase("nome")
        );
    }

    @Test
    public void umaPalavraComecaComMaiuscula() {
        assertEquals(
                Arrays.asList("nome"),
                conversor.converterCamelCase("Nome")
        );
    }
}
```

Os testes continuaram passando. Hora de criar mais um teste.

3 - Teste de camel case com duas palavras onde a primeira começa com minúscula:

```java
@Test
public void camelCaseComecaMinuscula() {
    assertEquals(
            Arrays.asList("nome", "composto"),
            conversor.converterCamelCase("nomeComposto")
    );
}
```

A implementação precisou ser alterada drasticamente para o terceiro teste. A primeira ideia, foi usar expressões regulares para separar o argumento recebido em grupos. Por exemplo, "`(^[a-z]+)`" casa com grupos de caracteres minúsculos no início, e "`([A-Z]{1}[a-z]+)`" casa com uma letra maiúscula seguida de um ou mais caracteres minúsculos, exatamente o que

precisamos para passar o nosso terceiro teste. A implementação do método ficou a seguinte:

```
public List<String> converterCamelCase(String original) {
    String grupos = "((^[a-z]+)|([A-Z]{1}[a-z]+))";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return gruposEmLista.stream().map(String::toLowerCase).collect(Collectors.toList());
}
```

Ou seja, primeiro os grupos são separados por espaço, em seguida é usado o método “split“ para que a String seja transformada em uma lista de palavras. Por último, é usado o método “toLowerCase “para deixar todas as palavras em minúscula. Após passar nos testes, achei melhor refatorar, pois muita gente tem dificuldade de entender expressões regulares:

```
public List<String> converterCamelCase(String original) {
    String grupos = "(" + sequenciaDeMinusculasNoComeco() + "|" + maiusculaSeguidaDeMinusculas() + ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return gruposEmLista.stream().map(String::toLowerCase).collect(Collectors.toList());
}

public String sequenciaDeMinusculasNoComeco() {
    return "(^[a-z]+)";
}

public String maiusculaSeguidaDeMinusculas() {
    return "([A-Z]{1}[a-z]+)";
}
```

Os testes continuaram passando.

4 - Teste de camel case com duas palavras onde a primeira começa com maiúscula:

```
@Test
public void CamelCaseComecaMaiuscula() {
    assertEquals(
            Arrays.asList("nome", "composto"),
            conversor.converterCamelCase("NomeComposto")
    );
}
```

Os testes continuaram passando.

5 - Teste de uma sigla:

```
@Test
public void apenasSigla() {
    assertEquals(
            Arrays.asList("CPF"),
            conversor.converterCamelCase("CPF")
    );
}
```

O teste falhou, pois o método sempre transforma todas as palavras em minúscula. Para contornar este problema, criei um método que converte as palavras em minúscula apenas

se ela não for uma sigla, ou seja, se possuir apenas caracteres maiúsculos (o métodos das expressões regulares não estão no print para dar ênfase aos métodos alterados ou criados):

```
public List<String> converterCamelCase(String original) {
    String grupos = "(" + sequenciaDeMinusculasNoComeco() + "|" + maiusculaSeguidaDeMinusculas() + ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return transformarEmMinusculaExcetoSigla(gruposEmLista);
}

private List<String> transformarEmMinusculaExcetoSigla(List<String> palavras) {
    List<String> resultado = new ArrayList<>();
    for (String palavra : palavras) {
        if (!ehSigla(palavra)) {
            palavra = palavra.toLowerCase();
        }
        resultado.add(palavra);
    }

    return resultado;
}

private boolean ehSigla(String string) {
    for (Character caractere : string.toCharArray()) {
        if (ehMinusculo(caractere)) {
            return false;
        }
    }

    return true;
}

private boolean ehMinusculo(Character caractere) {
    return Character.isLowerCase(caractere);
}
```

Todos os testes passaram. Dessa forma, foi visto mais uma oportunidade de refatoração. Dessa vez, no método “transformaMinusculaExcetoSigla”. Para evitar of “if” dentro do “for”, a lógica de transformar ou não uma determinada palavra foi extraída para um outro método, como pode ser visto abaixo:

```
private List<String> transformarEmMinusculaExcetoSigla(List<String> palavras) {
    List<String> resultado = new ArrayList<>();
    for (String palavra : palavras) {
        resultado.add(transformaNaoSiglasEmMinuscula(palavra));
    }

    return resultado;
}

private String transformaNaoSiglasEmMinuscula(String palavra) {
    return ehSigla(palavra) ? palavra : palavra.toLowerCase();
}
```

A refatoração anterior permitiu realizar mais um passo de refatoração. Usando o lambda do Java 8, foi possível retirar a variável temporária “resultado” e a lógica de dentro do “for”:

```
private List<String> transformarEmMinusculaExcetoSigla(List<String> palavras) {
    return palavras.stream()
                   .map(s -> transformaNaoSiglasEmMinuscula(s))
                   .collect(Collectors.toList());
}
```

Os testes continuaram passando.

6 - Teste com sigla + outra palavra:

```
@Test
public void siglaNoComeco() {
    assertEquals(
            Arrays.asList("CPF", "novo"),
            conversor.converterCamelCase("CPFNovo")
    );
}
```

O teste falhou, pois a expressão regular usada no momento não separa “CPFNovo” em dois grupos: “CPF” e “Novo”. Ela os mantém em apenas um “CPFNovo”, que depois de transformada, fica “cpfnovo”. Para isso, foi preciso adicionar mais uma expressão regular. Essa expressão regular não poderia separar em grupos de letras maiúsculas, pois o “N” de “CPFNovo”, faz parte de “Novo” e não de “CPF”. A expressão não pode casar a última letra maiuscula caso a próxima seja minúscula. Abaixo, o antes e o depois da introdução da nova expressão regular:
Antes:

```
public List<String> converterCamelCase(String original) {
    String grupos = "(" + sequenciaDeMinusculasNoComeco() + "|" + maiusculaSeguidaDeMinusculas() + ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return transformarEmMinusculaExcetoSigla(gruposEmLista);
}

public String sequenciaDeMinusculasNoComeco() {
    return "(^[a-z]+)";
}

public String maiusculaSeguidaDeMinusculas() {
    return "([A-Z]{1}[a-z]+)";
}
```

Depois (sem as expressões antigas para dar ênfase nas alterações):

```
public List<String> converterCamelCase(String original) {
    String grupos = "(" +
                        sequenciaDeMinusculasNoComeco() + "|" +
                        maiusculaSeguidaDeMinusculas() + "|" +
                        maiusculasConsecutivasSemAUltimaSeAProximaForMinuscula() +
                    ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return transformarEmMinusculaExcetoSigla(gruposEmLista);
}

public String maiusculasConsecutivasSemAUltimaSeAProximaForMinuscula() {
    return "([A-Z]+(?=([A-Z][a-z])|($)|([0-9])))";
}
```

Após a alteração acima, todos os testes passaram.

7 - Teste com outra palavra + sigla:

```
@Test
public void siglaNoFim() {
    assertEquals(
            Arrays.asList("numero", "CPF"),
            conversor.converterCamelCase("numeroCPF")
    );
}
```

Os testes continuaram passando.

8 - Teste com sigla no meio:

```
@Test
public void siglaNoFim() {
    assertEquals(
            Arrays.asList("numero", "CPF"),
            conversor.converterCamelCase("numeroCPF")
    );
}
```

Os testes continuam passando.

9 - Teste com dígito no meio:

```
@Test
public void comDigitoNoMeio() {
    assertEquals(
            Arrays.asList("recupera", "10", "primeiros"),
            conversor.converterCamelCase("recupera10primeiros")
    );
}
```

O teste falha pois não foi definido nenhum grupo que considera dígitos. Foi preciso criar mais um grupo na expressão regular:

```
public List<String> converterCamelCase(String original) {
    String grupos = "(" +
                    sequenciaDeMinusculasNoComeco() + "|" +
                    maiusculaSeguidaDeMinusculas() + "|" +
                    maiusculasConsecutivasSemAUltimaSeAProximaForMinuscula() + "|" +
                    sequenciaDeNumeros() +
                ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return transformarEmMinusculaExcetoSigla(gruposEmLista);
}

public String sequenciaDeNumeros() {
    return "([0-9]+)";
}
```

Os testes agora passaram.

10 - Percebi que no enunciado não havia exemplo com dígitos no final, então resolvi criar um teste para esta situação:

```
@Test
public void comDigitoNoFim() {
    assertEquals(
            Arrays.asList("recupera", "10"),
            conversor.converterCamelCase("recupera10")
    );
}
```

Os testes continuaram passando.

11 - Teste com dígito no começo: deve retornar erro. Neste caso, foi escolhido lançar uma IllegalArgumentException:

```
@Test(expected=IllegalArgumentException.class)
public void comecaComDigito() {
    conversor.converterCamelCase("10Primeiros");
}
```

O teste falha, já que não tratamos esta situação ainda. Para que o teste passasse, bastou adicionar um if no início do método que verifica se o primeiro caractere é um dígito:

```
public List<String> converterCamelCase(String original) {
    if (Character.isDigit(original.charAt(0))) {
        throw new IllegalArgumentException("Não pode começar com número");
    }
    String grupos = "(" +
                    sequenciaDeMinusculasNoComeco() + "|" +
                    maiusculaSeguidaDeMinusculas() + "|" +
                    maiusculasConsecutivasSemAUltimaSeAProximaForMinuscula() + "|" +
                    sequenciaDeNumeros() +
                ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return transformarEmMinusculaExcetoSigla(gruposEmLista);
}
```

O if fez com que o método ficasse com mais do que dez linhas, sendo necessário refatorá-lo. Para isso, o if foi extraído para um método e os grupos foram dispostos em um formato mais sucinto:

```
public List<String> converterCamelCase(String original) {
    validaParametro(original);
    String grupos = "(" + sequenciaDeMinusculasNoComeco() + "|" +
                          maiusculaSeguidaDeMinusculas() + "|" +
                          maiusculasConsecutivasSemAUltimaSeAProximaForMinuscula() + "|" +
                          sequenciaDeNumeros() + ")";

    String gruposSeparadoPorEspaco = original.replaceAll(grupos, "$1 ");
    List<String> gruposEmLista = Arrays.asList(gruposSeparadoPorEspaco.split(" "));
    return transformarEmMinusculaExcetoSigla(gruposEmLista);
}

private void validaParametro(String palavra) {
    if (comecaComDigito(palavra)) {
        throw new IllegalArgumentException("Não pode começar com número");
    }
}

private boolean comecaComDigito(String string) {
    return Character.isDigit(string.charAt(0));
}
```

12 - Teste com caractere especial: deve retornar erro. No caso, mais uma vez, foi escolhido lançar uma IllegalArgumentException:

```
@Test(expected=IllegalArgumentException.class)
public void possuiCaractereEspecial() {
    conversor.converterCamelCase("nome#Composto");
}
```

O teste falhou, pois não temos esta regra no código ainda. Para isso, foi necessário alterar o método que valida o parâmetro passado, mais uma vez, usando expressão regular:

```
private void validaParametro(String palavra) {
    if (comecaComDigito(palavra)) {
        throw new IllegalArgumentException("Não pode começar com número");
    } else if (possuiCaractereEspecial(palavra)) {
        throw new IllegalArgumentException("Não pode conter caractere(s) especial(is)");
    }
}

private boolean possuiCaractereEspecial(String string) {
    return !string.matches("([A-Za-z0-9])*");
}
```

13 - Ao final do exercício, com todos os testes passando, o código foi revisado mais uma vez e foi encontrado mais uma oportunidade de refatoração, diminuindo a complexidade do método “ehSigla”, além de eliminar a necessidade do método auxiliar “ehMinusculo”, que foi apagado.
Antes:

```java
private boolean ehSigla(String string) {
    for (Character caractere : string.toCharArray()) {
        if (ehMinusculo(caractere)) {
            return false;
        }
    }

    return true;
}

private boolean ehMinusculo(Character caractere) {
    return Character.isLowerCase(caractere);
}
```

Depois:

```java
private boolean ehSigla(String string) {
    return string.matches("([A-Z])*");
}
```